1. Introduction

In this project, we are going to convert the sequential Heat Distribution program provided to parallel version with the help of MPI, Pthread, OpenMP, CUDA and MPI with OpenMP. This report will give the details of the design of each version and compare their performance.

2. Methods

In the given sequential file, the program would basically calculate and update the new temperature for each point in the grid using the information of its neighbors.

a) Conduct resource-consuming computing step in parallel

Throughout the program, the part requiring the most computing resources is the calculation part where the program would iterate through all the points in the gird and use the information of its neighbors to generate new temperature for it. Actually, this calculation is perfect for parallel computing since in one iteration, the calculations for each point are independent. Thus, we can simply just assign the task to every process or thread and finally gather the calculation result from them.

b) Divide the task into several packages

Down to the details, we need a way to split the task into packages with approximately equal size. Assumed the size of the grid in grid is *size* (that is, there are totally size*size points in the grid). Assume that the total number of processes or threads in the program is *totalTaskSize*, we apply the following formula to divide the task. For process (thread) with task number ***i***, it will calculate the point with from the start index to the end index (start and end index included):

$$StartIndex_i = (\frac{size * size}{totalTaskSize}) * i \ + \ min((size * size)\%totalTaskSize, i)$$

$$EndIndex_i = (\frac{size * size}{totalTaskSize}) * (i + 1) \ + \ min((size * size)\%totalTaskSize, i + 1) - 1$$

For instance, with size equals to 20, there will be 20*20 points in the grid, three processes would proceed the points at [0,133], [134,266], [267,399] respectively. In this case, the sizes of the sub-tasks are 133, 132 and 132 separately, which are approximately same.

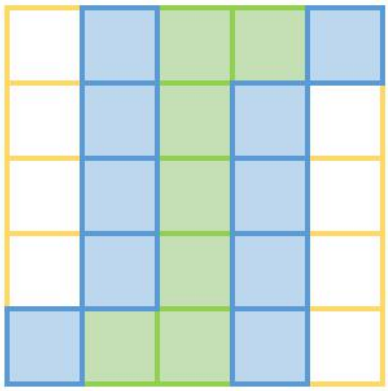
c) Mechanism of plotting with updated parameters in GUI

Notice that the calculation in the program is repeating endlessly since it always need to update the value of points after adjusting some special parameters (for example, roo_size) and replot the whole diagram. Thus, we need to make sure that other processes or threads would interact with the master process and thread correctly to compute with updated parameters.

d) Extra calculation of points in Sor

Different form Jacobi, Sor algorithm need to compute the points in the grid two rounds in each iteration and the second round need to use the information of the first round. This would be totally fine in multi-thread programming since after every thread reach the barrier at the end of the first round, all threads could easily use the calculated result of all points in the first round easily. However, in the multi-process programming, things become a little different since all process maintain their own local variables. To solve this problem, we could directly ask rank 0 to gather the

results from all processes and then broadcast all the calculation results to other processes though it will waste lot of time. Actually, an alternative way to solve this problem is to calculate the extra points in the first round to prepare for the second round. Basically, we just need to take care of the points around the calculating areas. One example to illustrate this is shown below:



Assumed the task assigned for this process is green squares, then it needs to calculate the blue squares as well in the first iteration in Sor to prepare for the second iteration.

e) Solve data race in multi-thread programs

There is one important part may cause data race in multi-thread programs, which is to check all threads' status of stabilized. To solve this problem, the program applies a vector to store every thread's status and check all of them to generate the final result. However, if directly using std::vector<bool> it may still cause some extra data race since C++ adopt a special design for vector where only one bit is used to represent a bool for the sake of saving memory. Then, there may be the chance that multiple threads are writing one byte to change their bit allocated for them which is then the data race. To solve this problem, std::vector<int> instead is used as alternative.

d) MPI

Since we use rank 0 process as the master program which generate GUI and fetch the updated parameters from users, we need to make sure rank 0 pass necessary information to other ranks to do the corresponding calculation work. Following procedures are designed to make sure each process does the calculation correctly when doing calculation.

Step 1: Rank 0 broadcast the state information of the current grid to each process.

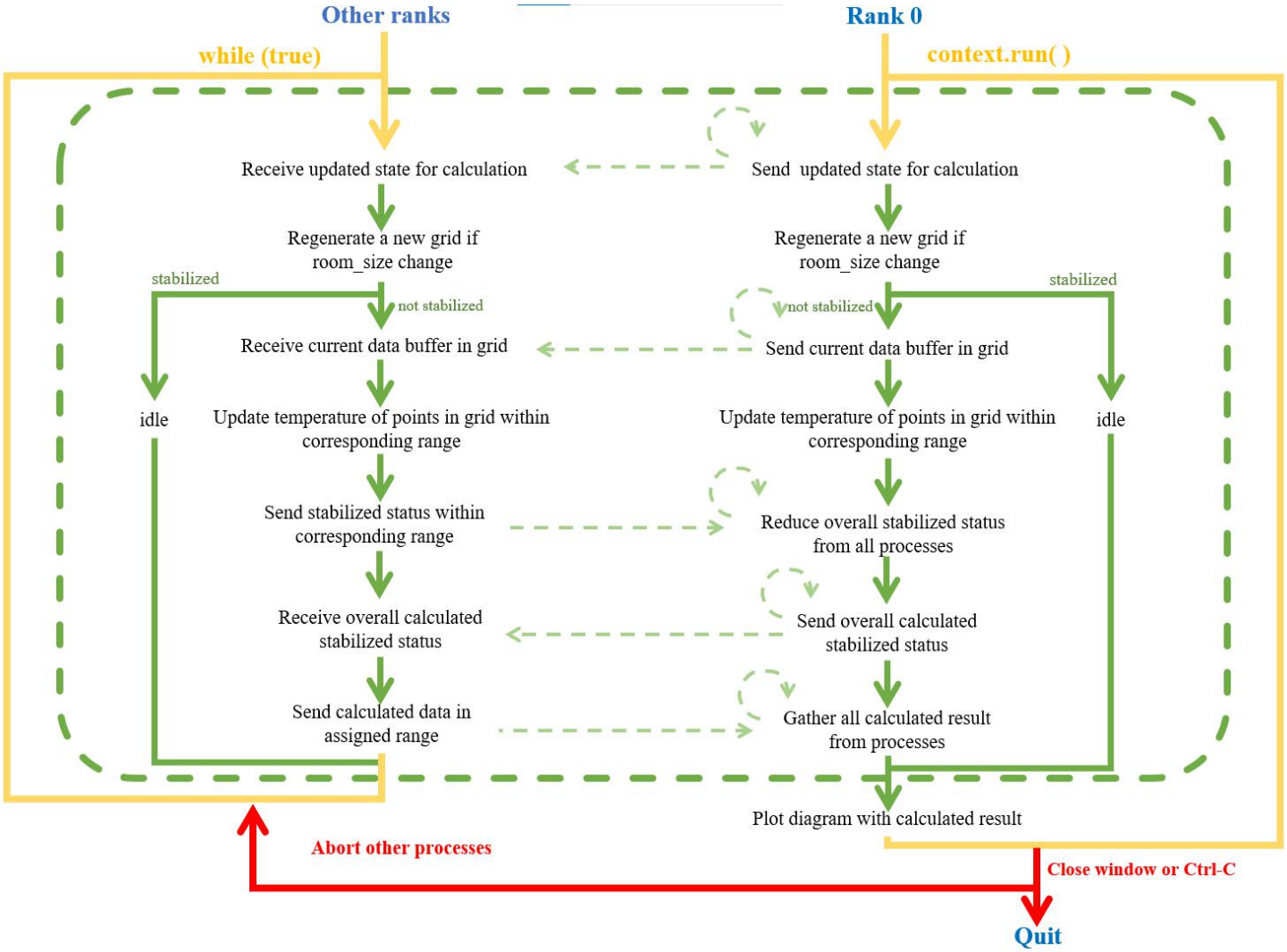
Step 2: All processes received the state information from rank 0. Then, all processes will check whether the room size has changed, if so, they will regenerate a new grid.

Step 3: All process would calculate the next temperature within their corresponding range.

Step 4: Rank 0 would reduce the stabilized information of each process local stabilized situation and generate the overall stabilized result which would then broadcast to all other processes.

Step 5: Rank 0 would gather the calculated result from all other processes after which the Rank 0 would plot the heat grid. Next, all process would go to step 1.

Step 6: If GUI window is closed or Ctrl-C is typed in the terminal, the rank 0 process would directly jump to step 7 and abort all other processes. In this case, you may see the abort signal prompted in the terminal which is a normal operation.



f) Pthread/OpenMP/CUDA

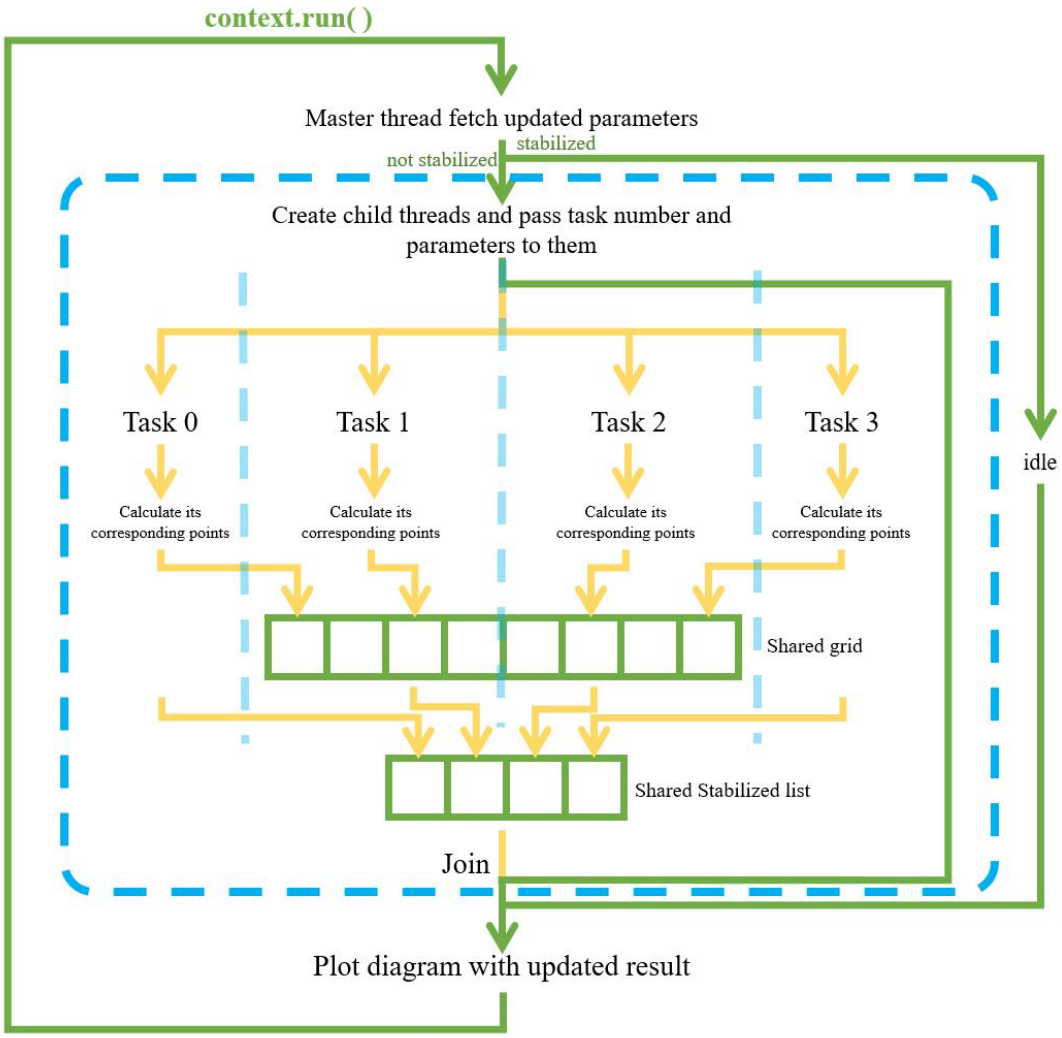
Besides using multi-processes, we can also use multi-threads to finish the calculation task. In this project, the Pthread, OpenMP and CUDA are implemented mainly follow the same logic.

With multiple threads, updating the diagram with new parameters would be much easier since we can directly pass parameters to threads when creating them. Waiting for all the threads finish their calculation task, the master thread would then plot the diagram directly. Followings are the main steps:

Step 1: Master program would create several threads and pass the corresponding parameters to them.

Step 2: All threads do the calculation next temperature on their corresponding regions. After that, they will write their local stabilized status into their corresponding positions on the status list.

Step 3: After joining all the threads, the master thread would then go through all the local status in the status list to generate the overall stabilized status. Finally, it will plot the heat grid and jump to step 1 again.



g) MPI with OpenMP

We can actually combine the multi-process and multi-thread mechanism together, that is, launch several processes and create multiple threads to finish the tasks assigned to each process. Followings are the detailed steps which is just a simple combination of MPI and OpenMP:

Step 1: Rank 0 broadcast the state information of the current grid to each process.

Step 2: All processes received the state information from rank 0. Then, all processes will check whether the room size has changed, if so, they will regenerate a new grid.

Step 3: All process would calculate the next temperature within their corresponding range.

Step 4: Master program in each process would create several threads and pass the corresponding parameters to them.

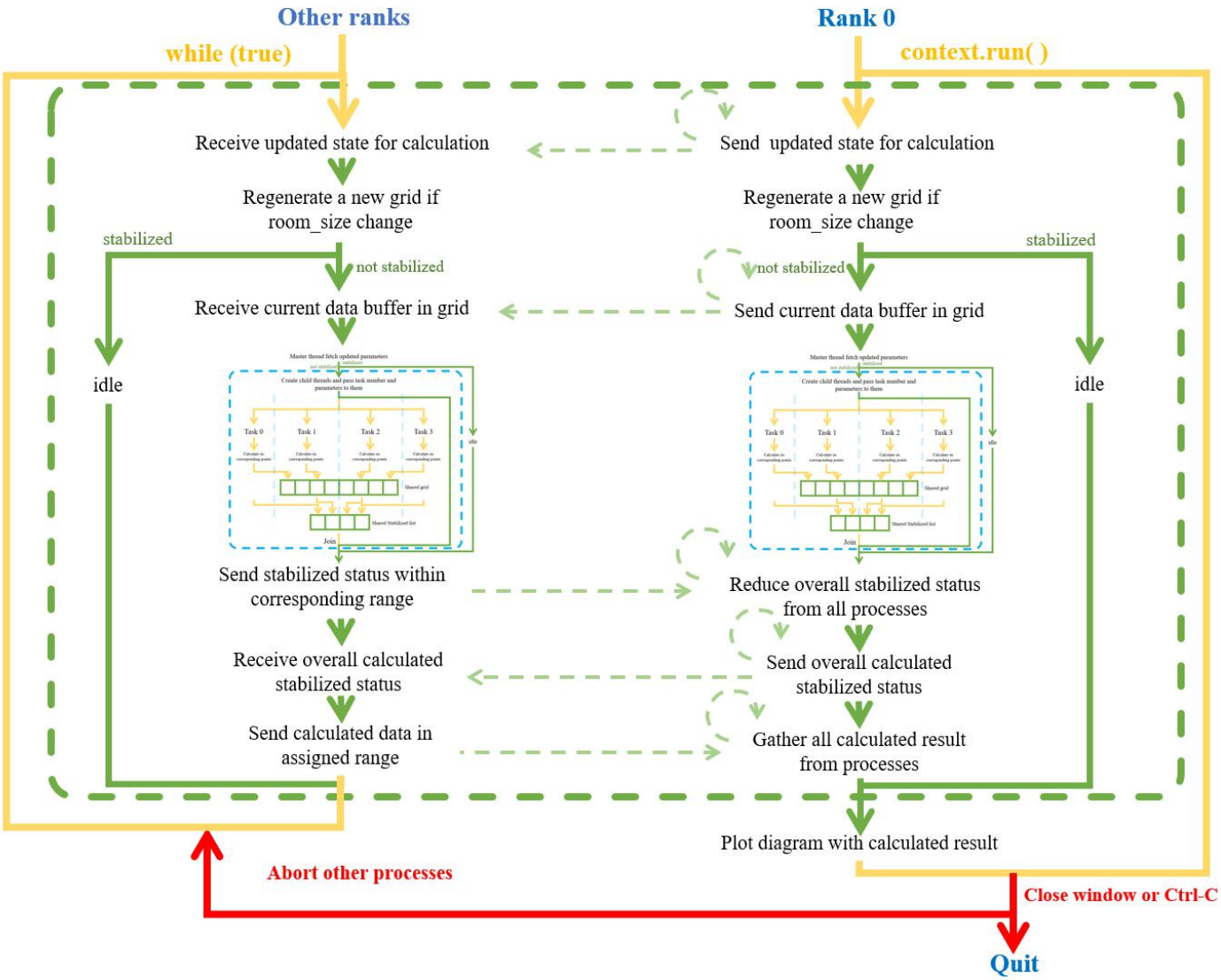
Step 5: All threads do the calculation next temperature on their corresponding regions. After that, they will write their local stabilized status into their corresponding positions on the status list in each process.

Step 6: After joining all the threads, the master thread would then go through all the local status in the status list to generate the overall stabilized status of one process.

Step 4: Rank 0 would reduce the stabilized information of each process local stabilized situation and generate the overall stabilized result which would then broadcast to all other processes.

Step 5: Rank 0 would gather the calculated result from all other processes after which the Rank 0 would plot the heat grid. Next, all process would go to step 1.

Step 6: If GUI window is closed or Ctrl-C is typed in the terminal, the rank 0 process would directly jump to step 7 and abort all other processes. In this case, you may see the abort signal prompted in the terminal which is a normal operation.



3. Program execution

a) Compile and build

First, you need to Put program folder into "/pvfsmnt/(student_id)". And you need to type "rm -rf build" to remove the previous build folder.

Then follow the procedures to compile and build different version of programs:

i. MPI, Pthread, OpenMPI, MPI with OpenMP

1) Type "mkdir build"
2) Type "cd build"
3) Type "cmake .. -DCMAKE_BUILD_TYPE=Debug"
4) Type "cmake --build . -j4"

ii. CUDA

1) Type "mkdir build"
2) Type "cd build"
3) Type "source scl_source enable devtoolset-10"
4) Type "CC=gcc CXX=g++ cmake .."
5) Type "make -j12"

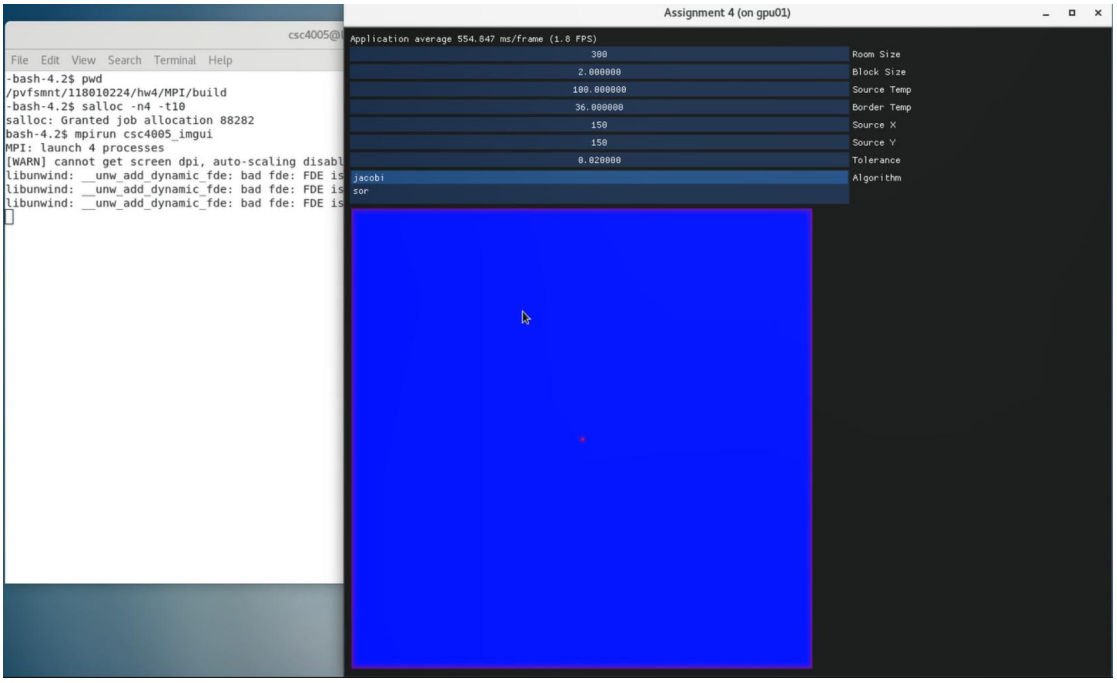
Then, you need to do some authorization settings before executing the programs. In a suitable environment (all the executions of programs in this project are done in the virtual machine set up in

tutorial 1), type “xhost +” first. Then connect to the server using “ssh -Y {student id}@10.26.1.30”. Then type “cd /pvfsmnt/$(whoami)”, “cp ~/.Xauthority /pvfsmnt/$(whoami)”, “export XAUTHORITY=/pvfsmnt/$(whoami)/.Xauthority”.

b) MPI

Since program run with multiple processes, you first need to allocate several cores. Type “salloc -n4 -t10” to request 4 cores for example. Then type “mpirun csc4005_imgui” in the build folder then it will prompt out the program.

