The Chinese University of Hong Kong, Shenzhen

CSC4005

Distributed and Parallel Computing

Assignment 4 Heat Distribution Simulation

Author:
Shi Wenlan

Student Number:
119010265

November 30, 2022

Contents

1 Introduction

In this assignment, I programmed 7 versions of programs to simulate the temperature in a room (inlcuding bonus):

1. A sequential version
2. An OpenMP version
3. A Pthread versoin
4. A CUDA version
5. Two MPI version (Broadcast version & Send-Recv version)
6. A MPI + OpenMP version (bonus)

Introduction of Heat Distribution Simulation: The room has four walls and a fireplace. The temperature of the wall is constant at 20°C, and the temperature of the fireplace is constant at 100°C. Programs use Jacobi iteration to compute the temperature inside the room and plot temperature contours at 5°C intervals in each iteration.

The room is divided into a fine mesh of points, $h_{i,j}$. The temperature at the an inside point is calculated as the average of the four neighboring points, i.e.

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

.

The programs would terminate after a fixed number of itertaions (300 in my code), or until the difference between iterations of a point is less than some very small prescribed amount (*threshold* in *src/headers/physics.h*, the code of convergence check has been implemented and temporarily commented).

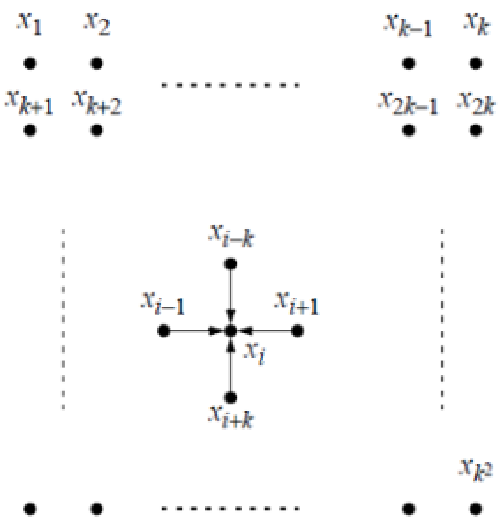


Figure 1: Heat Distribution Simulation

2 Method

2.1 Flow charts

The computation part of Heat Distribution Simulation is as following:

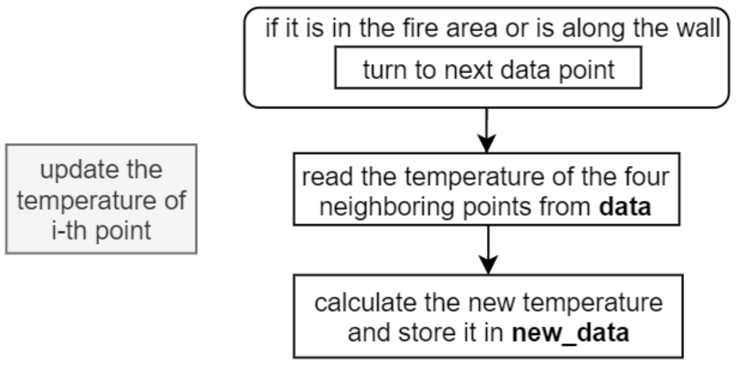

if it is in the fire area or is along the wall
turn to next data point
read the temperature of the four neighboring points from **data**
calculate the new temperature and store it in **new_data**
update the temperature of i-th point

Figure 2: Computation Part

In order to avoid data dependency and the overhead of data copy, two arrays, *data_odd* and *data_even*, are introduced to store old data and new data alternately. In other words, in the odd-th iteration, *data_odd* stores old data (*data* in *update*) and *data_even* stores new data (*new_data* in *update*). In the even-th iteration, *data_even* stores old data (*data* in *update*) and *data_add* stores new data (*new_data* in *update*).