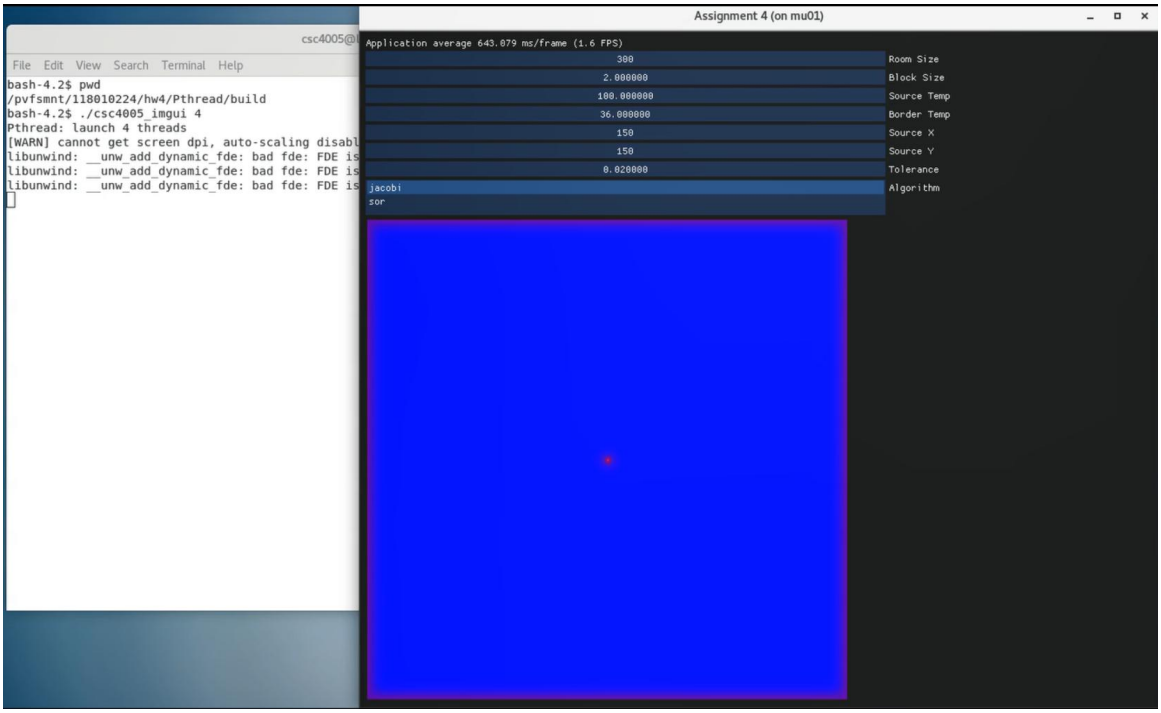
Running result:



c) Pthread

The steps to execute the pthread version program is relative easier, you just need to type “./csc4005_imgui [number of threads]” directly, for example, type “./csc4005_imgui 4” for execute program in 4 threads. The default setting is 4, which would be set up automatically if the input parameters is invalid or not given.

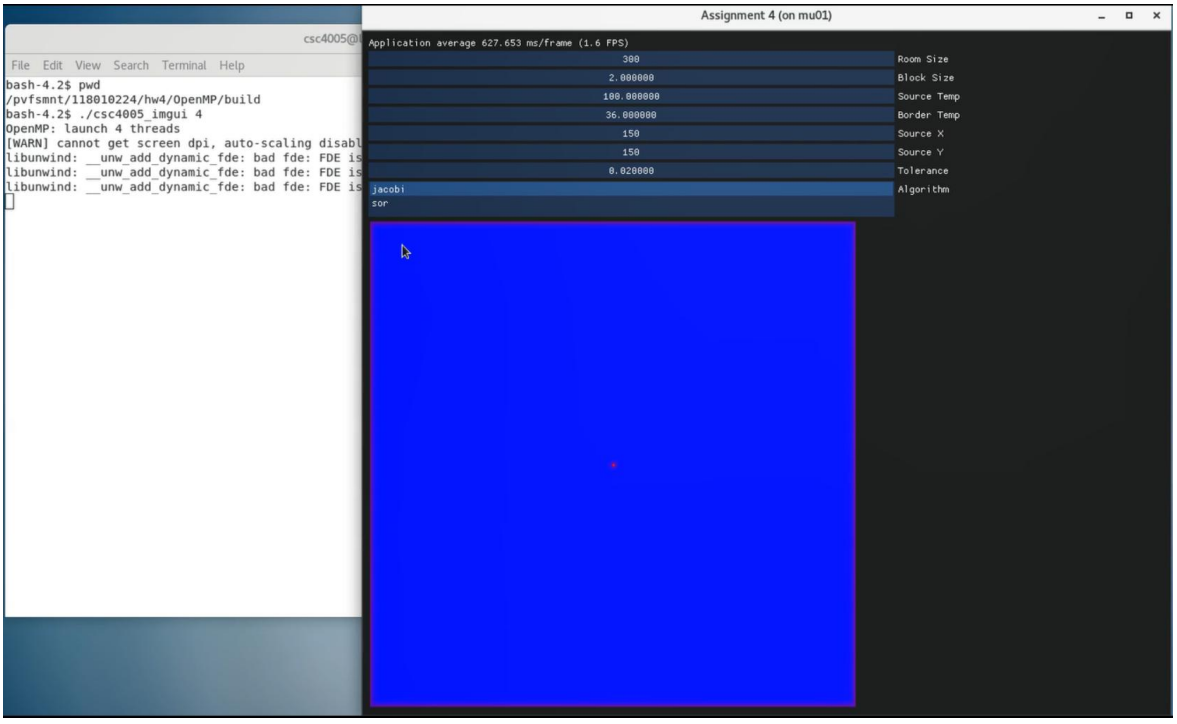
Running result:



d) OpenMP

The steps to execute the OpenMP version program is same as Pthread, you just need to type "./csc4005_imgui [number of threads]" directly, for example, type "./csc4005_imgui 4" for execute program in 4 threads. The default setting is 4, which would be set up automatically if the input parameters is invalid or not given.

Running result:

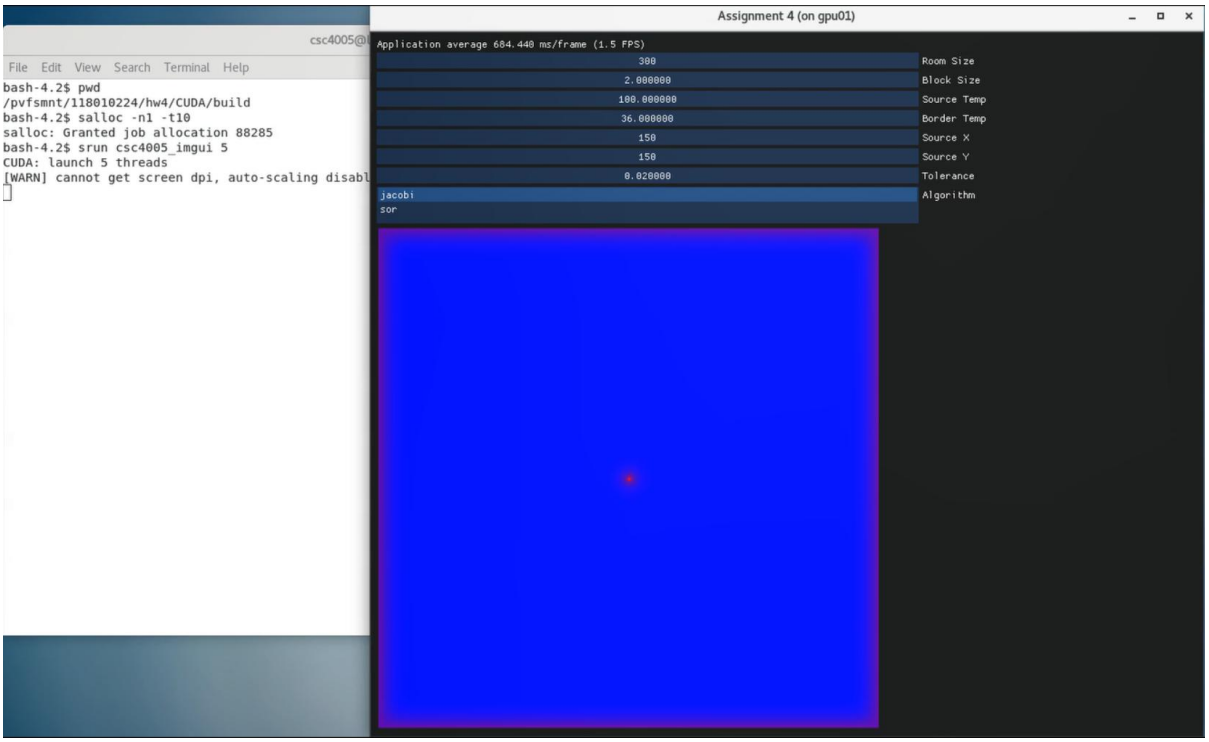


e) CUDA

The steps to execute the OpenMP version program is a little different. You first need to type "salloc -n1 -t10" to obtain one GPU node. Then you need to type "srun ./csc4005_imgui [number of

threads]" directly, for example, type "./csc4005_imgui 5" for execute program with the CUDA settings as 5 threads in one block. The default setting for all value is 4, which would be set up automatically if the input parameters is invalid or not given.

Running result:

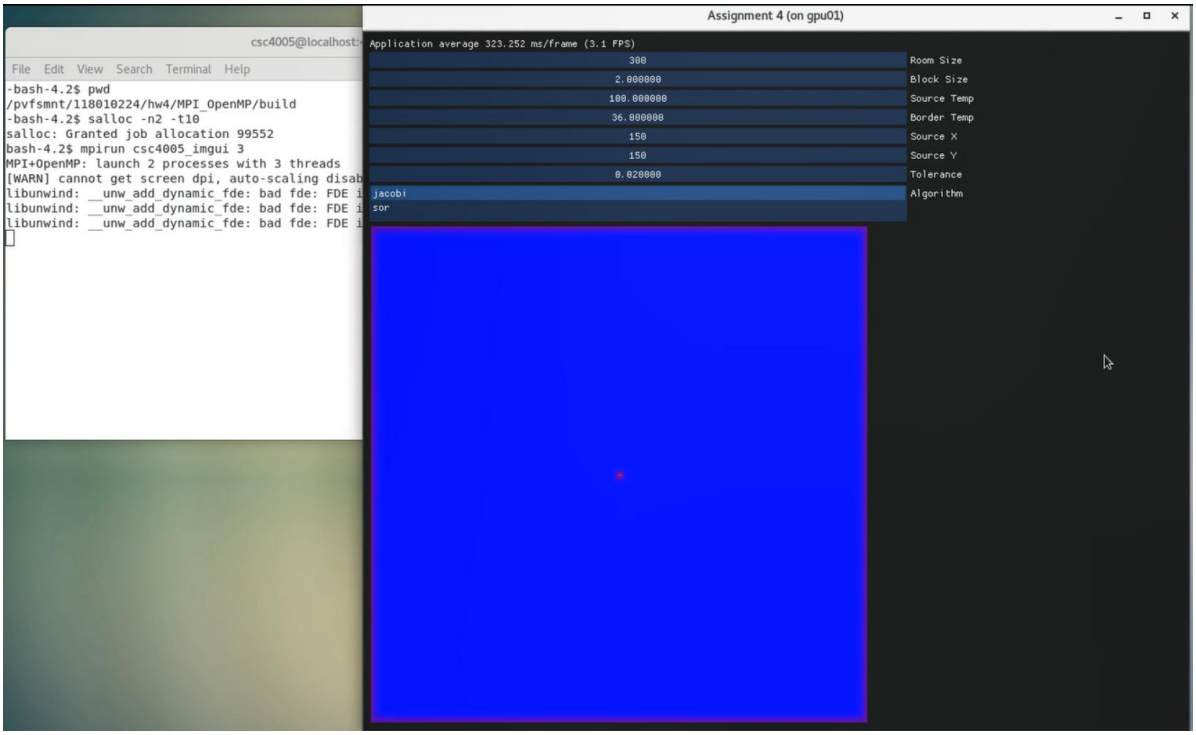


(Notice that the "srun error" in the snapshot is the error of allocating nodes in cluster, which has no relation with the codes)

f) MPI with OpenMP

Since program run with multiple processes, you first need to allocate several cores. Type "salloc -n2 -t10" to request 2 cores for example. Then you need to type "mpirun ./csc4005_imgui [number of threads in each process]". The default setting for number of threads in each process is 3, which would be set up automatically if the input parameters is invalid or not given.

Running result:



4. Performance analysis
   a) Time complexity

   In this analysis, we only consider the calculation task for one iteration. Also, we ignore the plotting steps.

   i. Sequential

   Assume the total number of points in the grid is *N*. For Jacobi algorithm, basically, one iteration would simply go through all the points and sum the current values of its 4 neighbors. Overall, it will take $O(4N) = O(N)$ time to finish the job.

   ii. MPI

   Assume the total number of points in the gird is *N* and there are *p* processes. For Jacobi, as for the communication part, the program would take $O(N)$ to broadcast its current data buffer and $O(p)$ to reduce the stabilized results in every process and $O(1)$ to broadcast the reduced result. Finally, it will take $O(N)$ to gather result. Therefore, we have $T_{comm} = O(N) + O(p) + O(1) + O(N) = O(N)$. Then for computation, each process would only need to take care of $\frac{N}{p}$ points which means $T_{comp} = O\left(4 * \frac{N}{p}\right) = O\left(\frac{N}{p}\right)$. Finally, the result would be $T = T_{comm} + T_{comp} = O(N) + O\left(\frac{N}{p}\right) = O(N)$. For Sor, the situation is nearly the same which has same time complexity as $O(N)$.

   iii. Pthread/OpenMP/CUDA

   Assume the total number of points in the grid is *N* and there are *t* threads. For Jacobi, first, it may take $O(t)$ to create *t* threads. With *t* threads running in parallel, each thread would only need to take $O\left(\frac{N}{t}\right)$ to do the calculation. Finally, master thread would take $O(t)$ to check all

stabilized situation in each thread to generate the overall stabilized status. Overall, we have $T = O\left(\frac{N}{t}\right) + O(t)$. The time complexity for Sor is quite similar which is also $T = O\left(\frac{N}{t}\right) + O(t)$. For CUDA, the time complexity for Jacobi and Sor would be $T = O\left(\frac{N}{t}\right) + O(t) + O(N) = O(N)$ since it would take extra time to copy and update the data buffer in grid.

iv. MPI with OpenMP

Assume the total number of points in the grid is *N* and there are *p* processes and each process create *t* threads. In this case, we just replace the computation time in MPI with the time complexity of OpenMP then we get the overall time complexity which is $T = T_{comm} + T_{comp} = O(N) + O\left(\frac{N}{pt}\right) + O(t) + O(N) = O(N)$.

b) Data race test in Pthread and OpenMP

Notice that all data race tests are conducted on virtual machine provided in the tutorials.

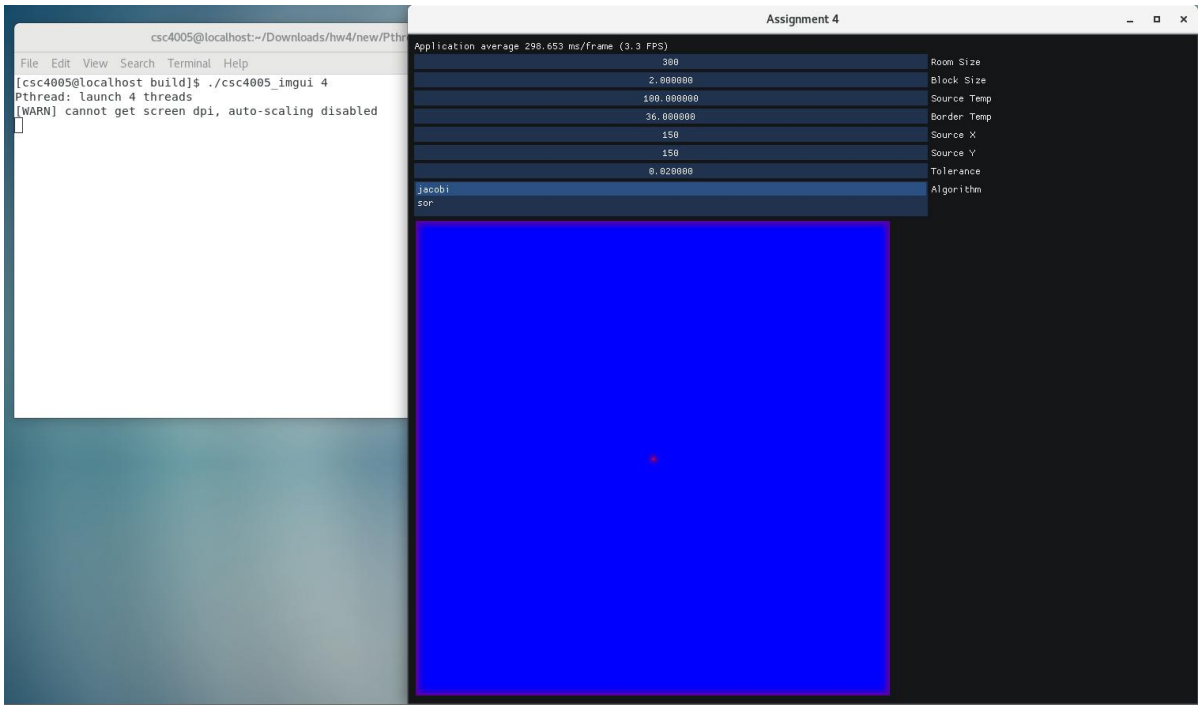
i. Pthread

To activate the data race mode, you need to do some modifications in the CMakeList.txt, which is shown in the followings:

Pthread > CMakeLists.txt

```
cmake_minimum_required(VERSION 3.2)
project(csc4005_imgui)

find_package(SDL2 REQUIRED)
find_package(Freetype REQUIRED)
find_package(MPI REQUIRED)
find_package(Threads REQUIRED)
set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_EXTENSIONS ON)
set(OpenGL_GL_PREFERENCE "GLVND")

set(CMAKE_CXX_FLAGS "-fsanitize=thread")

find_package(OpenGL REQUIRED)

include_directories(
        include
        imgui
        imgui/backends
        ${SDL2_INCLUDE_DIRS}
        ${FREETYPE_INCLUDE_DIRS}
        ${MPI_CXX_INCLUDE_DIRS})

file(GLOB IMGUI_SRC
        imgui/*.cpp
        imgui/backends/imgui_impl_sdl.cpp
        imgui/backends/imgui_impl_opengl2.cpp
        imgui/misc/freetype/imgui_freetype.cpp
        imgui/misc/cpp/imgui_stdlib.cpp
        )
add_library(core STATIC ${IMGUI_SRC})
file(GLOB CSC4005_PROJECT_SRC src/*.cpp src/*.c)
add_executable(csc4005_imgui ${CSC4005_PROJECT_SRC})
get_filename_component(FONT_PATH imgui/misc/fonts/DroidSans.ttf ABSOLUTE)
target_link_libraries(core PUBLIC
        Freetype::Freetype SDL2::SDL2 OpenGL::GL ${CMAKE_DL_LIBS} Threads::Threads ${MPI_CXX_LIBRARIES})
```

Running result:



ii. OpenMP

First, you need to do some modifications in CMakeList.txt before compile and build, which is shown in the followings:

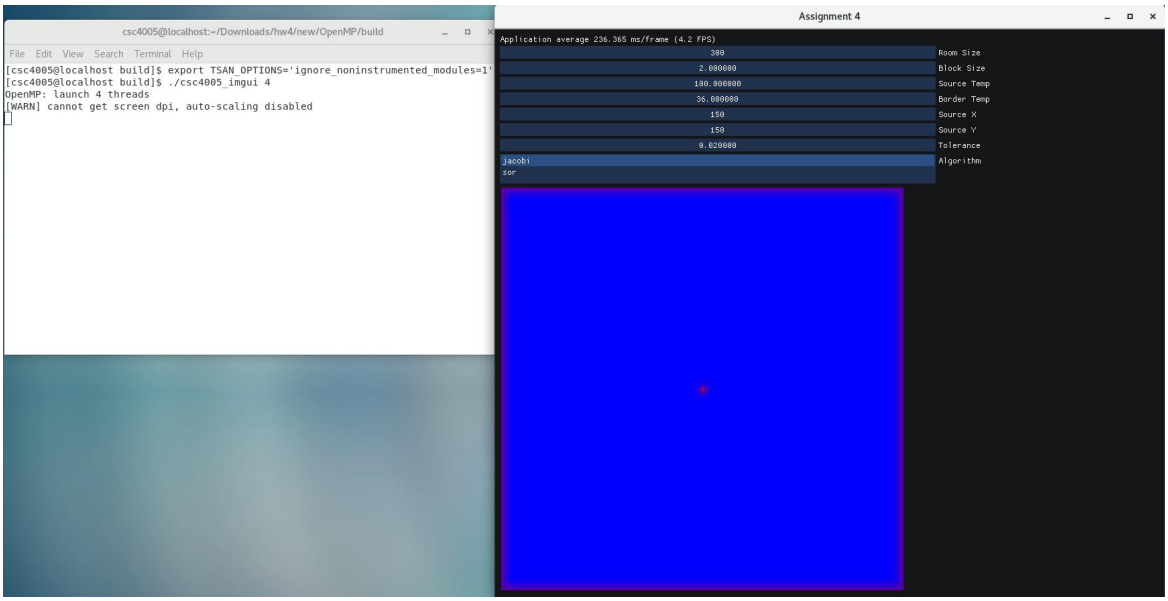
CMakeLists.txt

OpenMP > CMakeLists.txt

```
find_package(SDL2 REQUIRED)
find_package(Freetype REQUIRED)
find_package(MPI REQUIRED)
find_package(Threads REQUIRED)
set(CMAKE_CXX_STANDARD 20)
set(CMAKE_CXX_EXTENSIONS ON)
set(OpenGL_GL_PREFERENCE "GLVND")
find_package(OpenGL REQUIRED)

find_package(OpenMP REQUIRED)
set (CMAKE_C_FLAGS "${CMAKE_C_FLAGS} ${OpenMP_C_FLAGS}")
# set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS}")
set (CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} ${OpenMP_CXX_FLAGS} -fsanitize=thread")
set (CMAKE_EXE_LINKER_FLAGS "${CMAKE_EXE_LINKER_FLAGS} ${OpenMP_EXE_LINKER_FLAGS}")


include_directories(
        include
        imgui
        imgui/backends
        ${SDL2_INCLUDE_DIRS}
        ${FREETYPE_INCLUDE_DIRS}
        ${MPI_CXX_INCLUDE_DIRS})

file(GLOB IMGUI_SRC
        imgui/*.cpp
        imgui/backends/imgui_impl_sdl.cpp
        imgui/backends/imgui_impl_opengl2.cpp
        imgui/misc/freetype/imgui_freetype.cpp
        imgui/misc/cpp/imgui_stdlib.cpp
        )
add_library(core STATIC ${IMGUI_SRC})
file(GLOB CSC4005_PROJECT_SRC src/*.cpp src/*.c)
add_executable(csc4005_imgui ${CSC4005_PROJECT_SRC})
get_filename_component(FONT_PATH imgui/misc/fonts/DroidSans.ttf ABSOLUTE)
target_link_libraries(core PUBLIC
        Freetype::Freetype SDL2::SDL2 OpenGL::GL ${CMAKE_DL_LIBS} Threads::Threads ${MPI_CXX_LIBRARIES})
```

Before running the program, you need to type “ export TSAN_OPTIONS='ignore_noninstrumented_modules=1' ” to avoid some extra false positive errors.