The general idea flow charts of my 6 versions of program are as following:

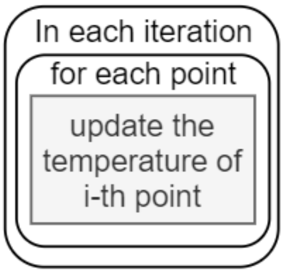


Figure 3: General Flow Chart of Sequential Version

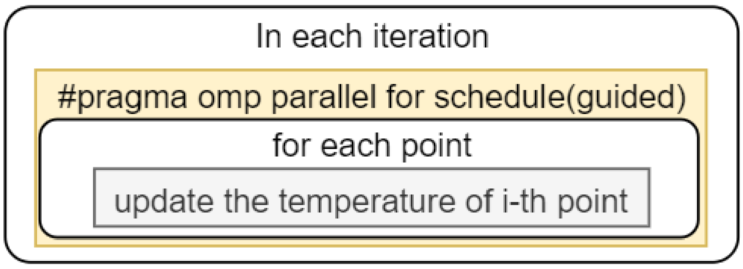

In each iteration
#pragma omp parallel for schedule(guided)
for each point
update the temperature of i-th point

Figure 4: General Flow Chart of OpenMP Version

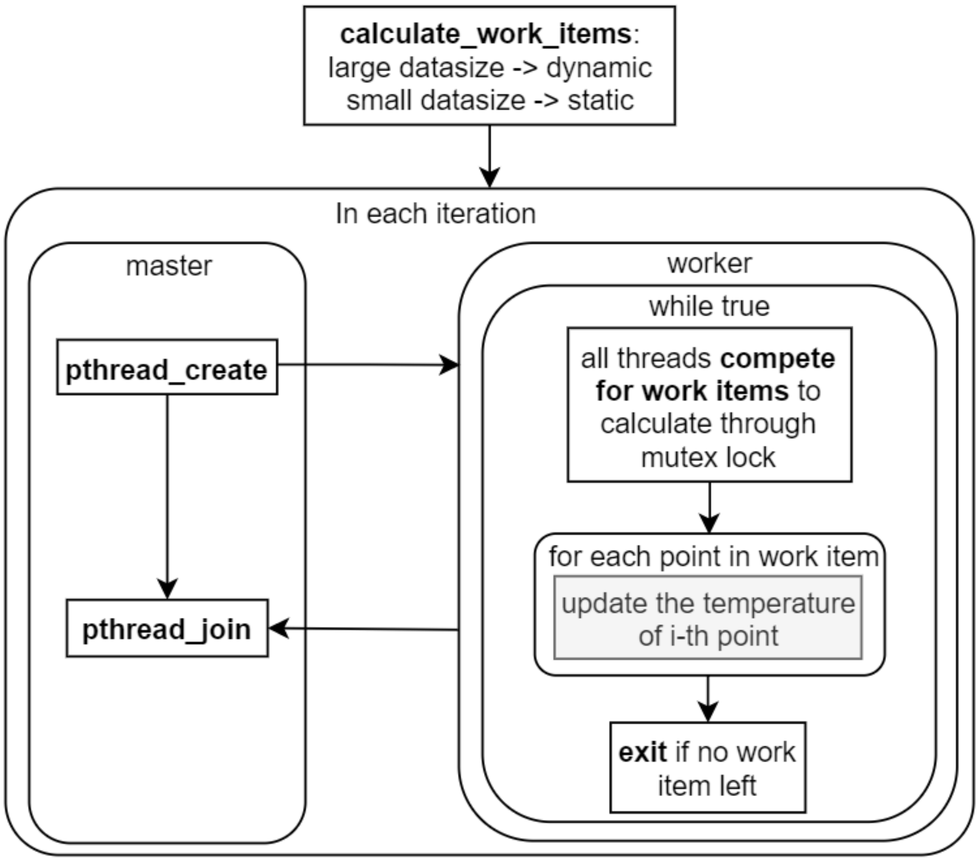

calculate_work_items:
large datasize -> dynamic
small datasize -> static
In each iteration
master
pthread_create
pthread_join
worker
while true
all threads compete for work items to calculate through mutex lock
for each point in work item
update the temperature of i-th point
exit if no work item left