Running result:



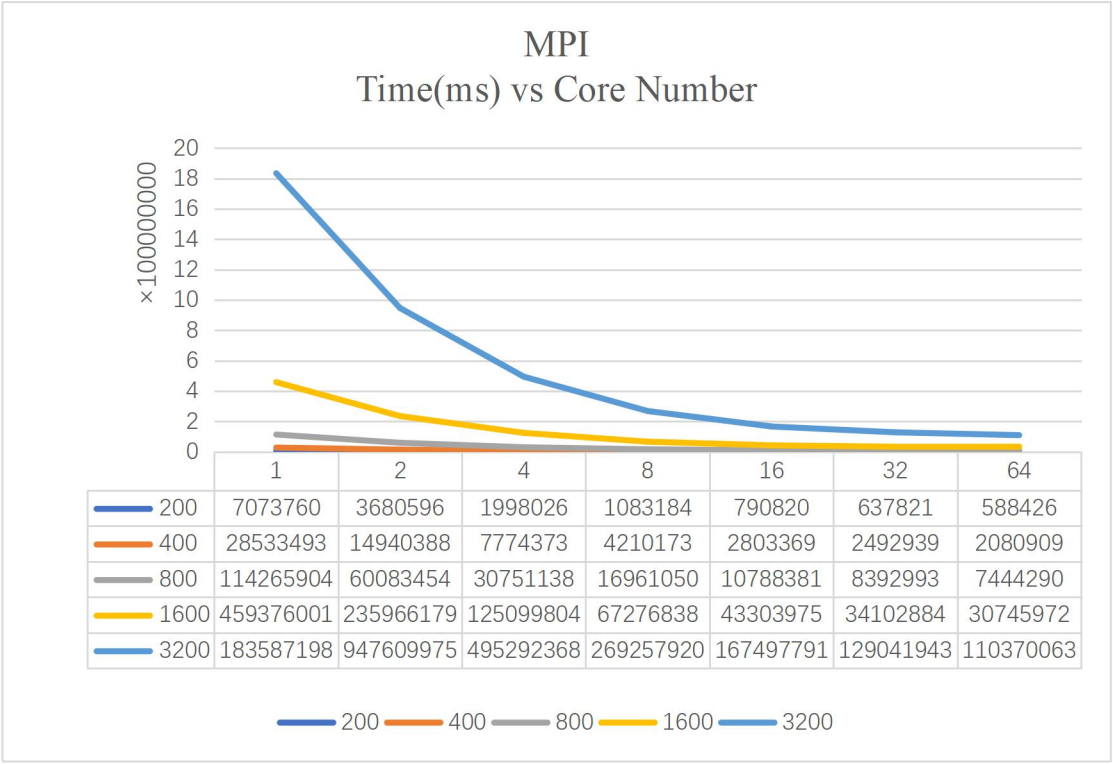
c) Performance Test

In this project, MPI, Pthread, OpenMP, CUDA and MPI with OpenMP would work in the same way as sequential version if total number of process or thread is 1. Thus, for testing, all versions with one process would work as the sequential version in their test group. All the test results generated from all versions would only focus on the calculation steps. Notice that we only consider one iteration of calculation instead of waiting the grid to be stabilized since all calculation are almost same in each iteration.

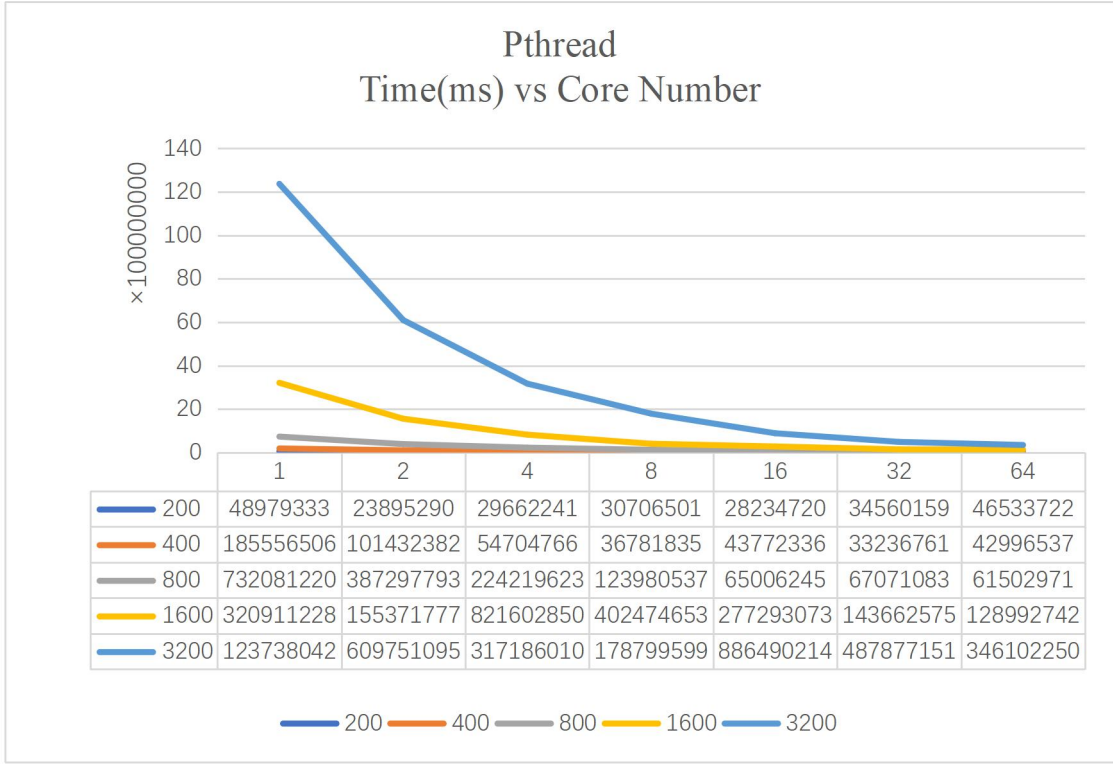
i. Time vs number of Core/Thread

Notice for testing, MPI will use one node for 1 process test and 2 nodes for all the remaining tests. For CUDA, we will only use one node. For MPI with OpenMP, we will only use one node for totally one thread test and two nodes for the remaining test.

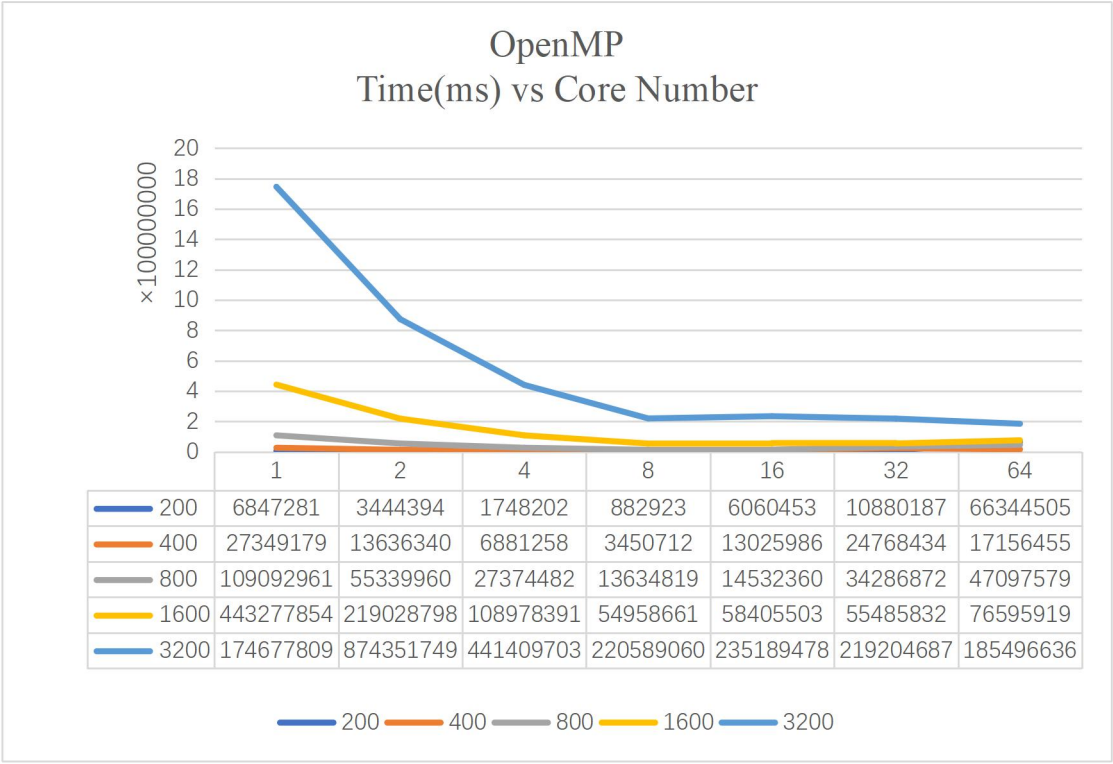
Part 1. All test sets:



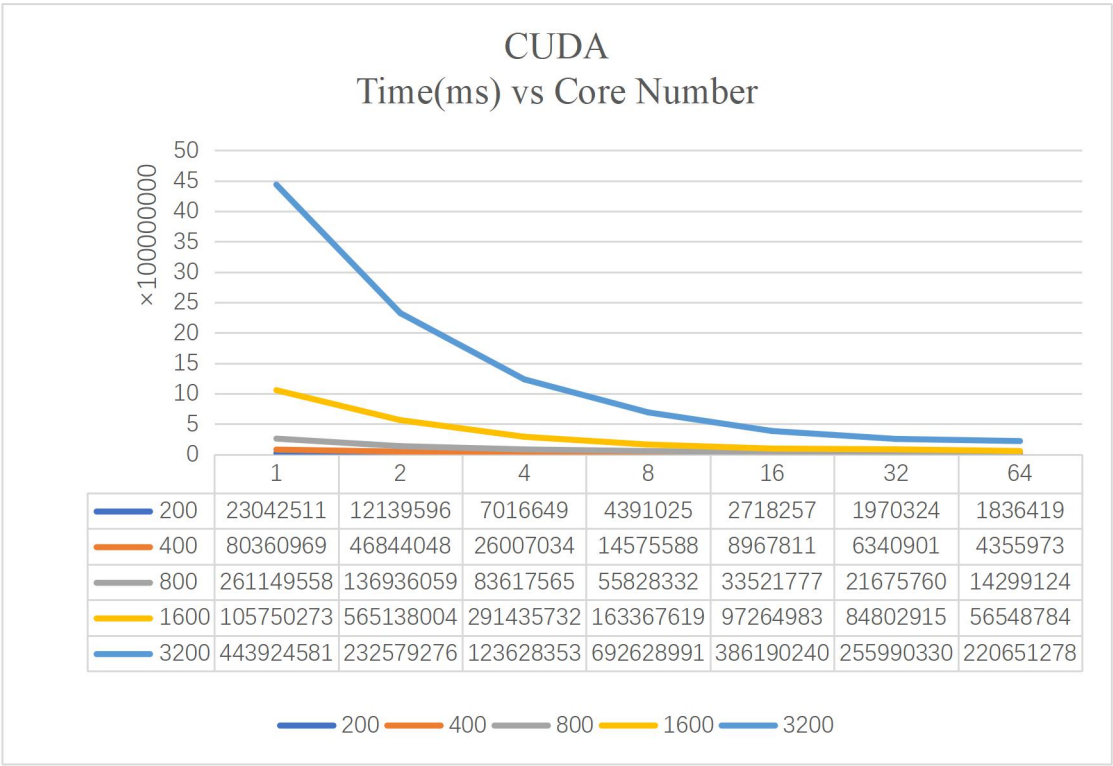
	1	2	4	8	16	32	64
200	7073760	3680596	1998026	1083184	790820	637821	588426
400	28533493	14940388	7774373	4210173	2803369	2492939	2080909
800	114265904	60083454	30751138	16961050	10788381	8392993	7444290
1600	459376001	235966179	125099804	67276838	43303975	34102884	30745972
3200	183587198	947609975	495292368	269257920	167497791	129041943	110370063



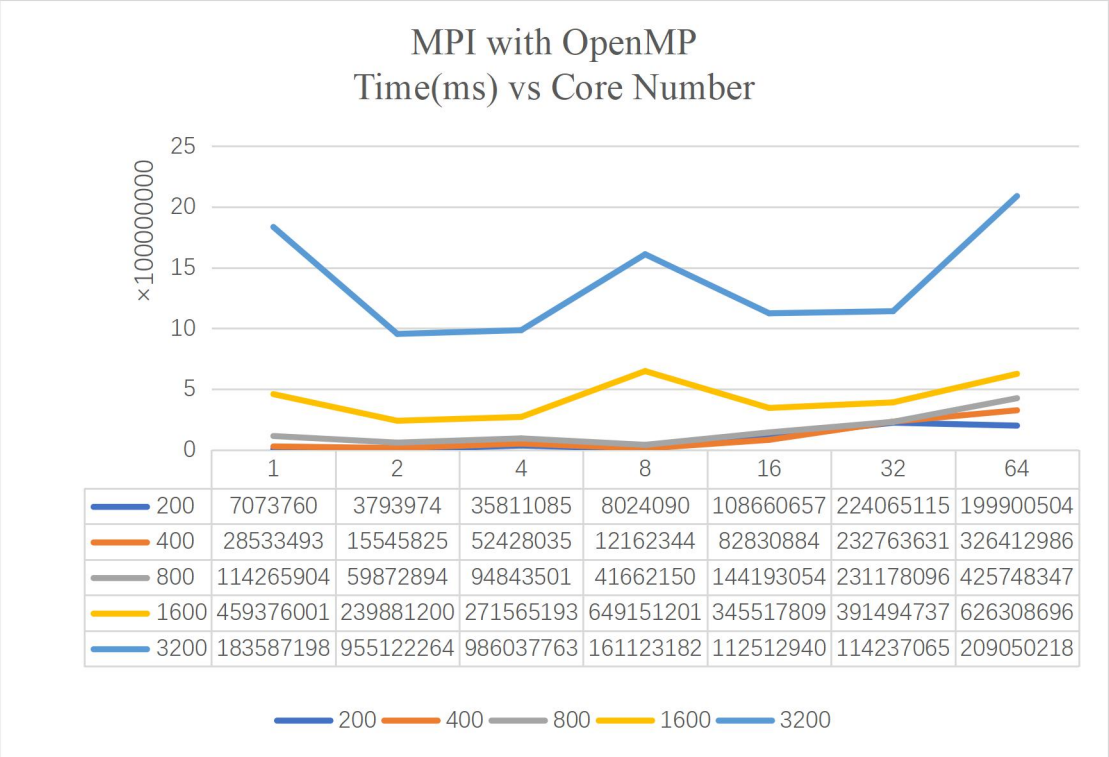
	1	2	4	8	16	32	64
200	48979333	23895290	29662241	30706501	28234720	34560159	46533722
400	185556506	101432382	54704766	36781835	43772336	33236761	42996537
800	732081220	387297793	224219623	123980537	65006245	67071083	61502971
1600	320911228	155371777	821602850	402474653	277293073	143662575	128992742
3200	123738042	609751095	317186010	178799599	886490214	487877151	346102250



	1	2	4	8	16	32	64
200	6847281	3444394	1748202	882923	6060453	10880187	66344505
400	27349179	13636340	6881258	3450712	13025986	24768434	17156455
800	109092961	55339960	27374482	13634819	14532360	34286872	47097579
1600	443277854	219028798	108978391	54958661	58405503	55485832	76595919
3200	174677809	874351749	441409703	220589060	235189478	219204687	185496636



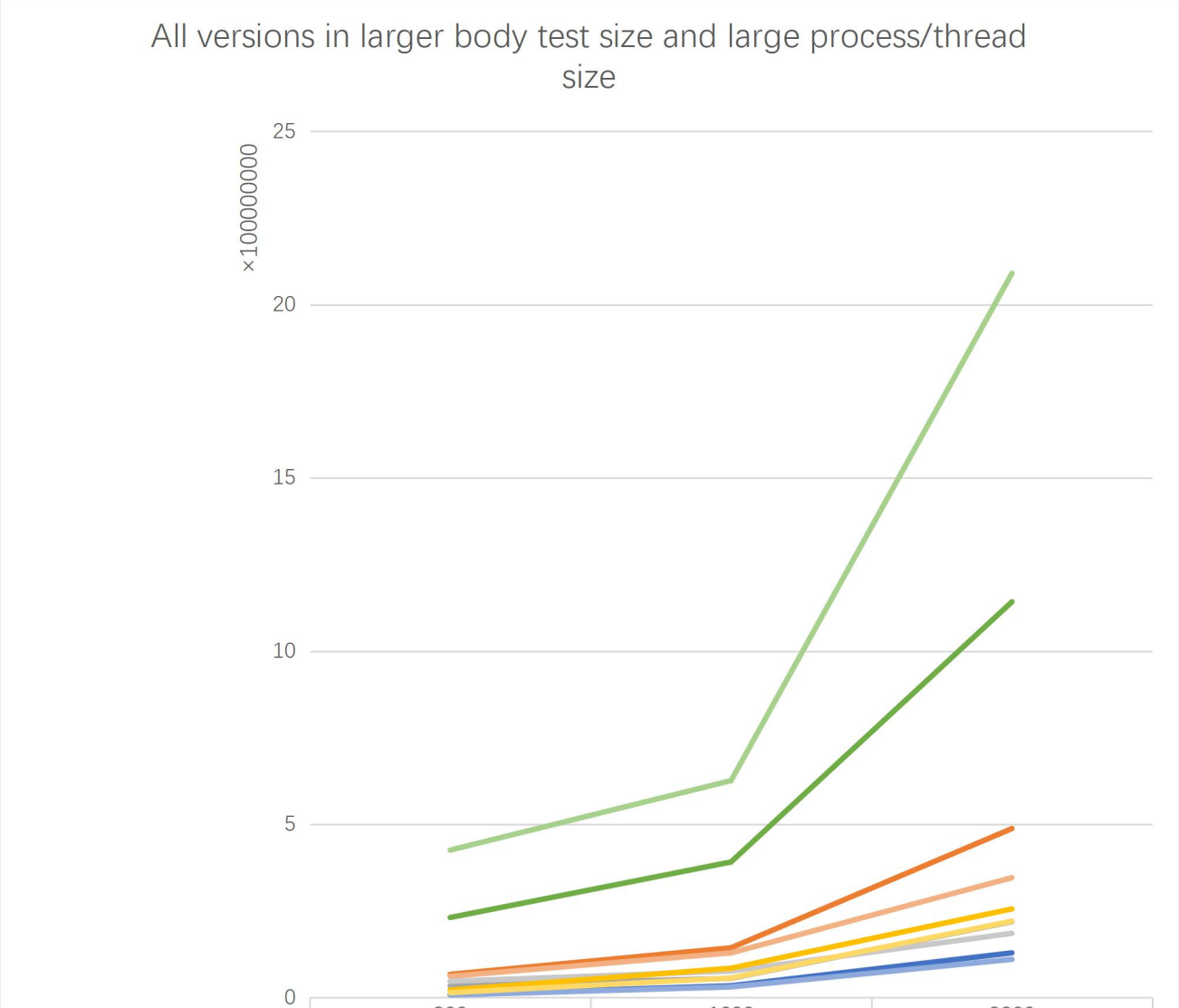
	1	2	4	8	16	32	64
200	23042511	12139596	7016649	4391025	2718257	1970324	1836419
400	80360969	46844048	26007034	14575588	8967811	6340901	4355973
800	261149558	136936059	83617565	55828332	33521777	21675760	14299124
1600	105750273	565138004	291435732	163367619	97264983	84802915	56548784
3200	443924581	232579276	123628353	692628991	386190240	255990330	220651278



	1	2	4	8	16	32	64
200	7073760	3793974	35811085	8024090	108660657	224065115	199900504
400	28533493	15545825	52428035	12162344	82830884	232763631	326412986
800	114265904	59872894	94843501	41662150	144193054	231178096	425748347
1600	459376001	239881200	271565193	649151201	345517809	391494737	626308696
3200	183587198	955122264	986037763	161123182	112512940	114237065	209050218

Generally, with more processes or threads involves, the execution time are further decreasing. Except MPI with OpenMP, the testing result shows that double the number of processes or threads would decrease the execution time by half, which indicating the programs are cost optimum. Things are a little different in MPI with OpenMP where the running time fluctuate with the increase of threads. One of the possible reasons is the communication overhead and threads creation effect offset the speedup bring up by parallel computing. Also, we can see that a good performance was generally reached when we have relatively small number of threads.

Part 2. All versions in larger body test size and large process/thread size



	800	1600	3200
MPI 32	8392993	34102884	129041943
MPI 64	7444290	30745972	110370063
Pthread 32	67071083	143662575	487877151
Pthread 64	61502971	128992742	346102250
OpenMP 32	34286872	55485832	219204687
OpenMP 64	47097579	76595919	185496636
CUDA 32	21675760	84802915	255990330
CUDA 64	14299124	56548784	220651278
MPI&OpenMP 32	231178096	391494737	1142370652
MPI&OpenMP 64	425748347	626308696	2090502187

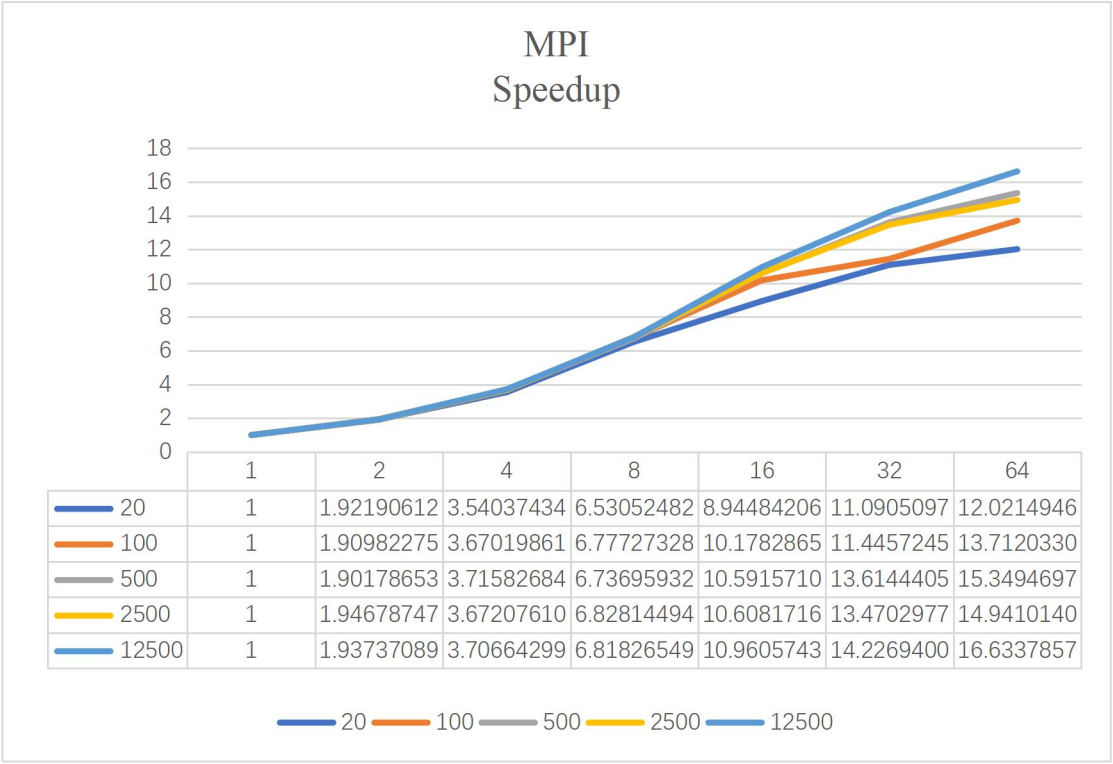
In the test with large data size and large thread or process size, the MPI with OpenMP generally works the worst among all which may be attributed to the communication time and the threads creation and management time. Among all, MPI performs the best, one of the possible reasons may be the relative powerful computing ability of each process.

Part 3. Speedup

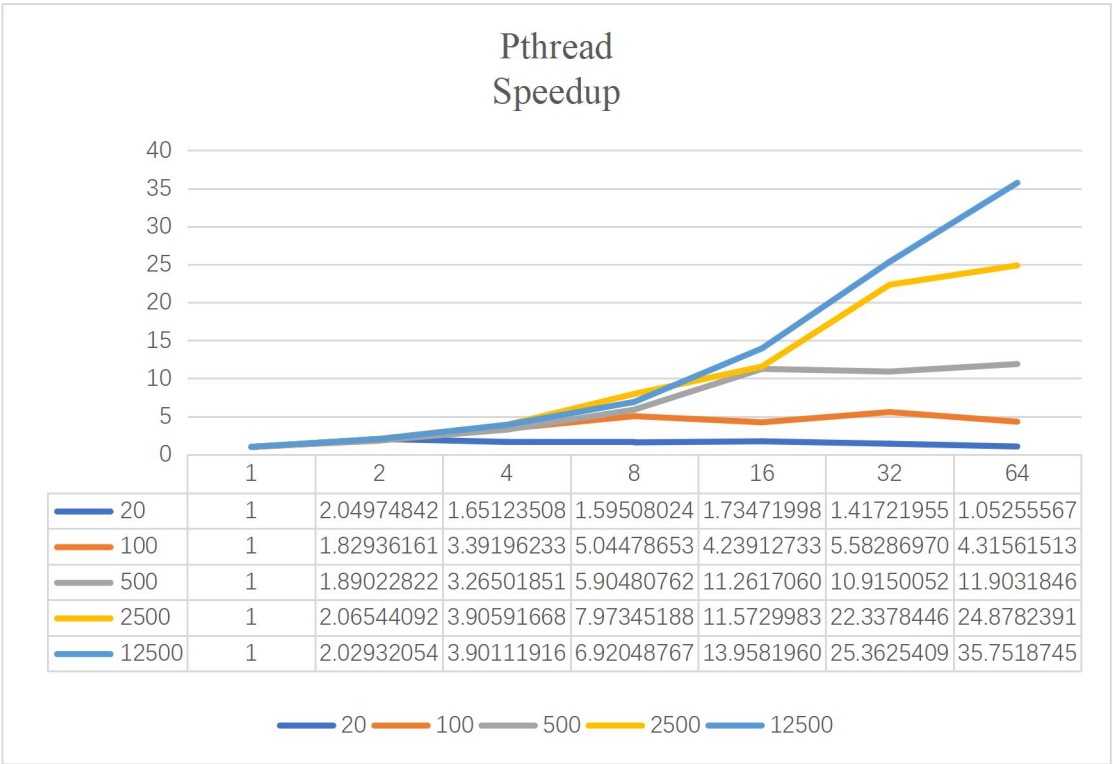
We can also calculate the Speedup of the program.

The formula of the Speedup is given as:

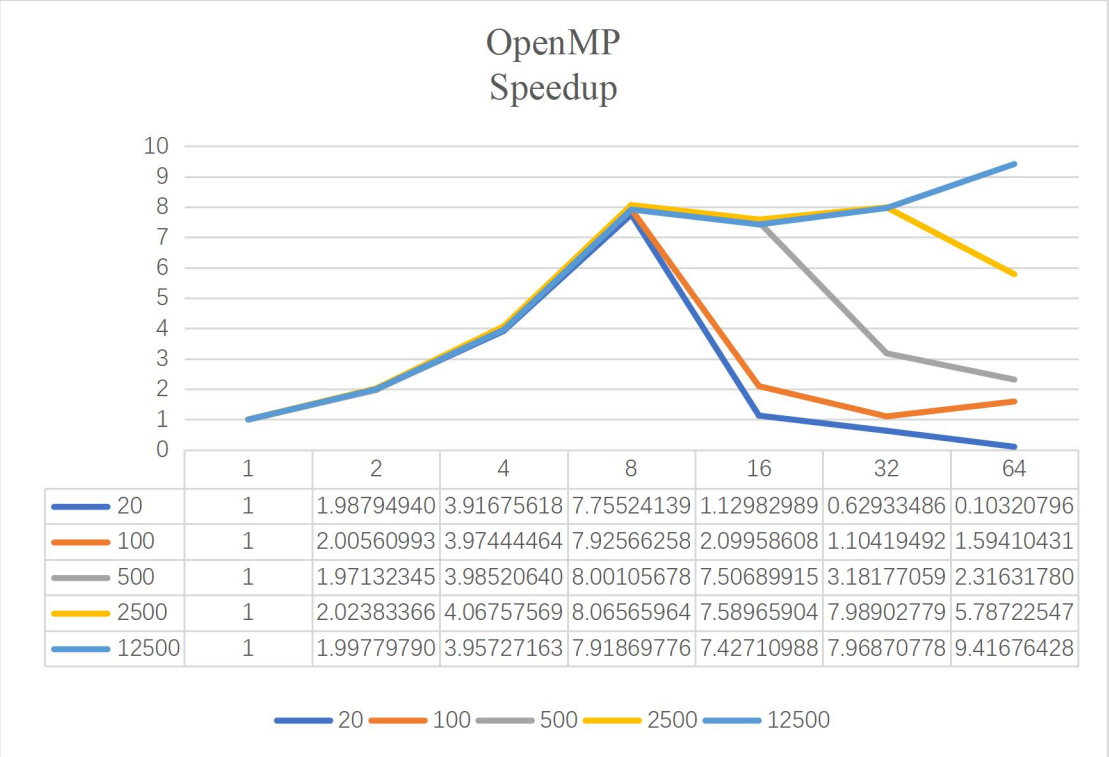
$$Speedup = \frac{running\ time\ of\ sequentail\ version}{running\ time\ of\ parallel\ version}$$



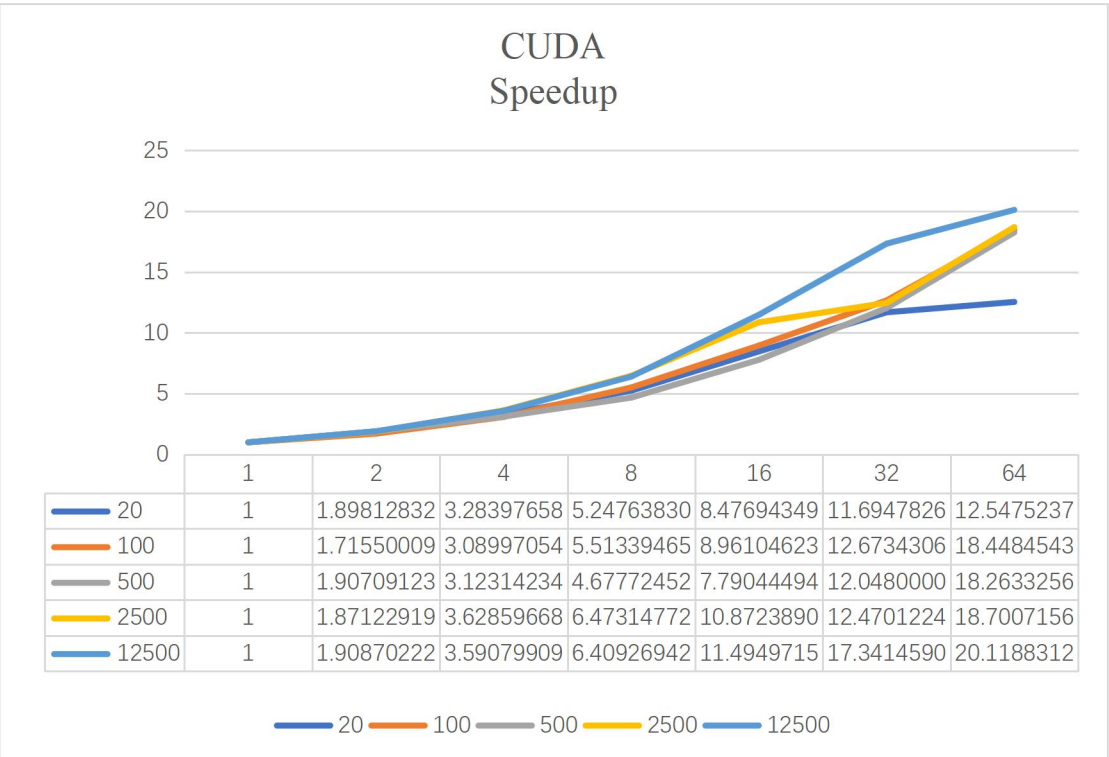
	1	2	4	8	16	32	64
20	1	1.92190612	3.54037434	6.53052482	8.94484206	11.0905097	12.0214946
100	1	1.90982275	3.67019861	6.77727328	10.1782865	11.4457245	13.7120330
500	1	1.90178653	3.71582684	6.73695932	10.5915710	13.6144405	15.3494697
2500	1	1.94678747	3.67207610	6.82814494	10.6081716	13.4702977	14.9410140
12500	1	1.93737089	3.70664299	6.81826549	10.9605743	14.2269400	16.6337857



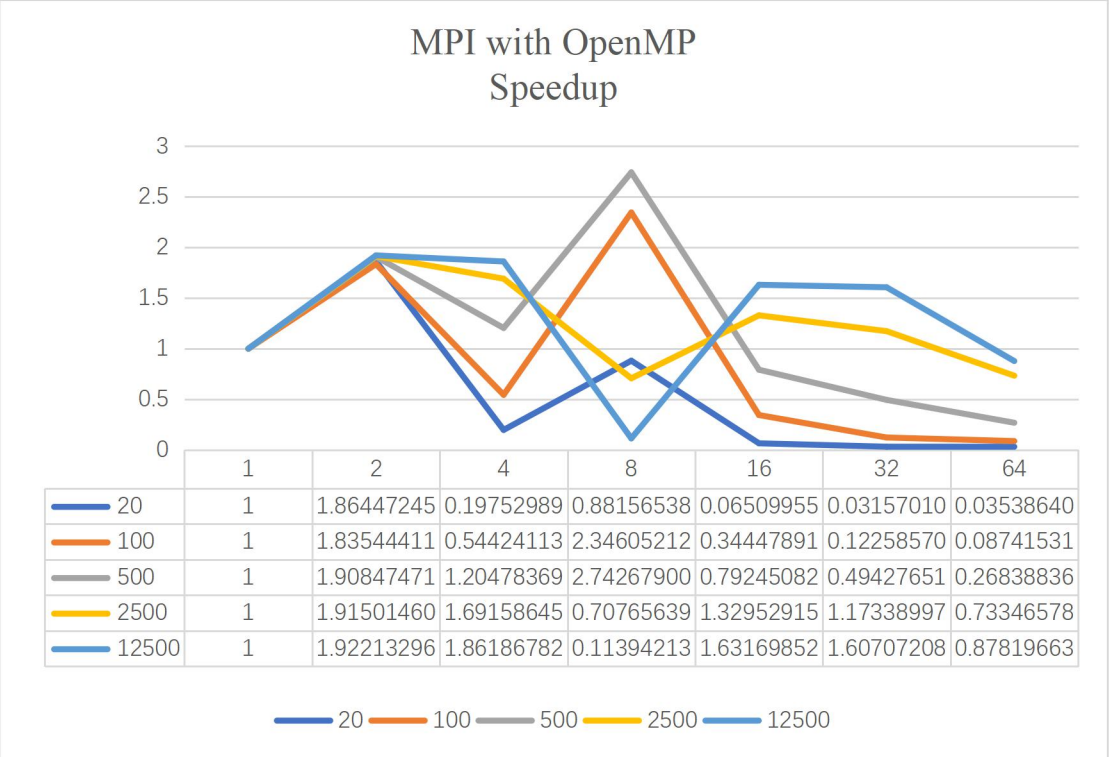
	1	2	4	8	16	32	64
20	1	2.04974842	1.65123508	1.59508024	1.73471998	1.41721955	1.05255567
100	1	1.82936161	3.39196233	5.04478653	4.23912733	5.58286970	4.31561513
500	1	1.89022822	3.26501851	5.90480762	11.2617060	10.9150052	11.9031846
2500	1	2.06544092	3.90591668	7.97345188	11.5729983	22.3378446	24.8782391
12500	1	2.02932054	3.90111916	6.92048767	13.9581960	25.3625409	35.7518745



	1	2	4	8	16	32	64
20	1	1.98794940	3.91675618	7.75524139	1.12982989	0.62933486	0.10320796
100	1	2.00560993	3.97444464	7.92566258	2.09958608	1.10419492	1.59410431
500	1	1.97132345	3.98520640	8.00105678	7.50689915	3.18177059	2.31631780
2500	1	2.02383366	4.06757569	8.06565964	7.58965904	7.98902779	5.78722547
12500	1	1.99779790	3.95727163	7.91869776	7.42710988	7.96870778	9.41676428



	1	2	4	8	16	32	64
20	1	1.89812832	3.28397658	5.24763830	8.47694349	11.6947826	12.5475237
100	1	1.71550009	3.08997054	5.51339465	8.96104623	12.6734306	18.4484543
500	1	1.90709123	3.12314234	4.67772452	7.79044494	12.0480000	18.2633256
2500	1	1.87122919	3.62859668	6.47314772	10.8723890	12.4701224	18.7007156
12500	1	1.90870222	3.59079909	6.40926942	11.4949715	17.3414590	20.1188312



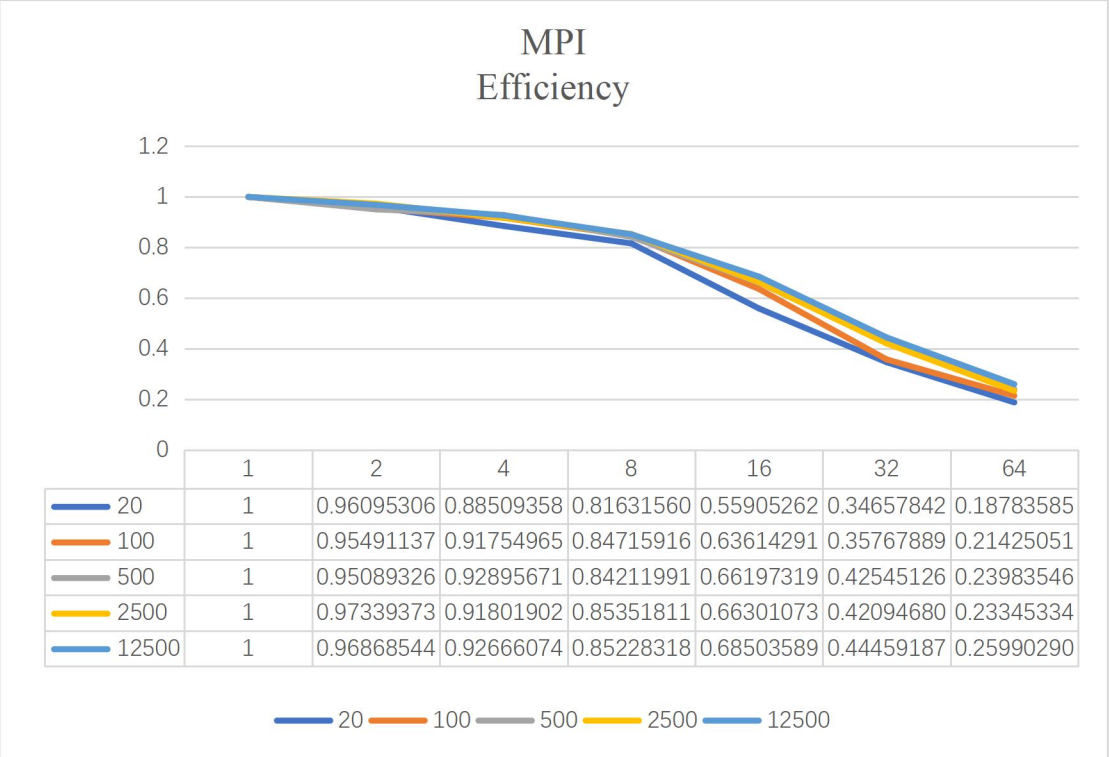
	1	2	4	8	16	32	64
20	1	1.86447245	0.19752989	0.88156538	0.06509955	0.03157010	0.03538640
100	1	1.83544411	0.54424113	2.34605212	0.34447891	0.12258570	0.08741531
500	1	1.90847471	1.20478369	2.74267900	0.79245082	0.49427651	0.26838836
2500	1	1.91501460	1.69158645	0.70765639	1.32952915	1.17338997	0.73346578
12500	1	1.92213296	1.86186782	0.11394213	1.63169852	1.60707208	0.87819663

Generally, all settings except the MPI with OpenMP achieve the best speedup in the large data size and large number of processes or threads. In this case, the communication overhead and threads management overhead are far much smaller than the time spent on computation. For the OpenMP, we can easily see that the performance are the worst in the smallest data test size which is definitely resulted from the extra threads management time. The speedup of MPI with OpenMP are quite different, one possible reason is the communication overhead and threads creation effect offset the speedup bring up by parallel computing.