Figure 5: General Flow Chart of Pthread Version

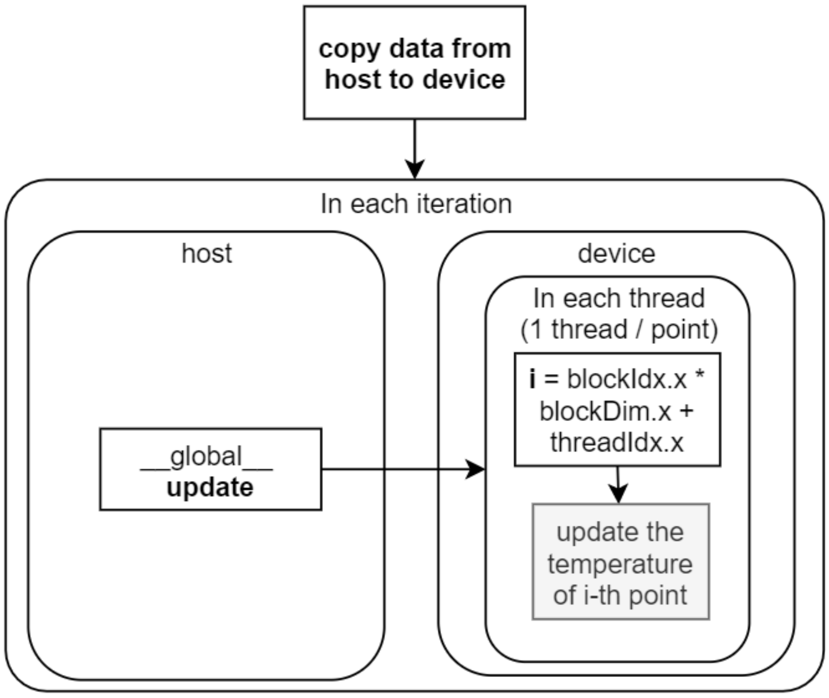

copy data from host to device
In each iteration
host
device
In each thread (1 thread / point)
i = blockIdx.x * blockDim.x + threadIdx.x
__global__ update
update the temperature of i-th point