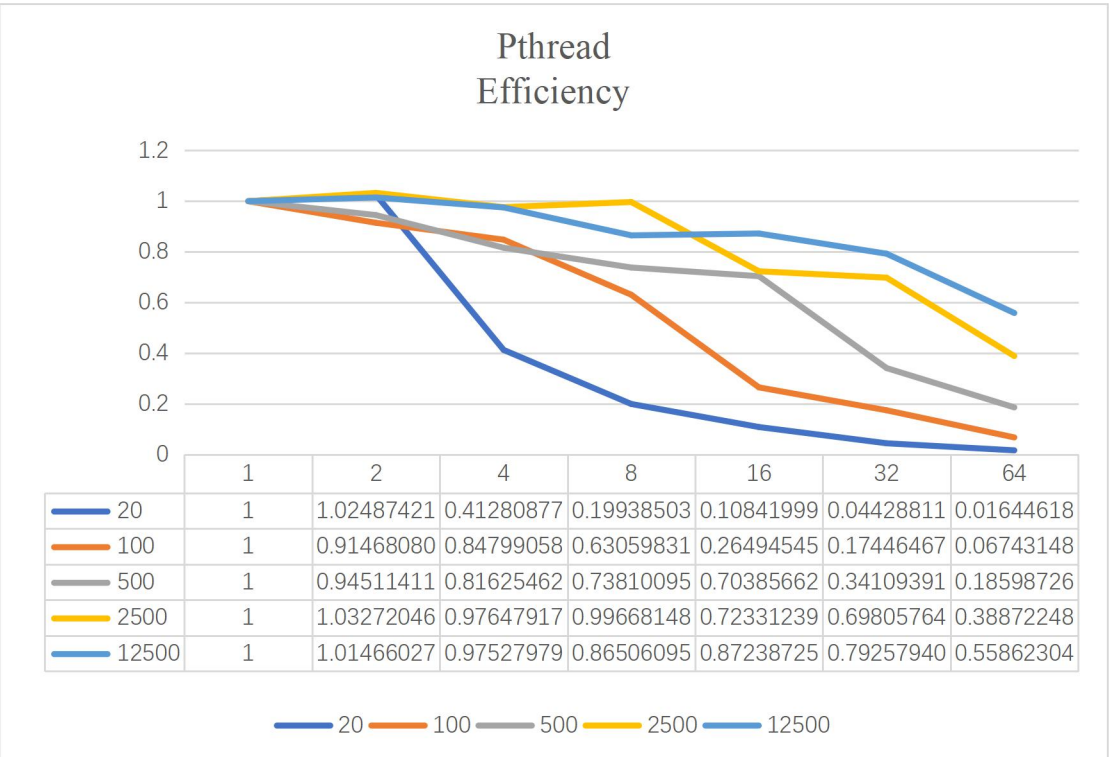
Part 4. Efficiency

The efficiency formula is given as:

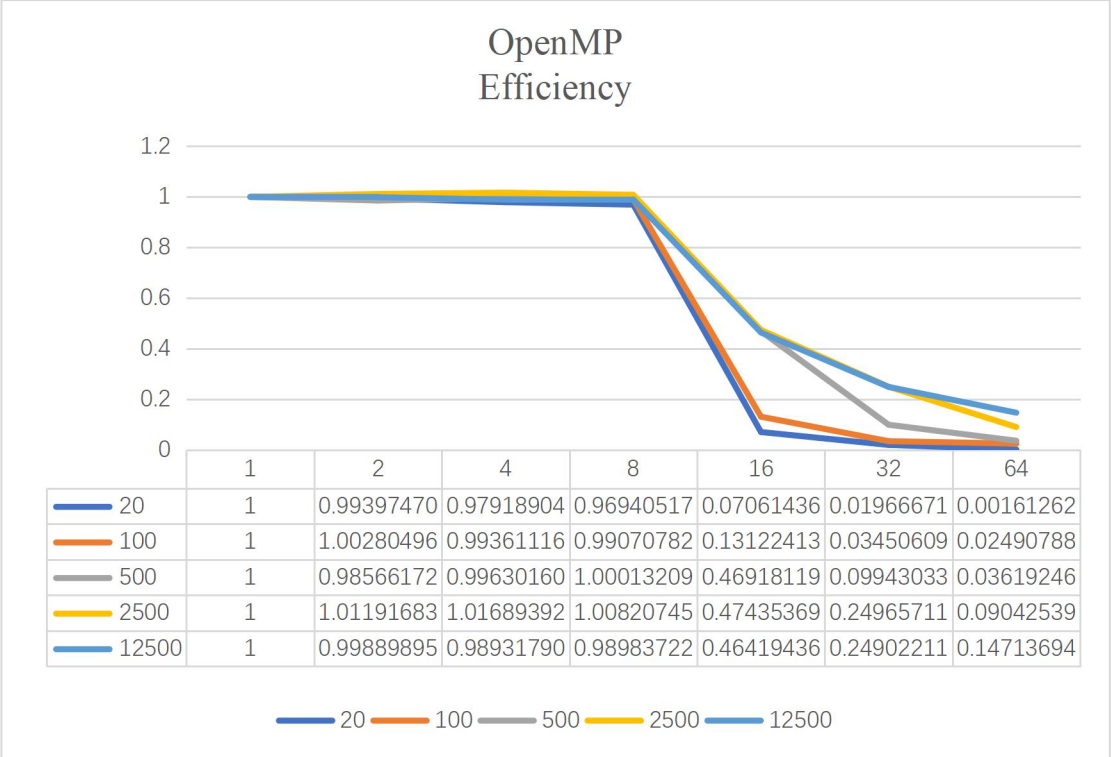
$$E = \frac{Speedup}{number\ of\ processes(or\ threads)}$$



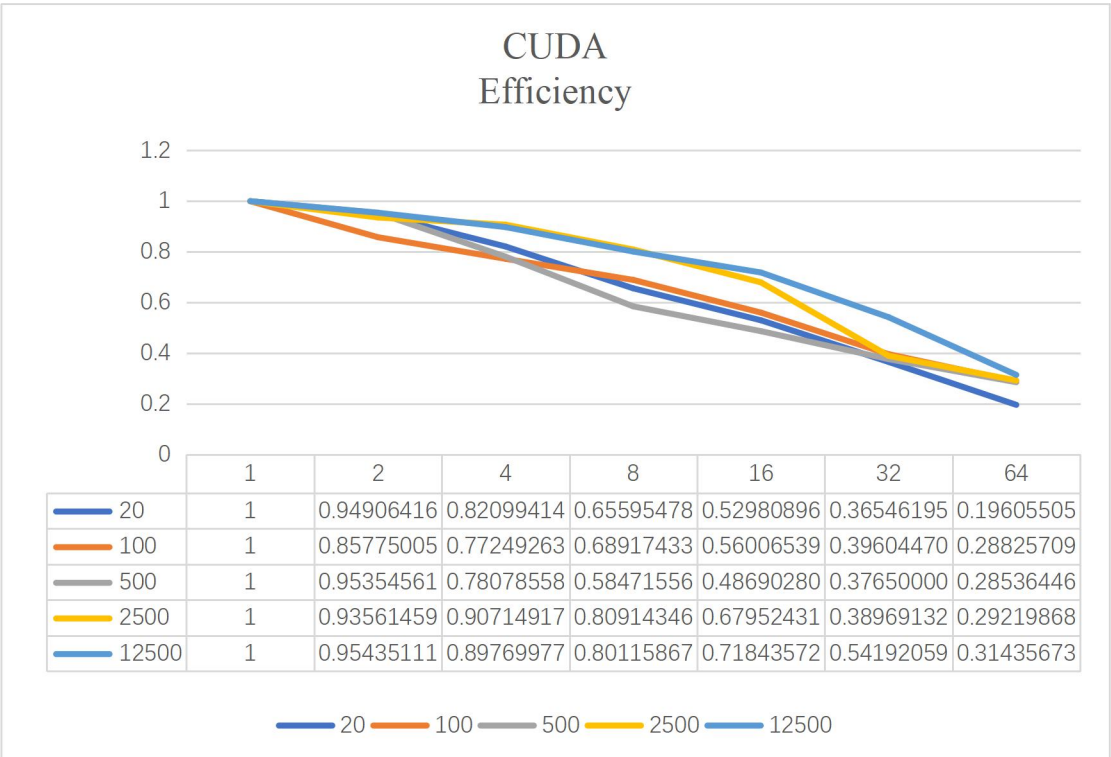
	1	2	4	8	16	32	64
20	1	0.96095306	0.88509358	0.81631560	0.55905262	0.34657842	0.18783585
100	1	0.95491137	0.91754965	0.84715916	0.63614291	0.35767889	0.21425051
500	1	0.95089326	0.92895671	0.84211991	0.66197319	0.42545126	0.23983546
2500	1	0.97339373	0.91801902	0.85351811	0.66301073	0.42094680	0.23345334
12500	1	0.96868544	0.92666074	0.85228318	0.68503589	0.44459187	0.25990290



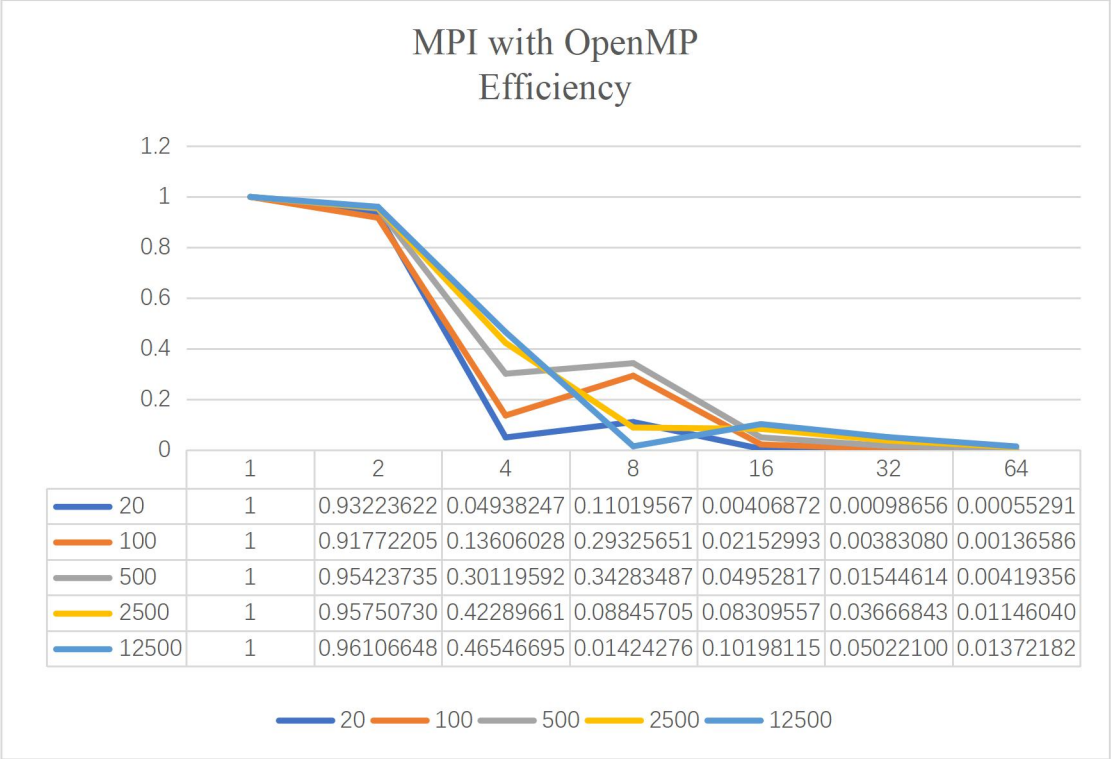
	1	2	4	8	16	32	64
20	1	1.02487421	0.41280877	0.19938503	0.10841999	0.04428811	0.01644618
100	1	0.91468080	0.84799058	0.63059831	0.26494545	0.17446467	0.06743148
500	1	0.94511411	0.81625462	0.73810095	0.70385662	0.34109391	0.18598726
2500	1	1.03272046	0.97647917	0.99668148	0.72331239	0.69805764	0.38872248
12500	1	1.01466027	0.97527979	0.86506095	0.87238725	0.79257940	0.55862304



	1	2	4	8	16	32	64
20	1	0.99397470	0.97918904	0.96940517	0.07061436	0.01966671	0.00161262
100	1	1.00280496	0.99361116	0.99070782	0.13122413	0.03450609	0.02490788
500	1	0.98566172	0.99630160	1.00013209	0.46918119	0.09943033	0.03619246
2500	1	1.01191683	1.01689392	1.00820745	0.47435369	0.24965711	0.09042539
12500	1	0.99889895	0.98931790	0.98983722	0.46419436	0.24902211	0.14713694



	1	2	4	8	16	32	64
20	1	0.94906416	0.82099414	0.65595478	0.52980896	0.36546195	0.19605505
100	1	0.85775005	0.77249263	0.68917433	0.56006539	0.39604470	0.28825709
500	1	0.95354561	0.78078558	0.58471556	0.48690280	0.37650000	0.28536446
2500	1	0.93561459	0.90714917	0.80914346	0.67952431	0.38969132	0.29219868
12500	1	0.95435111	0.89769977	0.80115867	0.71843572	0.54192059	0.31435673



	1	2	4	8	16	32	64
20	1	0.93223622	0.04938247	0.11019567	0.00406872	0.00098656	0.00055291
100	1	0.91772205	0.13606028	0.29325651	0.02152993	0.00383080	0.00136586
500	1	0.95423735	0.30119592	0.34283487	0.04952817	0.01544614	0.00419356
2500	1	0.95750730	0.42289661	0.08845705	0.08309557	0.03666843	0.01146040
12500	1	0.96106648	0.46546695	0.01424276	0.10198115	0.05022100	0.01372182

Generally, the efficiency would decrease with the increase of size of processes and threads since with more threads or process, each process would be assigned with less subtask size which may spent relatively small time for computing and more time for waiting other processes or threads to finish their tasks. For Pthread, when handling large data size with small number of threads, it could sometimes perform quite good at efficiency which is slightly larger than 1. The similar situation was also witness in OpenMP, when using smaller number of threads, it could still get a good efficiency which is remained around 1.

5. Conclusion and Limitations

This report mainly discusses five parallel versions (MPI, Pthread, OpenMP, CUDA, MPI with OpenMP) of the given sequential heat distribution sequential program. For parallel version, the design of program considers the potential data race and avoid it. Tests with different number of cores and threads and data size, the report shows the improvement of parallel calculation and also discuss the communication overhead and thread management overhead may offset the performance improvement brough by parallel calculation especially in MPI with OpenMP.

To further improve the computation efficiency in MPI, we can actually use scatter to distribute the data buffer of grid instead of using broadcast directly. Notice that when do the calculation of every point, we only need 4 neighbor points' information. In this scenario we could decrease the communication overhead of data distributing from $O(N)$ to $O\left(\frac{N}{p}\right)$ which could reduce the useless data transformation time and computing resources.

Codes:

1) MPI

a) main.cpp

```cpp
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <cstring>
#include <chrono>
#include <hdist/hdist.hpp>

template<typename ...Args>
void UNUSED(Args &&... args [[maybe_unused]]) {}

ImColor temp_to_color(double temp) {
    auto value = static_cast<uint8_t>(temp / 100.0 * 255.0);
    return {value, 0, 255 - value};
}

static MPI_Datatype MPI_State;

int main(int argc, char **argv) {
    int rank;
    int processSize;
    MPI_Init(&argc, &argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &processSize);

    // initialize MPI struct
    const int nitems=9;
    int blocklengths[9] = {1,1,1,1,1,1,1,1,1};
    MPI_Datatype types[9] = {MPI_INT, MPI_FLOAT, MPI_INT, MPI_INT,
                MPI_FLOAT, MPI_FLOAT, MPI_FLOAT, MPI_FLOAT, MPI_INT};

    MPI_Aint offsets[9];

    offsets[0] = offsetof(hdist::State, room_size);
    offsets[1] = offsetof(hdist::State, block_size);
    offsets[2] = offsetof(hdist::State, source_x);
    offsets[3] = offsetof(hdist::State, source_y);
    offsets[4] = offsetof(hdist::State, source_temp);
    offsets[5] = offsetof(hdist::State, border_temp);
    offsets[6] = offsetof(hdist::State, tolerance);
    offsets[7] = offsetof(hdist::State, sor_constant);
    offsets[8] = offsetof(hdist::State, algo);

    MPI_Type_create_struct(nitems, blocklengths, offsets, types, &MPI_State);
    MPI_Type_commit(&MPI_State);
```

```
    UNUSED(argc, argv);
    bool first = true;
    bool finished = false;
    static hdist::State current_state, last_state;
    auto grid = hdist::Grid{
            static_cast<size_t>(current_state.room_size),
            current_state.border_temp,
            current_state.source_temp,
            static_cast<size_t>(current_state.source_x),
            static_cast<size_t>(current_state.source_y)};


    if(rank == 0){
        printf("MPI: launch %d processes\n", processSize);

        // test codes
        // int testCases[] = {200, 400, 800, 1600, 3200};
        // int testCaseIdx = 0, testCnt = 0;
        // static std::chrono::high_resolution_clock::time_point beginIteration,
endIteration;

        static std::chrono::high_resolution_clock::time_point begin, end;
        static const char* algo_list[2] = { "jacobi", "sor" };
        graphic::GraphicContext context{"Assignment 4"};
        context.run([&](graphic::GraphicContext *context [[maybe_unused]], SDL_Window *) {
            auto io = ImGui::GetIO();
            ImGui::SetNextWindowPos(ImVec2(0.0f, 0.0f));
            ImGui::SetNextWindowSize(io.DisplaySize);
            ImGui::Begin("Assignment 4", nullptr,
                         ImGuiWindowFlags_NoMove
                         | ImGuiWindowFlags_NoCollapse
                         | ImGuiWindowFlags_NoTitleBar
                         | ImGuiWindowFlags_NoResize);
            ImDrawList *draw_list = ImGui::GetWindowDrawList();
            ImGui::Text("Application average %.3f ms/frame (%.1f FPS)", 1000.0f /
ImGui::GetIO().Framerate,
                        ImGui::GetIO().Framerate);
            ImGui::DragInt("Room Size", &current_state.room_size, 10, 200, 1600, "%d");
            ImGui::DragFloat("Block Size", &current_state.block_size, 0.01, 0.1, 10, "%f");
            ImGui::DragFloat("Source Temp", &current_state.source_temp, 0.1, 0, 100, "%f");
            ImGui::DragFloat("Border Temp", &current_state.border_temp, 0.1, 0, 100, "%f");
            ImGui::DragInt("Source X", &current_state.source_x, 1, 1, current_state.room_size
- 2, "%d");
            ImGui::DragInt("Source Y", &current_state.source_y, 1, 1, current_state.room_size
- 2, "%d");
            ImGui::DragFloat("Tolerance", &current_state.tolerance, 0.01, 0.01, 1, "%f");
            ImGui::ListBox("Algorithm", reinterpret_cast<int *>(&current_state.algo),
algo_list, 2);
```

```
            // testCnt++;

            // if(testCnt == 5){
            //     current_state.room_size = testCases[testCaseIdx];
            //     testCaseIdx = (testCaseIdx+1) % 5;
            //     testCnt = 0;
            // }

            if (current_state.algo == hdist::Algorithm::Sor) {
                ImGui::DragFloat("Sor Constant", &current_state.sor_constant, 0.01, 0.0, 20.0,
"%f");
            }

            // beginIteration = std::chrono::high_resolution_clock::now();

            // broadcast current state
            MPI_Bcast(&current_state, 1, MPI_State, 0, MPI_COMM_WORLD);

            // change size regenerate
            if (current_state.room_size != last_state.room_size) {
                grid = hdist::Grid{
                        static_cast<size_t>(current_state.room_size),
                        current_state.border_temp,
                        current_state.source_temp,
                        static_cast<size_t>(current_state.source_x),
                        static_cast<size_t>(current_state.source_y)};
                first = true;
            }

            if (current_state != last_state) {
                last_state = current_state;
                finished = false;
            }

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

            if (!finished) {
                int dataSize = current_state.room_size*current_state.room_size;

                // bcast get_current_buffer
                MPI_Bcast(grid.get_current_buffer().data(), dataSize, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

                // do calculation
                finished = hdist::calculate(current_state, grid);
```

```
                bool localFinished = finished;

                // reduce and broadcast finished
                MPI_Reduce(&localFinished, &finished, 1, MPI_C_BOOL, MPI_LAND, 0,
MPI_COMM_WORLD);
                MPI_Bcast(&finished, 1, MPI_C_BOOL, 0, MPI_COMM_WORLD);

                // gatherv get_current_buffer result from all ranks
                // calculate the job size of each rank
                std::vector<int> taskSizeList;
                for(int i=0;i<processSize;i++){
                    int startPosition = (dataSize/processSize)*i +
std::min(dataSize%processSize, i);
                    int endPosition = (dataSize/processSize)*(i+1) +
std::min(dataSize%processSize, i+1)-1;
                    taskSizeList.push_back(endPosition-startPosition+1);
                }

                // calculate the strips of received data packages
                std::vector<int> strips;
                strips.push_back(0);
                for(int i=1;i<processSize;i++){
                    strips.push_back(strips[i-1]+taskSizeList[i-1]);
                }

                // calcualte job size of itself
                int startPosition = (dataSize/processSize)*rank +
std::min(dataSize%processSize, rank);
                int endPosition = (dataSize/processSize)*(rank+1) +
std::min(dataSize%processSize, rank+1)-1;
                int jobSize = endPosition-startPosition+1;

                // rank 0 gather calculated result
                MPI_Gatherv(grid.get_current_buffer().data()+startPosition, jobSize,
MPI_DOUBLE,
                    grid.get_current_buffer().data(), taskSizeList.data(), strips.data(),
MPI_DOUBLE, 0, MPI_COMM_WORLD);

                if (finished) end = std::chrono::high_resolution_clock::now();

                // endIteration = std::chrono::high_resolution_clock::now();
                // printf("processes %d\tone iteration %lld (ns)\troom size %d\n",
                // processSize,
std::chrono::duration_cast<std::chrono::nanoseconds>(endIteration - beginIteration).count(),
current_state.room_size);

            } else {
```

```
                ImGui::Text("stabilized in %lld ns",
std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count());
                printf("stabilized in %lld ns\n",
std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count());
            }

            const ImVec2 p = ImGui::GetCursorScreenPos();
            float x = p.x + current_state.block_size, y = p.y + current_state.block_size;
            for (size_t i = 0; i < current_state.room_size; ++i) {
                for (size_t j = 0; j < current_state.room_size; ++j) {
                    auto temp = grid[{i, j}];
                    auto color = temp_to_color(temp);
                    draw_list->AddRectFilled(ImVec2(x, y), ImVec2(x +
current_state.block_size, y + current_state.block_size), color);
                    y += current_state.block_size;
                }
                x += current_state.block_size;
                y = p.y + current_state.block_size;
            }
            ImGui::End();
        });
    }
    else{
        while(true){
            // receive current state
            MPI_Bcast(&current_state, 1, MPI_State, 0, MPI_COMM_WORLD);

            // change size regenerate
            if (current_state.room_size != last_state.room_size) {
                grid = hdist::Grid{
                        static_cast<size_t>(current_state.room_size),
                        current_state.border_temp,
                        current_state.source_temp,
                        static_cast<size_t>(current_state.source_x),
                        static_cast<size_t>(current_state.source_y)};
                first = true;
            }

            if (current_state != last_state) {
                last_state = current_state;
                finished = false;
            }

            if (first) {
                first = false;
                finished = false;
            }

            if (!finished) {
```

```
                int dataSize = current_state.room_size*current_state.room_size;

                // bcast get_current_buffer
                MPI_Bcast(grid.get_current_buffer().data(), dataSize, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

                // do calculation
                finished = hdist::calculate(current_state, grid);

                // reduce and receive finished
                MPI_Reduce(&finished, nullptr, 1, MPI_C_BOOL, MPI_LAND, 0, MPI_COMM_WORLD);
                MPI_Bcast(&finished, 1, MPI_C_BOOL, 0, MPI_COMM_WORLD);

                // send get_current_buffer result to rank 0
                // calcualte job size of itself
                int startPosition = (dataSize/processSize)*rank +
std::min(dataSize%processSize, rank);
                int endPosition = (dataSize/processSize)*(rank+1) +
std::min(dataSize%processSize, rank+1)-1;
                int jobSize = endPosition-startPosition+1;

                // send calculated result
                MPI_Gatherv(grid.get_current_buffer().data()+startPosition, jobSize,
MPI_DOUBLE, nullptr, nullptr, nullptr,
                    MPI_DOUBLE, 0, MPI_COMM_WORLD);
            }

        }
    }

    // if ploting windows closed, rank 0 would then stop all other processes
    if(rank==0){
        printf("aborting all processes\n");
        MPI_Abort(MPI_COMM_WORLD, 0);
    }

    MPI_Type_free(&MPI_State);
    MPI_Finalize();
}
```

b) hdist.hpp

```
#pragma once

#include <vector>
#include <mpi.h>

namespace hdist {
```

```
enum class Algorithm : int {
    Jacobi = 0,
    Sor = 1
};

struct State {
    int room_size = 300;
    float block_size = 2;
    int source_x = room_size / 2;
    int source_y = room_size / 2;
    float source_temp = 100;
    float border_temp = 36;
    float tolerance = 0.02;
    float sor_constant = 4.0;
    Algorithm algo = hdist::Algorithm::Jacobi;

    bool operator==(const State &that) const = default;
};

struct Alt {
};

constexpr static inline Alt alt{};

struct Grid {
    std::vector<double> data0, data1;
    size_t current_buffer = 0;
    size_t length;

    explicit Grid(size_t size,
                  double border_temp,
                  double source_temp,
                  size_t x,
                  size_t y)
            : data0(size * size), data1(size * size), length(size) {
        for (size_t i = 0; i < length; ++i) {
            for (size_t j = 0; j < length; ++j) {
                if (i == 0 || j == 0 || i == length - 1 || j == length - 1) {
                    this->operator[]({i, j}) = border_temp;
                } else if (i == x && j == y) {
                    this->operator[]({i, j}) = source_temp;
                } else {
                    this->operator[]({i, j}) = 0;
                }
            }
        }
    }

    std::vector<double> &get_current_buffer() {
```

```
            if (current_buffer == 0) return data0;
            return data1;
        }

        std::vector<double> &get_alternate_buffer() {
            if (current_buffer == 0) return data1;
            return data0;
        }

        double &operator[](std::pair<size_t, size_t> index) {
            return get_current_buffer()[index.first * length + index.second];
        }

        double &operator[](std::tuple<Alt, size_t, size_t> index) {
            return current_buffer == 1 ? data0[std::get<1>(index) * length +
std::get<2>(index)] : data1[
                    std::get<1>(index) * length + std::get<2>(index)];
        }

        void switch_buffer() {
            current_buffer = !current_buffer;
        }
    };

    struct UpdateResult {
        bool stable;
        double temp;
    };

    UpdateResult update_single(size_t i, size_t j, Grid &grid, const State &state) {
        UpdateResult result{};
        if (i == 0 || j == 0 || i == state.room_size - 1 || j == state.room_size - 1) {
            result.temp = state.border_temp;
        } else if (i == state.source_x && j == state.source_y) {
            result.temp = state.source_temp;
        } else {
            auto sum = (grid[{i + 1, j}] + grid[{i - 1, j}] + grid[{i, j + 1}] + grid[{i, j -
1}]);
            switch (state.algo) {
                case Algorithm::Jacobi:
                    result.temp = 0.25 * sum;
                    break;
                case Algorithm::Sor:
                    result.temp = grid[{i, j}] + (1.0 / state.sor_constant) * (sum - 4.0 * grid[{i,
j}]);
                    break;
            }
        }
        result.stable = fabs(grid[{i, j}] - result.temp) < state.tolerance;
```

```
        return result;
    }

    bool calculate(const State &state, Grid &grid) {
        bool stabilized = true;
        int rank;
        int processSize;
        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &processSize);

        int dataSize = state.room_size*state.room_size;

        // calculate start and end position at buffer
        int startPosition = (dataSize/processSize)*rank + std::min(dataSize%processSize,
rank);
        int endPosition = (dataSize/processSize)*(rank+1) + std::min(dataSize%processSize,
rank+1)-1;
        int jobSize = endPosition-startPosition+1;

        // calculate start i and j
        int startI = startPosition/state.room_size;
        int i, j, count, endI;
        bool switchOnce = false;

        switch (state.algo) {
            case Algorithm::Jacobi:
                i = startI;
                j = startPosition%state.room_size;
                count = 0;
                for (; i < state.room_size; ++i) {
                    if(i!=startI){
                        j=0;
                    }
                    for (; j < state.room_size; ++j) {
                        // avoid extra calculation
                        if(count>=jobSize){
                            // break outside for loop
                            i = state.room_size;
                            break;
                        }
                        count++;

                        auto result = update_single(i, j, grid, state);
                        stabilized &= result.stable;
                        grid[{alt, i, j}] = result.temp;
                    }
                }
                grid.switch_buffer();
                break;
```

```
        case Algorithm::Sor:
            for (auto k : {0, 1}) {
                i = startI;
                j = startPosition%state.room_size;
                count = 0;

                // calculate the left and top extra points
                if(switchOnce==false){
                    // left points
                    if(i!=0){
                        int x = i-1;
                        for(int y=j;y<state.room_size;y++){
                            if (k == ((x + y) & 1)) {
                                auto result = update_single(x, y, grid, state);
                                // stabilized &= result.stable;
                                grid[{alt, x, y}] = result.temp;
                            } else {
                                grid[{alt, x, y}] = grid[{x, y}];
                            }
                        }
                    }

                    // top points
                    for(int y=0; y<j;y++){
                        if (k == ((i + y) & 1)) {
                            auto result = update_single(i, y, grid, state);
                            // stabilized &= result.stable;
                            grid[{alt, i, y}] = result.temp;
                        } else {
                            grid[{alt, i, y}] = grid[{i, y}];
                        }
                    }
                }


                bool breakLoop = false;

                for (; i < state.room_size; i++) {
                    if(i!=startI){
                        j=0;
                    }

                    for (; j < state.room_size; j++) {
                        // avoid extra calculation
                        if(count>=jobSize){
                            // break outside for loop
                            endI = i;
                            i = state.room_size;
                            break;
```

```
                }
                count++;

                if (k == ((i + j) & 1)) {
                    auto result = update_single(i, j, grid, state);
                    stabilized &= result.stable;
                    grid[{alt, i, j}] = result.temp;
                } else {
                    grid[{alt, i, j}] = grid[{i, j}];
                }
            }
        }

        // calculate the right and bottom extra points
        if(switchOnce==false){
            // last point reach the bottom exactly
            int rightJ = j;
            int rightI = endI;
            if(j==0){
                rightJ = state.room_size-1;
                // block bottom points calculation
                j = state.room_size;
            }
            else{
                rightI++;
            }

            // calculate right points
            if(rightI<state.room_size){
                int x = rightI;

                for(int y=0;y<rightJ; y++){
                    if (k == ((x + y) & 1)) {
                        auto result = update_single(x, y, grid, state);
                        // stabilized &= result.stable;
                        grid[{alt, x, y}] = result.temp;
                    } else {
                        grid[{alt, x, y}] = grid[{x, y}];
                    }
                }
            }

            // calculate bottom points
            for(int y=j;y<state.room_size;y++){
                if (k == ((endI + y) & 1)) {
                    auto result = update_single(endI, y, grid, state);
                    // stabilized &= result.stable;
                    grid[{alt, endI, y}] = result.temp;
                } else {
```

```
                            grid[{alt, endI, y}] = grid[{endI, y}];
                        }
                    }

                }

                switchOnce = true;

                grid.switch_buffer();
            }
        }

        // printf("rank %d %d %d %d %d %d\n", rank, jobSize, starti, startj, curi, j);

        return stabilized;
    };


} // namespace hdist
```

2) Pthread

a) main.cpp

```
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <cstring>
#include <chrono>
#include <hdist/hdist.hpp>

template<typename ...Args>
void UNUSED(Args &&... args [[maybe_unused]]) {}

ImColor temp_to_color(double temp) {
    auto value = static_cast<uint8_t>(temp / 100.0 * 255.0);
    return {value, 0, 255 - value};
}

int main(int argc, char **argv) {
    // use 4 threadsd as default
    int totalThreadSize = 4;

    // fetch thread size from command line
    if(argc>=2){
        totalThreadSize = atoi(argv[1]);
    }

    // avoid invalid input
    if(totalThreadSize==0){
```

```
        totalThreadSize = 4;
    }

    printf("Pthread: launch %d threads\n", totalThreadSize);
    // test codes
    // int testCases[] = {200, 400, 800, 1600, 3200};
    // int testCaseIdx = 0, testCnt = 0;
    // static std::chrono::high_resolution_clock::time_point beginIteration, endIteration;

    UNUSED(argc, argv);
    bool first = true;
    bool finished = false;
    static hdist::State current_state, last_state;
    static std::chrono::high_resolution_clock::time_point begin, end;
    static const char* algo_list[2] = { "jacobi", "sor" };
    graphic::GraphicContext context{"Assignment 4"};
    auto grid = hdist::Grid{
            static_cast<size_t>(current_state.room_size),
            current_state.border_temp,
            current_state.source_temp,
            static_cast<size_t>(current_state.source_x),
            static_cast<size_t>(current_state.source_y)};
    context.run([&](graphic::GraphicContext *context [[maybe_unused]], SDL_Window *) {
        auto io = ImGui::GetIO();
        ImGui::SetNextWindowPos(ImVec2(0.0f, 0.0f));
        ImGui::SetNextWindowSize(io.DisplaySize);
        ImGui::Begin("Assignment 4", nullptr,
                     ImGuiWindowFlags_NoMove
                     | ImGuiWindowFlags_NoCollapse
                     | ImGuiWindowFlags_NoTitleBar
                     | ImGuiWindowFlags_NoResize);
        ImDrawList *draw_list = ImGui::GetWindowDrawList();
        ImGui::Text("Application average %.3f ms/frame (%.1f FPS)", 1000.0f /
ImGui::GetIO().Framerate,
                    ImGui::GetIO().Framerate);
        ImGui::DragInt("Room Size", &current_state.room_size, 10, 200, 1600, "%d");
        ImGui::DragFloat("Block Size", &current_state.block_size, 0.01, 0.1, 10, "%f");
        ImGui::DragFloat("Source Temp", &current_state.source_temp, 0.1, 0, 100, "%f");
        ImGui::DragFloat("Border Temp", &current_state.border_temp, 0.1, 0, 100, "%f");
        ImGui::DragInt("Source X", &current_state.source_x, 1, 1, current_state.room_size - 2,
"%d");
        ImGui::DragInt("Source Y", &current_state.source_y, 1, 1, current_state.room_size - 2,
"%d");
        ImGui::DragFloat("Tolerance", &current_state.tolerance, 0.01, 0.01, 1, "%f");
        ImGui::ListBox("Algorithm", reinterpret_cast<int *>(&current_state.algo), algo_list,
2);

        // testCnt++;
```

```
        // if(testCnt == 5){
        //     current_state.room_size = testCases[testCaseIdx];
        //     testCaseIdx = (testCaseIdx+1) % 5;
        //     testCnt = 0;
        // }

        if (current_state.algo == hdist::Algorithm::Sor) {
            ImGui::DragFloat("Sor Constant", &current_state.sor_constant, 0.01, 0.0, 20.0,
"%f");
        }

        // beginIteration = std::chrono::high_resolution_clock::now();

        if (current_state.room_size != last_state.room_size) {
            grid = hdist::Grid{
                    static_cast<size_t>(current_state.room_size),
                    current_state.border_temp,
                    current_state.source_temp,
                    static_cast<size_t>(current_state.source_x),
                    static_cast<size_t>(current_state.source_y)};
            first = true;
        }

        if (current_state != last_state) {
            last_state = current_state;
            finished = false;
        }

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

        if (!finished) {
            finished = hdist::calculate(current_state, grid, totalThreadSize);
            if (finished) end = std::chrono::high_resolution_clock::now();

            // endIteration = std::chrono::high_resolution_clock::now();
            // printf("processes %d\tone iteration %lld (ns)\troom size %d\n",
            // totalThreadSize,
std::chrono::duration_cast<std::chrono::nanoseconds>(endIteration - beginIteration).count(),
current_state.room_size);

        } else {
            ImGui::Text("stabilized in %lld ns",
std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count());
            printf("stabilized in %lld ns\n",
std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count());
```

```
        }

        const ImVec2 p = ImGui::GetCursorScreenPos();
        float x = p.x + current_state.block_size, y = p.y + current_state.block_size;
        for (size_t i = 0; i < current_state.room_size; ++i) {
            for (size_t j = 0; j < current_state.room_size; ++j) {
                auto temp = grid[{i, j}];
                auto color = temp_to_color(temp);
                draw_list->AddRectFilled(ImVec2(x, y), ImVec2(x + current_state.block_size, y
+ current_state.block_size), color);
                y += current_state.block_size;
            }
            x += current_state.block_size;
            y = p.y + current_state.block_size;
        }
        ImGui::End();
    });
}
```

b) hdist.hpp

```
#pragma once

#include <vector>
#include <iostream>

namespace hdist {
    // use to hold the barrier
    static pthread_barrier_t barrier;

    enum class Algorithm : int {
        Jacobi = 0,
        Sor = 1
    };

    struct State {
        int room_size = 300;
        float block_size = 2;
        int source_x = room_size / 2;
        int source_y = room_size / 2;
        float source_temp = 100;
        float border_temp = 36;
        float tolerance = 0.02;
        float sor_constant = 4.0;
        Algorithm algo = hdist::Algorithm::Jacobi;

        bool operator==(const State &that) const = default;
    };
```

```
    struct Alt {
    };

    constexpr static inline Alt alt{};

    struct Grid {
        std::vector<double> data0, data1;
        size_t current_buffer = 0;
        size_t length;

        explicit Grid(size_t size,
                      double border_temp,
                      double source_temp,
                      size_t x,
                      size_t y)
                : data0(size * size), data1(size * size), length(size) {
            for (size_t i = 0; i < length; ++i) {
                for (size_t j = 0; j < length; ++j) {
                    if (i == 0 || j == 0 || i == length - 1 || j == length - 1) {
                        this->operator[]({i, j}) = border_temp;
                    } else if (i == x && j == y) {
                        this->operator[]({i, j}) = source_temp;
                    } else {
                        this->operator[]({i, j}) = 0;
                    }
                }
            }
        }

        std::vector<double> &get_current_buffer() {
            if (current_buffer == 0) return data0;
            return data1;
        }

        double &operator[](std::pair<size_t, size_t> index) {
            return get_current_buffer()[index.first * length + index.second];
        }

        double &operator[](std::tuple<Alt, size_t, size_t> index) {
            return current_buffer == 1 ? data0[std::get<1>(index) * length +
std::get<2>(index)] : data1[
                    std::get<1>(index) * length + std::get<2>(index)];
        }

        void switch_buffer() {
            current_buffer = !current_buffer;
        }
```

```
        static void switch_buffer(Grid & grid) {
            grid.current_buffer = !grid.current_buffer;
        }
    };

    // struct to hold parameters
    struct Arguments{
        const State &state;
        Grid &grid;
        std::vector<int> &stateList;
        int taskNum;
        int totalThreadSize;
    };

    struct UpdateResult {
        bool stable;
        double temp;
    };

    UpdateResult update_single(size_t i, size_t j, Grid &grid, const State &state) {
        UpdateResult result{};
        if (i == 0 || j == 0 || i == state.room_size - 1 || j == state.room_size - 1) {
            result.temp = state.border_temp;
        } else if (i == state.source_x && j == state.source_y) {
            result.temp = state.source_temp;
        } else {
            auto sum = (grid[{i + 1, j}] + grid[{i - 1, j}] + grid[{i, j + 1}] + grid[{i, j -
1}]);
            switch (state.algo) {
                case Algorithm::Jacobi:
                    result.temp = 0.25 * sum;
                    break;
                case Algorithm::Sor:
                    result.temp = grid[{i, j}] + (1.0 / state.sor_constant) * (sum - 4.0 * grid[{i,
j}]);
                    break;
            }
        }
        result.stable = fabs(grid[{i, j}] - result.temp) < state.tolerance;
        return result;
    }

    void* calculateInside(void *arg_ptr){
        // retrieve information from passed parameters
        auto arguments = static_cast<Arguments *>(arg_ptr);
        const State & state = arguments->state;
        Grid & grid = arguments->grid;
        std::vector<int> & stateList = arguments->stateList;
        int taskNum = arguments->taskNum;
```

```
        int totalThreadSize = arguments->totalThreadSize;

        // std::cout << &state << " " << &grid << " " << &stateList << " [" << taskNum << std::endl;

        bool stabilized = true;

        int dataSize = state.room_size*state.room_size;

        // calculate start and end position at buffer
        int startPosition = (dataSize/totalThreadSize)*taskNum +
std::min(dataSize%totalThreadSize, taskNum);
        int endPosition = (dataSize/totalThreadSize)*(taskNum+1) +
std::min(dataSize%totalThreadSize, taskNum+1)-1;
        int jobSize = endPosition-startPosition+1;

        // calculate start i and j
        int startI = startPosition/state.room_size;
        int i, j, count;
        bool switchOnce = false;

        switch (state.algo) {
            case Algorithm::Jacobi:
                i = startI;
                j = startPosition%state.room_size;
                count = 0;
                for (; i < state.room_size; ++i) {
                    if(i!=startI){
                        j=0;
                    }
                    for (; j < state.room_size; ++j) {
                        // avoid extra calculation
                        if(count>=jobSize){
                            // break outside for loop
                            i = state.room_size;
                            break;
                        }
                        count++;

                        auto result = update_single(i, j, grid, state);
                        stabilized &= result.stable;
                        grid[{alt, i, j}] = result.temp;
                    }
                }
                break;
            case Algorithm::Sor:
                for (auto k : {0, 1}) {
                    i = startI;
                    j = startPosition%state.room_size;
                    count = 0;
```

```
                for (; i < state.room_size; i++) {
                    if(i!=startI){
                        j=0;
                    }
                    for (; j < state.room_size; j++) {
                        // avoid extra calculation
                        if(count>=jobSize){
                            // break outside for loop
                            i = state.room_size;
                            break;
                        }
                        count++;

                        if (k == ((i + j) & 1)) {
                            auto result = update_single(i, j, grid, state);
                            stabilized &= result.stable;
                            grid[{alt, i, j}] = result.temp;
                        } else {
                            grid[{alt, i, j}] = grid[{i, j}];
                        }
                    }
                }
                // wait until all threads finish calculation
                pthread_barrier_wait(&barrier);
                if(taskNum==0 && switchOnce==false){
                    grid.switch_buffer();
                    switchOnce = true;
                }
                pthread_barrier_wait(&barrier);
            }
        }

        stateList[taskNum] = stabilized;

        return nullptr;
    }

    bool calculate(const State &state, Grid &grid, int totalThreadSize) {
        std::vector<int> stateList(totalThreadSize, true);

        std::vector<pthread_t> threads(totalThreadSize);

        pthread_barrier_init(&barrier, NULL, totalThreadSize);

        // std::cout << &state << " " << &grid << " " << &stateList << std::endl;

        // initialize each thread with task information
        for(int i=0;i<totalThreadSize;i++){
            pthread_create(&threads[i], nullptr, calculateInside, new Arguments{
```

```
                    .state = state,
                    .grid = grid,
                    .stateList = stateList,
                    .taskNum = i,
                    .totalThreadSize = totalThreadSize
            });
        }

        // wait until all the threads finish
        for (auto & i : threads) {
            pthread_join(i, nullptr);
        }

        pthread_barrier_destroy(&barrier);

        bool statelizedAll = true;

        for(bool stabilized : stateList){
            // printf("checking %d\n", stabilized);
            statelizedAll &= stabilized;
        }

        grid.switch_buffer();

        return statelizedAll;
    }



} // namespace hdist
```