Figure 6: General Flow Chart of CUDA Version

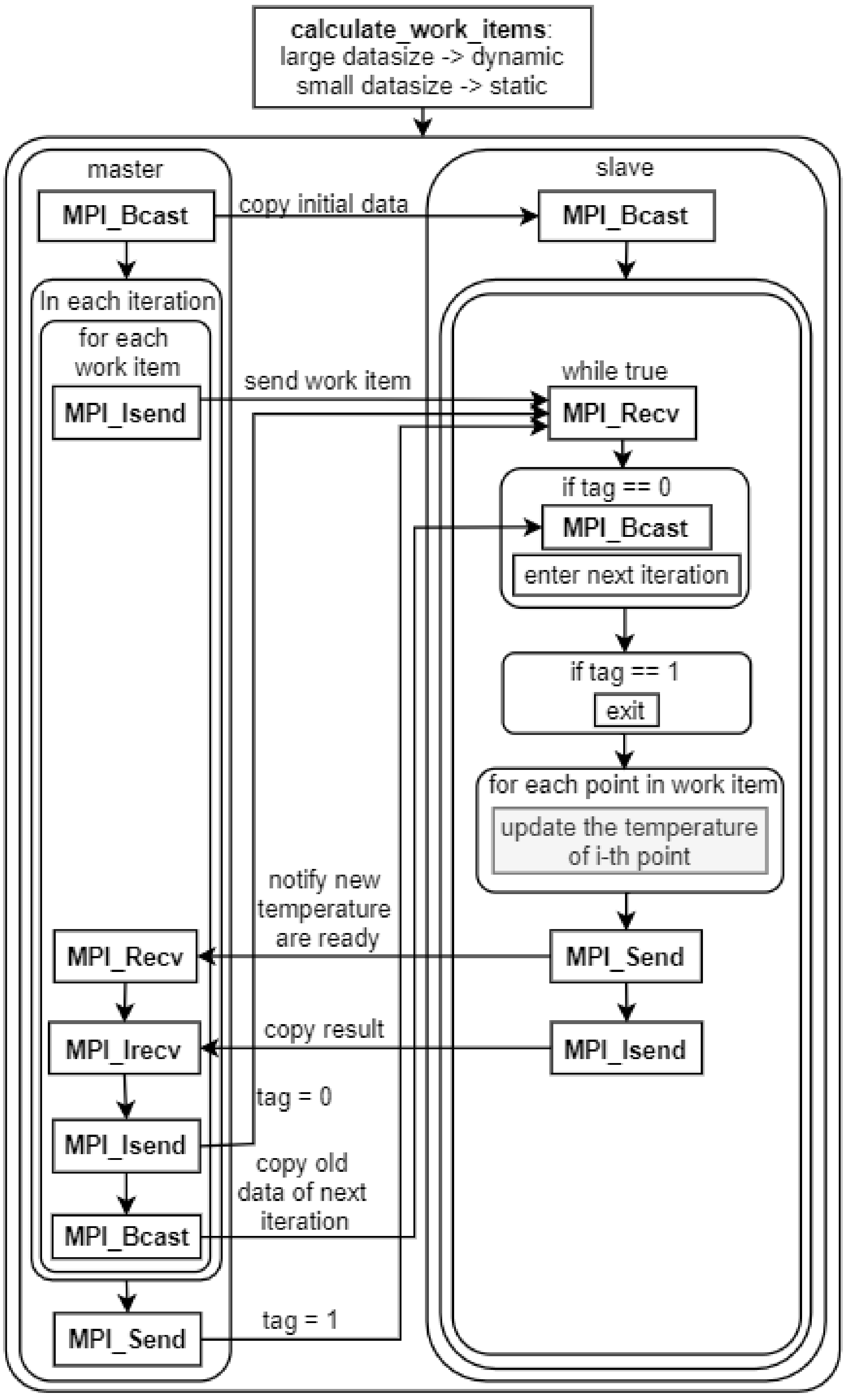


Figure 7: General Flow Chart of MPI Broadcast Version

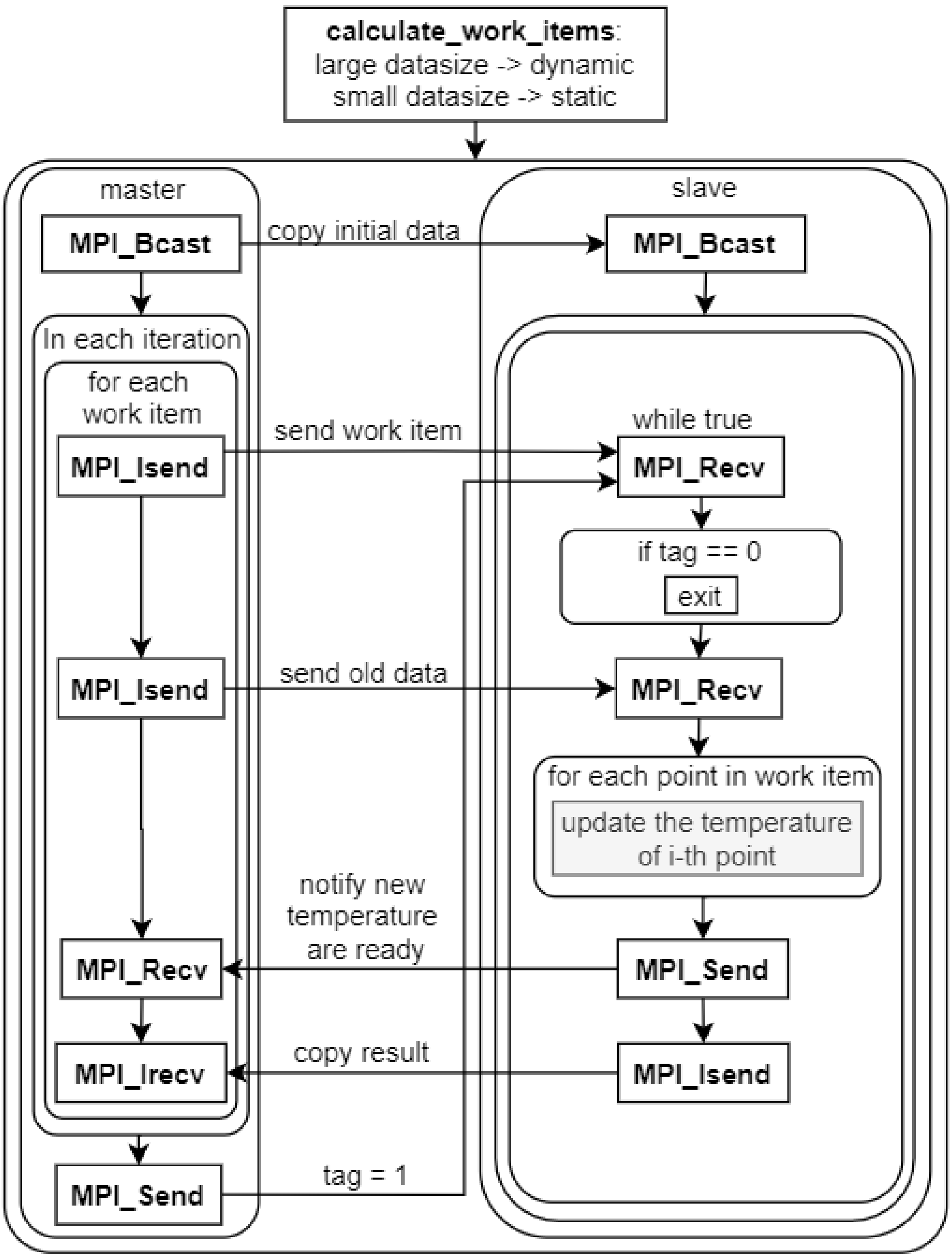


Figure 8: General Flow Chart of MPI SendRecv Version

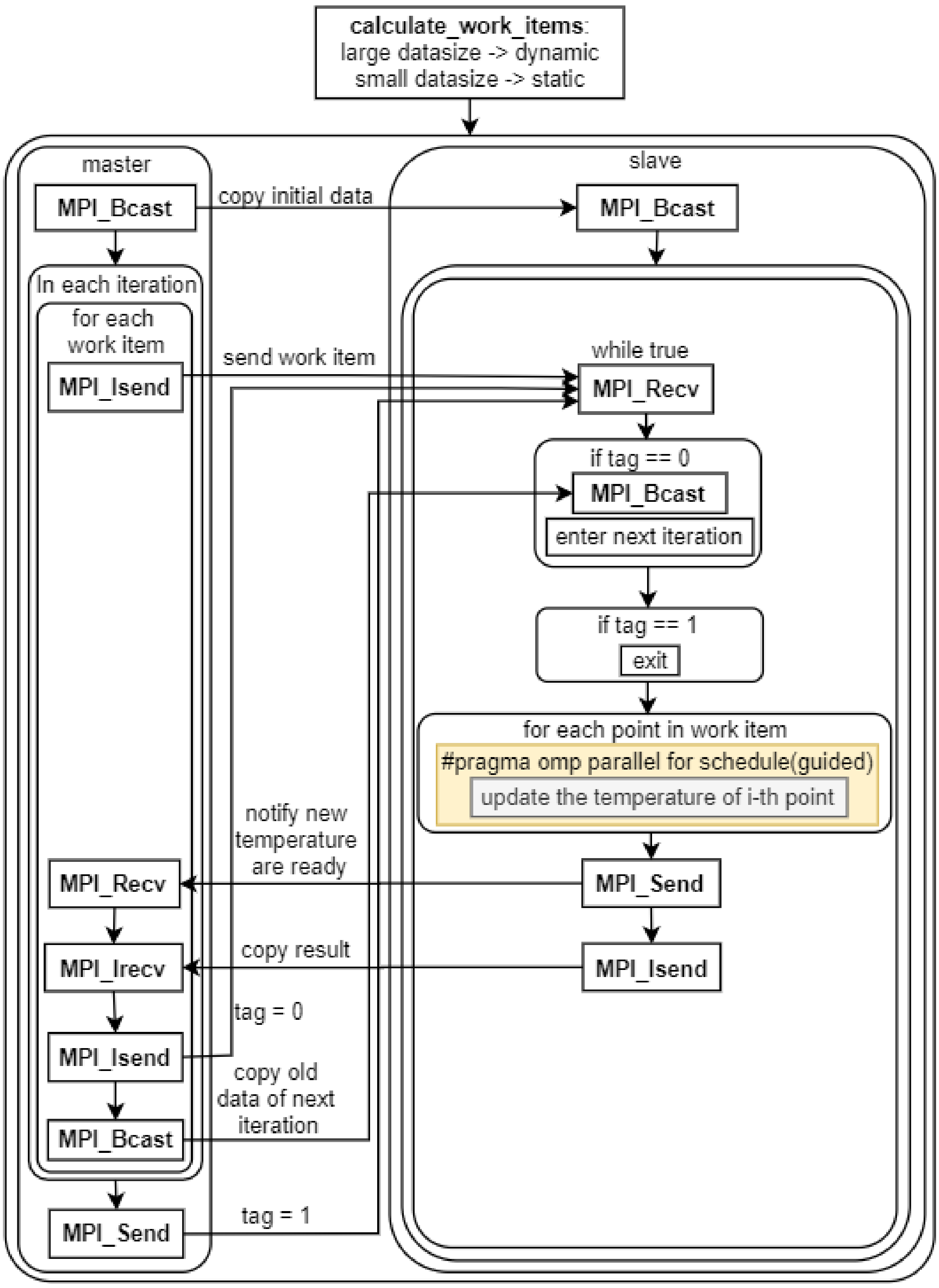


Figure 9: General Flow Chart of MPI+OpenMP Version

2.2 Extra explainations

The following are some extra explainations:

Why to use dynamic work item allocation: In this simulation, the calculation of some points is much less than that of others: the temperature of the points along the wall or in the fire area are constant, so there is no need to calculate. This will cause the actual workload of different workitem to be inconsistent. Therefore, dynamic allocation should be used to balance the load.

Two MPI versions: I first wrote the broadcast version, and when I experimented, I found that the overhead of the data copy was too large to show any acceleration effect, so I implemented a sendrecv version. The former copies all old data to the child processes at the beginning of each iteration, specifying the calculated data interval when assigning workitems. The latter copies old data on demand when assigning WorkItems.

CUDA version's *shouldCal* check: In CUDA logic, each point corresponds to a thread. Therefore, it cannot skip the calculation of points along the wall by limiting the traversal range. So I wrote a function *shouldCal* for it that checks whether the thread can skip the calculation of the point which it is responsible.

MPI SendRecv version's *shouldCal* check: In this version, the data the main process transmits to child processes includes the rows arranged by the workitem, and one row before and after it. And the result that the child processes transmit back to the main process includes only the rows arranged by the workitem. In this situation, it cannot skip the calculation of points along the wall by limiting the traversal range, too. Because regardless of whether this point needs to be calculated, the program needs to fill in something in the *new_data*: copy the content in the *data* to the *new_data* as it can skip calculation. So I wrote a special *shouldCal* for it as well.

How to add OpenMP to MPI: Based on MPI broadcast version, add "*#pragma omp parallel for schedule(guided)*" to the for loop in computation parts in child processes.