3) OpenMP:

a) main.cpp

```
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <cstring>
#include <chrono>
#include <hdist/hdist.hpp>

template<typename ...Args>
void UNUSED(Args &&... args [[maybe_unused]]) {}

ImColor temp_to_color(double temp) {
    auto value = static_cast<uint8_t>(temp / 100.0 * 255.0);
    return {value, 0, 255 - value};
}
```

```
int main(int argc, char **argv) {
    // use 4 threadsd as default
    int totalThreadSize = 4;

    // fetch thread size from command line
    if(argc>=2){
        totalThreadSize = atoi(argv[1]);
    }

    // avoid invalid input
    if(totalThreadSize==0){
        totalThreadSize = 4;
    }

    printf("OpenMP: launch %d threads\n", totalThreadSize);
    // test codes
    // int testCases[] = {200, 400, 800, 1600, 3200};
    // int testCaseIdx = 0, testCnt = 0;
    // static std::chrono::high_resolution_clock::time_point beginIteration, endIteration;

    omp_set_num_threads(totalThreadSize);

    UNUSED(argc, argv);
    bool first = true;
    bool finished = false;
    static hdist::State current_state, last_state;
    static std::chrono::high_resolution_clock::time_point begin, end;
    static const char* algo_list[2] = { "jacobi", "sor" };
    graphic::GraphicContext context{"Assignment 4"};
    auto grid = hdist::Grid{
            static_cast<size_t>(current_state.room_size),
            current_state.border_temp,
            current_state.source_temp,
            static_cast<size_t>(current_state.source_x),
            static_cast<size_t>(current_state.source_y)};
    context.run([&](graphic::GraphicContext *context [[maybe_unused]], SDL_Window *) {
        auto io = ImGui::GetIO();
        ImGui::SetNextWindowPos(ImVec2(0.0f, 0.0f));
        ImGui::SetNextWindowSize(io.DisplaySize);
        ImGui::Begin("Assignment 4", nullptr,
                     ImGuiWindowFlags_NoMove
                     | ImGuiWindowFlags_NoCollapse
                     | ImGuiWindowFlags_NoTitleBar
                     | ImGuiWindowFlags_NoResize);
        ImDrawList *draw_list = ImGui::GetWindowDrawList();
        ImGui::Text("Application average %.3f ms/frame (%.1f FPS)", 1000.0f /
ImGui::GetIO().Framerate,
                    ImGui::GetIO().Framerate);
        ImGui::DragInt("Room Size", &current_state.room_size, 10, 200, 1600, "%d");
```

```
        ImGui::DragFloat("Block Size", &current_state.block_size, 0.01, 0.1, 10, "%f");
        ImGui::DragFloat("Source Temp", &current_state.source_temp, 0.1, 0, 100, "%f");
        ImGui::DragFloat("Border Temp", &current_state.border_temp, 0.1, 0, 100, "%f");
        ImGui::DragInt("Source X", &current_state.source_x, 1, 1, current_state.room_size - 2,
"%d");
        ImGui::DragInt("Source Y", &current_state.source_y, 1, 1, current_state.room_size - 2,
"%d");
        ImGui::DragFloat("Tolerance", &current_state.tolerance, 0.01, 0.01, 1, "%f");
        ImGui::ListBox("Algorithm", reinterpret_cast<int *>(&current_state.algo), algo_list,
2);

        // testCnt++;

        // if(testCnt == 5){
        //     current_state.room_size = testCases[testCaseIdx];
        //     testCaseIdx = (testCaseIdx+1) % 5;
        //     testCnt = 0;
        // }

        if (current_state.algo == hdist::Algorithm::Sor) {
            ImGui::DragFloat("Sor Constant", &current_state.sor_constant, 0.01, 0.0, 20.0,
"%f");
        }

        // beginIteration = std::chrono::high_resolution_clock::now();

        if (current_state.room_size != last_state.room_size) {
            grid = hdist::Grid{
                    static_cast<size_t>(current_state.room_size),
                    current_state.border_temp,
                    current_state.source_temp,
                    static_cast<size_t>(current_state.source_x),
                    static_cast<size_t>(current_state.source_y)};
            first = true;
        }

        if (current_state != last_state) {
            last_state = current_state;
            finished = false;
        }

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

        if (!finished) {
            finished = hdist::calculate(current_state, grid, totalThreadSize);
```

```
            if (finished) end = std::chrono::high_resolution_clock::now();

            // endIteration = std::chrono::high_resolution_clock::now();
            // printf("thread %d\tone iteration %lld (ns)\troom size %d\n",
            // totalThreadSize,
std::chrono::duration_cast<std::chrono::nanoseconds>(endIteration - beginIteration).count(),
current_state.room_size);
        } else {
            ImGui::Text("stabilized in %lld ns",
std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count());
            printf("stabilized in %lld ns\n",
std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count());
        }

        const ImVec2 p = ImGui::GetCursorScreenPos();
        float x = p.x + current_state.block_size, y = p.y + current_state.block_size;
        for (size_t i = 0; i < current_state.room_size; ++i) {
            for (size_t j = 0; j < current_state.room_size; ++j) {
                auto temp = grid[{i, j}];
                auto color = temp_to_color(temp);
                draw_list->AddRectFilled(ImVec2(x, y), ImVec2(x + current_state.block_size, y
+ current_state.block_size), color);
                y += current_state.block_size;
            }
            x += current_state.block_size;
            y = p.y + current_state.block_size;
        }
        ImGui::End();
    });
}
```

b) hdist.hpp

```
#pragma once

#include <vector>
#include <omp.h>

namespace hdist {

    enum class Algorithm : int {
        Jacobi = 0,
        Sor = 1
    };

    struct State {
        int room_size = 300;
        float block_size = 2;
```

```
    int source_x = room_size / 2;
    int source_y = room_size / 2;
    float source_temp = 100;
    float border_temp = 36;
    float tolerance = 0.02;
    float sor_constant = 4.0;
    Algorithm algo = hdist::Algorithm::Jacobi;

    bool operator==(const State &that) const = default;
};

struct Alt {
};

constexpr static inline Alt alt{};

struct Grid {
    std::vector<double> data0, data1;
    size_t current_buffer = 0;
    size_t length;

    explicit Grid(size_t size,
                  double border_temp,
                  double source_temp,
                  size_t x,
                  size_t y)
            : data0(size * size), data1(size * size), length(size) {
        for (size_t i = 0; i < length; ++i) {
            for (size_t j = 0; j < length; ++j) {
                if (i == 0 || j == 0 || i == length - 1 || j == length - 1) {
                    this->operator[]({i, j}) = border_temp;
                } else if (i == x && j == y) {
                    this->operator[]({i, j}) = source_temp;
                } else {
                    this->operator[]({i, j}) = 0;
                }
            }
        }
    }

    std::vector<double> &get_current_buffer() {
        if (current_buffer == 0) return data0;
        return data1;
    }

    double &operator[](std::pair<size_t, size_t> index) {
        return get_current_buffer()[index.first * length + index.second];
    }
```

```
        double &operator[](std::tuple<Alt, size_t, size_t> index) {
            return current_buffer == 1 ? data0[std::get<1>(index) * length +
std::get<2>(index)] : data1[
                    std::get<1>(index) * length + std::get<2>(index)];
        }

        void switch_buffer() {
            current_buffer = !current_buffer;
        }
    };

    struct UpdateResult {
        bool stable;
        double temp;
    };

    UpdateResult update_single(size_t i, size_t j, Grid &grid, const State &state) {
        UpdateResult result{};
        if (i == 0 || j == 0 || i == state.room_size - 1 || j == state.room_size - 1) {
            result.temp = state.border_temp;
        } else if (i == state.source_x && j == state.source_y) {
            result.temp = state.source_temp;
        } else {
            auto sum = (grid[{i + 1, j}] + grid[{i - 1, j}] + grid[{i, j + 1}] + grid[{i, j -
1}]);
            switch (state.algo) {
                case Algorithm::Jacobi:
                    result.temp = 0.25 * sum;
                    break;
                case Algorithm::Sor:
                    result.temp = grid[{i, j}] + (1.0 / state.sor_constant) * (sum - 4.0 * grid[{i,
j}]);
                    break;
            }
        }
        result.stable = fabs(grid[{i, j}] - result.temp) < state.tolerance;
        return result;
    }

    bool calculate(const State &state, Grid &grid, int totalThreadSize) {
        std::vector<int> stateList(totalThreadSize, true);

        // total data size
        int dataSize = state.room_size*state.room_size;

        #pragma omp parallel
        {
            bool stabilized = true;
```

```
            int taskNum = omp_get_thread_num();

            // calculate start and end position at buffer
            int startPosition = (dataSize/totalThreadSize)*taskNum +
std::min(dataSize%totalThreadSize, taskNum);
            int endPosition = (dataSize/totalThreadSize)*(taskNum+1) +
std::min(dataSize%totalThreadSize, taskNum+1)-1;
            int jobSize = endPosition-startPosition+1;

            // calculate start i and j
            int startI = startPosition/state.room_size;
            int i, j, count;
            bool switchOnce = false;

            switch (state.algo) {
                case Algorithm::Jacobi:
                    i = startI;
                    j = startPosition%state.room_size;
                    count = 0;
                    for (; i < state.room_size; ++i) {
                        if(i!=startI){
                            j=0;
                        }
                        for (; j < state.room_size; ++j) {
                            // avoid extra calculation
                            if(count>=jobSize){
                                // break outside for loop
                                i = state.room_size;
                                break;
                            }
                            count++;

                            auto result = update_single(i, j, grid, state);
                            stabilized &= result.stable;
                            grid[{alt, i, j}] = result.temp;
                        }
                    }
                    break;
                case Algorithm::Sor:
                    for (auto k : {0, 1}) {
                        i = startI;
                        j = startPosition%state.room_size;
                        count = 0;
                        for (; i < state.room_size; i++) {
                            if(i!=startI){
                                j=0;
                            }
                            for (; j < state.room_size; j++) {
                                // avoid extra calculation
```

```
                            if(count>=jobSize){
                                // break outside for loop
                                i = state.room_size;
                                break;
                            }
                            count++;

                            if (k == ((i + j) & 1)) {
                                auto result = update_single(i, j, grid, state);
                                stabilized &= result.stable;
                                grid[{alt, i, j}] = result.temp;
                            } else {
                                grid[{alt, i, j}] = grid[{i, j}];
                            }
                        }
                    }
                    // wait until all threads finish calculation
                    #pragma omp barrier
                    if(taskNum==0 && switchOnce == false){
                        grid.switch_buffer();
                        switchOnce = true;
                    }
                    #pragma omp barrier
                }
            }

            stateList[taskNum] = stabilized;

            // printf("rank %d %d %d %d %d %d %d\n", taskNum, jobSize, starti, startj, curi,
j, stabilized);

        }

        bool statelizedAll = true;

        for(bool stabilized : stateList){
            // printf("checking %d\n", stabilized);
            statelizedAll &= stabilized;
        }

        grid.switch_buffer();
        return statelizedAll;
    };


} // namespace hdist
```

4) CUDA

a) main.cu

```
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <cstring>
#include <chrono>
#include <hdist/hdist.hpp>

template<typename ...Args>
void UNUSED(Args &&... args [[maybe_unused]]) {}

ImColor temp_to_color(double temp) {
    auto value = static_cast<uint8_t>(temp / 100.0 * 255.0);
    return {value, 0, 255 - value};
}

int main(int argc, char **argv) {
    // use 4 threadsd as default
    int totalThreadSize = 4;

    // fetch thread size from command line
    if(argc>=2){
        totalThreadSize = atoi(argv[1]);
    }

    // avoid invalid input
    if(totalThreadSize==0){
        totalThreadSize = 4;
    }

    printf("CUDA: launch %d threads\n", totalThreadSize);
    // test codes
    // int testCases[] = {200, 400, 800, 1600, 3200};
    // int testCaseIdx = 0, testCnt = 0;
    // static std::chrono::high_resolution_clock::time_point beginIteration, endIteration;

    UNUSED(argc, argv);
    bool first = true;
    bool finished = false;
    static hdist::State current_state, last_state;
    static std::chrono::high_resolution_clock::time_point begin, end;
    static const char* algo_list[2] = { "jacobi", "sor" };
    graphic::GraphicContext context{"Assignment 4"};
    hdist::Grid * gridPtr = new hdist::Grid{
        static_cast<size_t>(current_state.room_size),
        current_state.border_temp,
        current_state.source_temp,
        static_cast<size_t>(current_state.source_x),
```

```
        static_cast<size_t>(current_state.source_y)};
    hdist::Grid & grid = *gridPtr;
    context.run([&](graphic::GraphicContext *context [[maybe_unused]], SDL_Window *) {
        auto io = ImGui::GetIO();
        ImGui::SetNextWindowPos(ImVec2(0.0f, 0.0f));
        ImGui::SetNextWindowSize(io.DisplaySize);
        ImGui::Begin("Assignment 4", nullptr,
                     ImGuiWindowFlags_NoMove
                     | ImGuiWindowFlags_NoCollapse
                     | ImGuiWindowFlags_NoTitleBar
                     | ImGuiWindowFlags_NoResize);
        ImDrawList *draw_list = ImGui::GetWindowDrawList();
        ImGui::Text("Application average %.3f ms/frame (%.1f FPS)", 1000.0f /
ImGui::GetIO().Framerate,
                    ImGui::GetIO().Framerate);
        ImGui::DragInt("Room Size", &current_state.room_size, 10, 200, 1600, "%d");
        ImGui::DragFloat("Block Size", &current_state.block_size, 0.01, 0.1, 10, "%f");
        ImGui::DragFloat("Source Temp", &current_state.source_temp, 0.1, 0, 100, "%f");
        ImGui::DragFloat("Border Temp", &current_state.border_temp, 0.1, 0, 100, "%f");
        ImGui::DragInt("Source X", &current_state.source_x, 1, 1, current_state.room_size - 2,
"%d");
        ImGui::DragInt("Source Y", &current_state.source_y, 1, 1, current_state.room_size - 2,
"%d");
        ImGui::DragFloat("Tolerance", &current_state.tolerance, 0.01, 0.01, 1, "%f");
        ImGui::ListBox("Algorithm", reinterpret_cast<int *>(&current_state.algo), algo_list,
2);

        // testCnt++;

        // if(testCnt == 5){
        //     current_state.room_size = testCases[testCaseIdx];
        //     testCaseIdx = (testCaseIdx+1) % 5;
        //     testCnt = 0;
        // }

        if (current_state.algo == hdist::Algorithm::Sor) {
            ImGui::DragFloat("Sor Constant", &current_state.sor_constant, 0.01, 0.0, 20.0,
"%f");
        }

        // beginIteration = std::chrono::high_resolution_clock::now();

        if (current_state.room_size != last_state.room_size) {
            delete gridPtr;
            gridPtr = new hdist::Grid{
                static_cast<size_t>(current_state.room_size),
                current_state.border_temp,
                current_state.source_temp,
                static_cast<size_t>(current_state.source_x),
```

```
                static_cast<size_t>(current_state.source_y)};
            grid = *gridPtr;
            first = true;
        }

        if (!(current_state == last_state)) {
            last_state = current_state;
            finished = false;
        }

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

        if (!finished) {
            finished = hdist::calculate(current_state, grid, totalThreadSize);
            if (finished) end = std::chrono::high_resolution_clock::now();

            // endIteration = std::chrono::high_resolution_clock::now();
            // printf("thread %d\tone iteration %lld (ns)\troom size %d\n",
            // totalThreadSize,
std::chrono::duration_cast<std::chrono::nanoseconds>(endIteration - beginIteration).count(),
current_state.room_size);
        } else {
            ImGui::Text("stabilized in %lld ns",
std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count());
            printf("stabilized in %lld ns\n",
std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count());
        }

        const ImVec2 p = ImGui::GetCursorScreenPos();
        float x = p.x + current_state.block_size, y = p.y + current_state.block_size;
        for (size_t i = 0; i < current_state.room_size; ++i) {
            for (size_t j = 0; j < current_state.room_size; ++j) {
                auto temp = grid.fetch(i, j);
                auto color = temp_to_color(temp);
                draw_list->AddRectFilled(ImVec2(x, y), ImVec2(x + current_state.block_size, y
+ current_state.block_size), color);
                y += current_state.block_size;
            }
            x += current_state.block_size;
            y = p.y + current_state.block_size;
        }
        ImGui::End();
    });
}
```

b) hdist.hpp

```
#pragma once

namespace hdist {
    __device__
    int getBlockId() {
        return blockIdx.x + blockIdx.y * gridDim.x + gridDim.x * gridDim.y * blockIdx.z;
    }

    __device__
    int getLocalThreadId() {
        return (threadIdx.z * (blockDim.x * blockDim.y)) + (threadIdx.y * blockDim.x) +
threadIdx.x;
    }

    __device__
    int getThreadId() {
        int blockId = getBlockId();
        int localThreadId = getLocalThreadId();
        return blockId * (blockDim.x * blockDim.y * blockDim.z) + localThreadId;
    }

    enum class Algorithm : int {
        Jacobi = 0,
        Sor = 1
    };

    struct State {
        int room_size = 300;
        float block_size = 2;
        int source_x = room_size / 2;
        int source_y = room_size / 2;
        float source_temp = 100;
        float border_temp = 36;
        float tolerance = 0.02;
        float sor_constant = 4.0;
        Algorithm algo = hdist::Algorithm::Jacobi;

        bool operator==(const State &that) const{
            return
            (this->room_size == that.room_size) &&
            (this->block_size == that.block_size) &&
            (this->source_x == that.source_x) &&
            (this->source_y == that.source_y) &&
            (this->source_temp == that.source_temp) &&
            (this->border_temp == that.border_temp) &&
            (this->tolerance == that.tolerance) &&
```

```
        (this->sor_constant == that.sor_constant) &&
        (this->algo == that.algo);
    }
};

struct Alt {
};

constexpr static inline Alt alt{};

struct Grid {
    double *data0, *data1;
    size_t current_buffer = 0;
    size_t length;

    __host__
    explicit Grid(size_t size,
                  double border_temp,
                  double source_temp,
                  size_t x,
                  size_t y)
            : length(size) {

        data0 = (double*)malloc(size*size*sizeof(double));
        data1 = (double*)malloc(size*size*sizeof(double));

        for (size_t i = 0; i < length; ++i) {
            for (size_t j = 0; j < length; ++j) {
                if (i == 0 || j == 0 || i == length - 1 || j == length - 1) {
                    this->set(i, j, border_temp);
                } else if (i == x && j == y) {
                    this->set(i, j, source_temp);
                } else {
                    this->set(i, j, 0);
                }
            }
        }
    }

    __host__ __device__
    double* get_current_buffer() {
        if (current_buffer == 0) return data0;
        return data1;
    }

    __host__ __device__
    double fetch(int index1, int index2) {
        return get_current_buffer()[index1 * length + index2];
    }
```

```
        __host__ __device__
        double fetch(Alt alt, int index1, int index2) {
            return current_buffer == 1 ? data0[index1 * length + index2] : data1[
                    index1 * length + index2];
        }

        __host__ __device__
        void set(int index1, int index2, double setVal) {
            get_current_buffer()[index1 * length + index2] = setVal;
        }

        __host__ __device__
        void set(Alt alt, int index1, int index2, double setVal) {
            if(current_buffer == 1){
                data0[index1 * length + index2] = setVal;
            }
            else{
                data1[index1 * length + index2] = setVal;
            }
        }

        __host__ __device__
        void switch_buffer() {
            current_buffer = !current_buffer;
        }

        __host__
        ~Grid(){
            free(data0);
            free(data1);
        }
    };

    struct UpdateResult {
        bool stable;
        double temp;
    };

    __host__ __device__
    UpdateResult update_single(size_t i, size_t j, Grid &grid, const State &state) {
        UpdateResult result{};
        if (i == 0 || j == 0 || i == state.room_size - 1 || j == state.room_size - 1) {
            result.temp = state.border_temp;
        } else if (i == state.source_x && j == state.source_y) {
            result.temp = state.source_temp;
        } else {
            auto sum = (grid.fetch(i + 1, j) + grid.fetch(i - 1, j) + grid.fetch(i, j + 1) +
grid.fetch(i, j - 1));
```

```
            switch (state.algo) {
                case Algorithm::Jacobi:
                    result.temp = 0.25 * sum;
                    break;
                case Algorithm::Sor:
                    result.temp = grid.fetch(i, j) + (1.0 / state.sor_constant) * (sum - 4.0
* grid.fetch(i, j));
                    break;
            }
        }
        result.stable = fabs(grid.fetch(i, j) - result.temp) < state.tolerance;
        return result;
    }

    __global__
    void cuda_calculate(const State &state, Grid &grid, int totalThreadSize, bool * stateList){
        // total data size
        int dataSize = state.room_size*state.room_size;

        bool stabilized = true;

        int taskNum = getThreadId();

        // calculate start and end position at buffer
        int startPosition = (dataSize/totalThreadSize)*taskNum + min(dataSize%totalThreadSize,
taskNum);
        int endPosition = (dataSize/totalThreadSize)*(taskNum+1) +
min(dataSize%totalThreadSize, taskNum+1)-1;
        int jobSize = endPosition-startPosition+1;

        // calculate start i and j
        int startI = startPosition/state.room_size;
        int i, j, count;
        bool switchOnce = false;

        switch (state.algo) {
            case Algorithm::Jacobi:
                i = startI;
                j = startPosition%state.room_size;
                count = 0;
                for (; i < state.room_size; ++i) {
                    if(i!=startI){
                        j=0;
                    }
                    for (; j < state.room_size; ++j) {
                        // avoid extra calculation
                        if(count>=jobSize){
                            // break outside for loop
                            i = state.room_size;
```

```
                    break;
                }
                count++;

                auto result = update_single(i, j, grid, state);
                stabilized &= result.stable;
                grid.set(alt, i, j, result.temp);
            }
        }
        break;
    case Algorithm::Sor:
        for (auto k : {0, 1}) {
            i = startI;
            j = startPosition%state.room_size;
            count = 0;
            for (; i < state.room_size; i++) {
                if(i!=startI){
                    j=0;
                }
                for (; j < state.room_size; j++) {
                    // avoid extra calculation
                    if(count>=jobSize){
                        // break outside for loop
                        i = state.room_size;
                        break;
                    }
                    count++;

                    if (k == ((i + j) & 1)) {
                        auto result = update_single(i, j, grid, state);
                        stabilized &= result.stable;
                        grid.set(alt, i, j, result.temp);
                    } else {
                        grid.set(alt, i, j, grid.fetch(i, j));
                    }
                }
            }

            // make sure all threads finish calculation
            __syncthreads();
            if(taskNum==0 && switchOnce == false){
                    grid.switch_buffer();
                    switchOnce = true;
            }
            // make sure all threads finish calculation
            __syncthreads();

        }
}
```

```
        stateList[taskNum] = stabilized;
    }

    __host__
    bool calculate(const State &state, Grid &grid, int totalThreadSize) {

        // allocate state
        State * stateCopyPtr;
        cudaMallocManaged((void**)&stateCopyPtr, sizeof(state));
        cudaMemcpy(stateCopyPtr, &state, sizeof(state), cudaMemcpyHostToDevice);
        State & stateCopyReference = *stateCopyPtr;

        // allocate grid
        Grid * gridCopyPtr;
        double *data0, *data1;
        cudaMallocManaged((void**)&gridCopyPtr, sizeof(grid));
        cudaMallocManaged((void**)&data0, state.room_size*state.room_size*sizeof(double));
        cudaMallocManaged((void**)&data1, state.room_size*state.room_size*sizeof(double));
        cudaMemcpy(gridCopyPtr, &grid, sizeof(grid), cudaMemcpyHostToDevice);
        Grid & gridCopyReference = *gridCopyPtr;
        gridCopyReference.data0 = data0;
        gridCopyReference.data1 = data1;
        cudaMemcpy(gridCopyReference.get_current_buffer(), grid.get_current_buffer(),
state.room_size*state.room_size*sizeof(double), cudaMemcpyHostToDevice);

        // allocate stateList
        bool * stateList;
        cudaMallocManaged((void**)&stateList, totalThreadSize*sizeof(bool));
        memset(stateList, true, sizeof(stateList));

        cuda_calculate<<<1, totalThreadSize>>>(stateCopyReference, gridCopyReference,
totalThreadSize, stateList);
        cudaDeviceSynchronize();

        gridCopyReference.switch_buffer();
        grid.switch_buffer();

        // copy back the newest data
        memcpy(grid.get_current_buffer(), gridCopyReference.get_current_buffer(),
state.room_size*state.room_size*sizeof(double));

        bool statelizedAll = true;

        for(int i=0;i<totalThreadSize; i++){
            // printf("checking %d\n", stabilized);
            statelizedAll &= stateList[i];
        }
```

```
        // free allocated space
        cudaFree(stateCopyPtr);
        cudaFree(gridCopyPtr);
        cudaFree(data0);
        cudaFree(data1);
        cudaFree(stateList);

        return statelizedAll;
    };

} // namespace hdist
```

5) MPI with OpenMP

a) main.cpp

```
#include <graphic/graphic.hpp>
#include <imgui_impl_sdl.h>
#include <cstring>
#include <chrono>
#include <hdist/hdist.hpp>

template<typename ...Args>
void UNUSED(Args &&... args [[maybe_unused]]) {}

ImColor temp_to_color(double temp) {
    auto value = static_cast<uint8_t>(temp / 100.0 * 255.0);
    return {value, 0, 255 - value};
}

static MPI_Datatype MPI_State;

int main(int argc, char **argv) {
    // use 4 threadsd as default
    int totalThreadSize = 4;

    // fetch thread size from command line
    if(argc>=2){
        totalThreadSize = atoi(argv[1]);
    }

    // avoid invalid input
    if(totalThreadSize==0){
        totalThreadSize = 4;
    }


    int rank;
    int processSize;
    MPI_Init(&argc, &argv);
```

```
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &processSize);

    if(rank==0){
        printf("MPI+OpenMP: launch %d processes with %d threads\n", processSize,
totalThreadSize);
    }

    omp_set_num_threads(totalThreadSize);

    // initialize MPI struct
    const int nitems=9;
    int blocklengths[9] = {1,1,1,1,1,1,1,1,1};
    MPI_Datatype types[9] = {MPI_INT, MPI_FLOAT, MPI_INT, MPI_INT,
                MPI_FLOAT, MPI_FLOAT, MPI_FLOAT, MPI_FLOAT, MPI_INT};

    MPI_Aint offsets[9];

    offsets[0] = offsetof(hdist::State, room_size);
    offsets[1] = offsetof(hdist::State, block_size);
    offsets[2] = offsetof(hdist::State, source_x);
    offsets[3] = offsetof(hdist::State, source_y);
    offsets[4] = offsetof(hdist::State, source_temp);
    offsets[5] = offsetof(hdist::State, border_temp);
    offsets[6] = offsetof(hdist::State, tolerance);
    offsets[7] = offsetof(hdist::State, sor_constant);
    offsets[8] = offsetof(hdist::State, algo);

    MPI_Type_create_struct(nitems, blocklengths, offsets, types, &MPI_State);
    MPI_Type_commit(&MPI_State);

    UNUSED(argc, argv);
    bool first = true;
    bool finished = false;
    static hdist::State current_state, last_state;
    auto grid = hdist::Grid{
            static_cast<size_t>(current_state.room_size),
            current_state.border_temp,
            current_state.source_temp,
            static_cast<size_t>(current_state.source_x),
            static_cast<size_t>(current_state.source_y)};

    if(rank == 0){
        // test codes
        // int testCases[] = {200, 400, 800, 1600, 3200};
        // int testCaseIdx = 0, testCnt = 0;
        // static std::chrono::high_resolution_clock::time_point beginIteration,
endIteration;
```

```
        static std::chrono::high_resolution_clock::time_point begin, end;
        static const char* algo_list[2] = { "jacobi", "sor" };
        graphic::GraphicContext context{"Assignment 4"};
        context.run([&](graphic::GraphicContext *context [[maybe_unused]], SDL_Window *) {
            auto io = ImGui::GetIO();
            ImGui::SetNextWindowPos(ImVec2(0.0f, 0.0f));
            ImGui::SetNextWindowSize(io.DisplaySize);
            ImGui::Begin("Assignment 4", nullptr,
                         ImGuiWindowFlags_NoMove
                         | ImGuiWindowFlags_NoCollapse
                         | ImGuiWindowFlags_NoTitleBar
                         | ImGuiWindowFlags_NoResize);
            ImDrawList *draw_list = ImGui::GetWindowDrawList();
            ImGui::Text("Application average %.3f ms/frame (%.1f FPS)", 1000.0f /
ImGui::GetIO().Framerate,
                        ImGui::GetIO().Framerate);
            ImGui::DragInt("Room Size", &current_state.room_size, 10, 200, 1600, "%d");
            ImGui::DragFloat("Block Size", &current_state.block_size, 0.01, 0.1, 10, "%f");
            ImGui::DragFloat("Source Temp", &current_state.source_temp, 0.1, 0, 100, "%f");
            ImGui::DragFloat("Border Temp", &current_state.border_temp, 0.1, 0, 100, "%f");
            ImGui::DragInt("Source X", &current_state.source_x, 1, 1, current_state.room_size
- 2, "%d");
            ImGui::DragInt("Source Y", &current_state.source_y, 1, 1, current_state.room_size
- 2, "%d");
            ImGui::DragFloat("Tolerance", &current_state.tolerance, 0.01, 0.01, 1, "%f");
            ImGui::ListBox("Algorithm", reinterpret_cast<int *>(&current_state.algo),
algo_list, 2);

            // testCnt++;

            // if(testCnt == 5){
            //     current_state.room_size = testCases[testCaseIdx];
            //     testCaseIdx = (testCaseIdx+1) % 5;
            //     testCnt = 0;
            // }

            if (current_state.algo == hdist::Algorithm::Sor) {
                ImGui::DragFloat("Sor Constant", &current_state.sor_constant, 0.01, 0.0, 20.0,
"%f");
            }

            // beginIteration = std::chrono::high_resolution_clock::now();

            // broadcast current state
            MPI_Bcast(&current_state, 1, MPI_State, 0, MPI_COMM_WORLD);

            // change size regenerate
            if (current_state.room_size != last_state.room_size) {
                grid = hdist::Grid{
```

```
                    static_cast<size_t>(current_state.room_size),
                    current_state.border_temp,
                    current_state.source_temp,
                    static_cast<size_t>(current_state.source_x),
                    static_cast<size_t>(current_state.source_y)};
            first = true;
        }

        if (current_state != last_state) {
            last_state = current_state;
            finished = false;
        }

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

        if (!finished) {
            int dataSize = current_state.room_size*current_state.room_size;

            // bcast get_current_buffer
            MPI_Bcast(grid.get_current_buffer().data(), dataSize, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

            // do calculation
            finished = hdist::calculate(current_state, grid, totalThreadSize);

            bool localFinished = finished;

            // reduce and broadcast finished
            MPI_Reduce(&localFinished, &finished, 1, MPI_C_BOOL, MPI_LAND, 0,
MPI_COMM_WORLD);
            MPI_Bcast(&finished, 1, MPI_C_BOOL, 0, MPI_COMM_WORLD);

            // gatherv get_current_buffer result from all ranks
            // calculate the job size of each rank
            std::vector<int> taskSizeList;
            for(int i=0;i<processSize;i++){
                int startPosition = (dataSize/processSize)*i +
std::min(dataSize%processSize, i);
                int endPosition = (dataSize/processSize)*(i+1) +
std::min(dataSize%processSize, i+1)-1;
                taskSizeList.push_back(endPosition-startPosition+1);
            }

            // calculate the strips of received data packages
            std::vector<int> strips;
```

```
                strips.push_back(0);
                for(int i=1;i<processSize;i++){
                    strips.push_back(strips[i-1]+taskSizeList[i-1]);
                }

                // calcualte job size of itself
                int startPosition = (dataSize/processSize)*rank +
std::min(dataSize%processSize, rank);
                int endPosition = (dataSize/processSize)*(rank+1) +
std::min(dataSize%processSize, rank+1)-1;
                int jobSize = endPosition-startPosition+1;

                // rank 0 gather calculated result
                MPI_Gatherv(grid.get_current_buffer().data()+startPosition, jobSize,
MPI_DOUBLE,
                    grid.get_current_buffer().data(), taskSizeList.data(), strips.data(),
MPI_DOUBLE, 0, MPI_COMM_WORLD);

                if (finished) end = std::chrono::high_resolution_clock::now();

                // endIteration = std::chrono::high_resolution_clock::now();
                // printf("threads %d\tone iteration %lld (ns)\troom size %d\n",
                // processSize*totalThreadSize,
std::chrono::duration_cast<std::chrono::nanoseconds>(endIteration - beginIteration).count(),
current_state.room_size);

            } else {
                ImGui::Text("stabilized in %lld ns",
std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count());
                printf("stabilized in %lld ns\n",
std::chrono::duration_cast<std::chrono::nanoseconds>(end - begin).count());
            }

            const ImVec2 p = ImGui::GetCursorScreenPos();
            float x = p.x + current_state.block_size, y = p.y + current_state.block_size;
            for (size_t i = 0; i < current_state.room_size; ++i) {
                for (size_t j = 0; j < current_state.room_size; ++j) {
                    auto temp = grid[{i, j}];
                    auto color = temp_to_color(temp);
                    draw_list->AddRectFilled(ImVec2(x, y), ImVec2(x +
current_state.block_size, y + current_state.block_size), color);
                    y += current_state.block_size;
                }
                x += current_state.block_size;
                y = p.y + current_state.block_size;
            }
            ImGui::End();
        });
    }
```

```
    else{
        while(true){
            // receive current state
            MPI_Bcast(&current_state, 1, MPI_State, 0, MPI_COMM_WORLD);

            // change size regenerate
            if (current_state.room_size != last_state.room_size) {
                grid = hdist::Grid{
                        static_cast<size_t>(current_state.room_size),
                        current_state.border_temp,
                        current_state.source_temp,
                        static_cast<size_t>(current_state.source_x),
                        static_cast<size_t>(current_state.source_y)};
                first = true;
            }

            if (current_state != last_state) {
                last_state = current_state;
                finished = false;
            }

            if (first) {
                first = false;
                finished = false;
            }

            if (!finished) {
                int dataSize = current_state.room_size*current_state.room_size;

                // bcast get_current_buffer
                MPI_Bcast(grid.get_current_buffer().data(), dataSize, MPI_DOUBLE, 0,
MPI_COMM_WORLD);

                // do calculation
                finished = hdist::calculate(current_state, grid, totalThreadSize);

                // reduce and receive finished
                MPI_Reduce(&finished, nullptr, 1, MPI_C_BOOL, MPI_LAND, 0, MPI_COMM_WORLD);
                MPI_Bcast(&finished, 1, MPI_C_BOOL, 0, MPI_COMM_WORLD);

                // send get_current_buffer result to rank 0
                // calcualte job size of itself
                int startPosition = (dataSize/processSize)*rank +
std::min(dataSize%processSize, rank);
                int endPosition = (dataSize/processSize)*(rank+1) +
std::min(dataSize%processSize, rank+1)-1;
                int jobSize = endPosition-startPosition+1;

                // send calculated result
```

```
                MPI_Gatherv(grid.get_current_buffer().data()+startPosition, jobSize,
MPI_DOUBLE, nullptr, nullptr, nullptr,
                    MPI_DOUBLE, 0, MPI_COMM_WORLD);
            }

        }
    }

    // if ploting windows closed, rank 0 would then stop all other processes
    if(rank==0){
        printf("aborting all processes\n");
        MPI_Abort(MPI_COMM_WORLD, 0);
    }

    MPI_Type_free(&MPI_State);
    MPI_Finalize();
}
```

b) hdist.hpp

```
#pragma once

#include <vector>
#include <mpi.h>
#include <omp.h>

namespace hdist {

    enum class Algorithm : int {
        Jacobi = 0,
        Sor = 1
    };

    struct State {
        int room_size = 300;
        float block_size = 2;
        int source_x = room_size / 2;
        int source_y = room_size / 2;
        float source_temp = 100;
        float border_temp = 36;
        float tolerance = 0.02;
        float sor_constant = 4.0;
        Algorithm algo = hdist::Algorithm::Jacobi;

        bool operator==(const State &that) const = default;
    };

    struct Alt {
```

```
    };

    constexpr static inline Alt alt{};

    struct Grid {
        std::vector<double> data0, data1;
        size_t current_buffer = 0;
        size_t length;

        explicit Grid(size_t size,
                      double border_temp,
                      double source_temp,
                      size_t x,
                      size_t y)
                : data0(size * size), data1(size * size), length(size) {
            for (size_t i = 0; i < length; ++i) {
                for (size_t j = 0; j < length; ++j) {
                    if (i == 0 || j == 0 || i == length - 1 || j == length - 1) {
                        this->operator[]({i, j}) = border_temp;
                    } else if (i == x && j == y) {
                        this->operator[]({i, j}) = source_temp;
                    } else {
                        this->operator[]({i, j}) = 0;
                    }
                }
            }
        }

        std::vector<double> &get_current_buffer() {
            if (current_buffer == 0) return data0;
            return data1;
        }

        std::vector<double> &get_alternate_buffer() {
            if (current_buffer == 0) return data1;
            return data0;
        }

        double &operator[](std::pair<size_t, size_t> index) {
            return get_current_buffer()[index.first * length + index.second];
        }

        double &operator[](std::tuple<Alt, size_t, size_t> index) {
            return current_buffer == 1 ? data0[std::get<1>(index) * length +
std::get<2>(index)] : data1[
                    std::get<1>(index) * length + std::get<2>(index)];
        }

        void switch_buffer() {
```

```
            current_buffer = !current_buffer;
        }
    };

    struct UpdateResult {
        bool stable;
        double temp;
    };

    UpdateResult update_single(size_t i, size_t j, Grid &grid, const State &state) {
        UpdateResult result{};
        if (i == 0 || j == 0 || i == state.room_size - 1 || j == state.room_size - 1) {
            result.temp = state.border_temp;
        } else if (i == state.source_x && j == state.source_y) {
            result.temp = state.source_temp;
        } else {
            auto sum = (grid[{i + 1, j}] + grid[{i - 1, j}] + grid[{i, j + 1}] + grid[{i, j -
1}]);
            switch (state.algo) {
                case Algorithm::Jacobi:
                    result.temp = 0.25 * sum;
                    break;
                case Algorithm::Sor:
                    result.temp = grid[{i, j}] + (1.0 / state.sor_constant) * (sum - 4.0 * grid[{i,
j}]);
                    break;
            }
        }
        result.stable = fabs(grid[{i, j}] - result.temp) < state.tolerance;
        return result;
    }

    bool calculate(const State &state, Grid &grid, int totalThreadSize) {
        bool stabilized = true;
        int rank;
        int processSize;
        std::vector<int> stateList(totalThreadSize, true);

        MPI_Comm_rank(MPI_COMM_WORLD, &rank);
        MPI_Comm_size(MPI_COMM_WORLD, &processSize);

        int dataSize = state.room_size*state.room_size;

        // calculate start and end position at buffer
        int processStartPosition = (dataSize/processSize)*rank +
std::min(dataSize%processSize, rank);
        int processEndPosition = (dataSize/processSize)*(rank+1) +
std::min(dataSize%processSize, rank+1)-1;
        int processTaskSize = processEndPosition - processStartPosition + 1;
```

```
        #pragma omp parallel
        {
            int taskNum = omp_get_thread_num();

            int startPosition = (processTaskSize/totalThreadSize)*taskNum +
std::min(processTaskSize%totalThreadSize, taskNum) + processStartPosition;
            int endPosition = (processTaskSize/totalThreadSize)*(taskNum+1) +
std::min(processTaskSize%totalThreadSize, taskNum+1)-1 + processStartPosition;
            int jobSize = endPosition-startPosition+1;

            // calculate start i and j
            int startI = startPosition/state.room_size;
            int i, j, count, endI;
            bool switchOnce = false;
            int starti, startj, curi;

            switch (state.algo) {
                case Algorithm::Jacobi:
                    i = startI;
                    j = startPosition%state.room_size;
                    starti = i;
                    startj = j;
                    count = 0;
                    for (; i < state.room_size; ++i) {
                        if(i!=startI){
                            j=0;
                        }
                        curi = i;
                        for (; j < state.room_size; ++j) {
                            // avoid extra calculation
                            if(count>=jobSize){
                                // break outside for loop
                                i = state.room_size;
                                break;
                            }
                            count++;

                            auto result = update_single(i, j, grid, state);
                            stabilized &= result.stable;
                            grid[{alt, i, j}] = result.temp;
                        }
                    }
                    break;
                case Algorithm::Sor:
                    for (auto k : {0, 1}) {
                        i = startI;
                        j = startPosition%state.room_size;
                        count = 0;
```

```
                    // calculate the left and top extra points
                    if(switchOnce==false){
                        // left points
                        if(i!=0){
                            int x = i-1;
                            for(int y=j;y<state.room_size;y++){
                                if (k == ((x + y) & 1)) {
                                    auto result = update_single(x, y, grid, state);
                                    // stabilized &= result.stable;
                                    grid[{alt, x, y}] = result.temp;
                                } else {
                                    grid[{alt, x, y}] = grid[{x, y}];
                                }
                            }
                        }

                        // top points
                        for(int y=0; y<j;y++){
                            if (k == ((i + y) & 1)) {
                                auto result = update_single(i, y, grid, state);
                                // stabilized &= result.stable;
                                grid[{alt, i, y}] = result.temp;
                            } else {
                                grid[{alt, i, y}] = grid[{i, y}];
                            }
                        }
                    }

                    for (; i < state.room_size; i++) {
                        if(i!=startI){
                            j=0;
                        }

                        for (; j < state.room_size; j++) {
                            // avoid extra calculation
                            if(count>=jobSize){
                                // break outside for loop
                                endI = i;
                                i = state.room_size;
                                break;
                            }
                            count++;

                            if (k == ((i + j) & 1)) {
                                auto result = update_single(i, j, grid, state);
                                stabilized &= result.stable;
                                grid[{alt, i, j}] = result.temp;
                            } else {
```

```
                grid[{alt, i, j}] = grid[{i, j}];
            }
        }
    }

    // calculate the right and bottom extra points
    if(switchOnce==false){
        // last point reach the bottom exactly
        int rightJ = j;
        int rightI = endI;
        if(j==0){
            rightJ = state.room_size-1;
            // block bottom points calculation
            j = state.room_size;
        }
        else{
            rightI++;
        }

        // calculate right points
        if(rightI<state.room_size){
            int x = rightI;

            for(int y=0;y<rightJ; y++){
                if (k == ((x + y) & 1)) {
                    auto result = update_single(x, y, grid, state);
                    // stabilized &= result.stable;
                    grid[{alt, x, y}] = result.temp;
                } else {
                    grid[{alt, x, y}] = grid[{x, y}];
                }
            }
        }

        // calculate bottom points
        for(int y=j;y<state.room_size;y++){
            if (k == ((endI + y) & 1)) {
                auto result = update_single(endI, y, grid, state);
                // stabilized &= result.stable;
                grid[{alt, endI, y}] = result.temp;
            } else {
                grid[{alt, endI, y}] = grid[{endI, y}];
            }
        }

    }

    // wait until all threads finish calculation
    #pragma omp barrier
```

```
                    if(taskNum==0 && switchOnce == false){
                        grid.switch_buffer();
                    }

                    switchOnce = true;

                    #pragma omp barrier

                }
            }

            stateList[taskNum] = stabilized;

            // printf("%d %d %d %d %d %d [%d]\n", rank, taskNum, starti, startj, curi, j,
stabilized);

        }

        bool statelizedAll = true;

        for(bool stabilized : stateList){
            // printf("checking %d\n", stabilized);
            statelizedAll &= stabilized;
        }

        grid.switch_buffer();

        return statelizedAll;
    };


} // namespace hdist
```