The memory of each MPI process is independent with each other, and is shared by its own OpenMP threads.

3 Result

3.1 How to run my code

If you want to reproduce the experiment in Subsection *Output screenshots***:** Open source code folder in the terminal, garantee that there are enough processors available, then type in the following instruction to run the script.

```
/bin/bash show.sh
```

Note that if enough resources are not prepared, the program will not report an error, but logically different processes and threads will take turns to use the CPU to simulate multi-threading and multi-process, and the context exchange caused by this will generate a large overhead and make the program run extremely slow. Anyway, the program results are still correct.

If you want to reproduce the experiment in Subsection *Raw data***:** Open source code folder in the terminal, garantee that there are enough processors available, then type in the following instruction to run the script.

```
/bin/bash slurm/cuda.sh
/bin/bash slurm/mpi_openmp.sh
/bin/bash slurm/mpi_broadcast.sh
/bin/bash slurm/mpi_sendrecv.sh
/bin/bash slurm/openmp.sh
/bin/bash slurm/pthread.sh
/bin/bash slurm/seq.sh
```

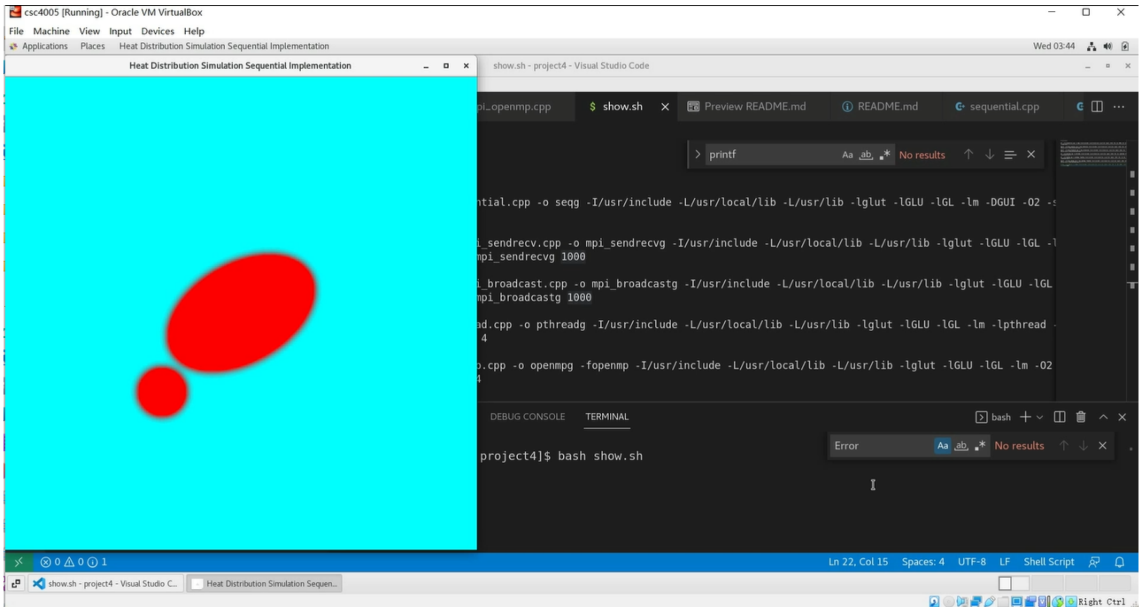
If you want to run the code more flexibly: Please refer to README.md for more detailed information.

3.2 Output screenshots

The expected output of the program is: the edges of the red area representing the fire area are blurred and expanded.

The CUDA GUI version is recorded on visual studio 2019 on the host machine (Windows), since the virtual machine does not support CUDA compile, the cluster does not support GUI display, and the CUDA simulator does not support C++11, which is too inconvenient to use. The VS project folder *CUDA* 11.1 *Heat Distribution* is attached. And the CUDA version which can be run on the virtual machine and the cluster is in *src* folder. Other GUI versions are recorded on the virtual machine.

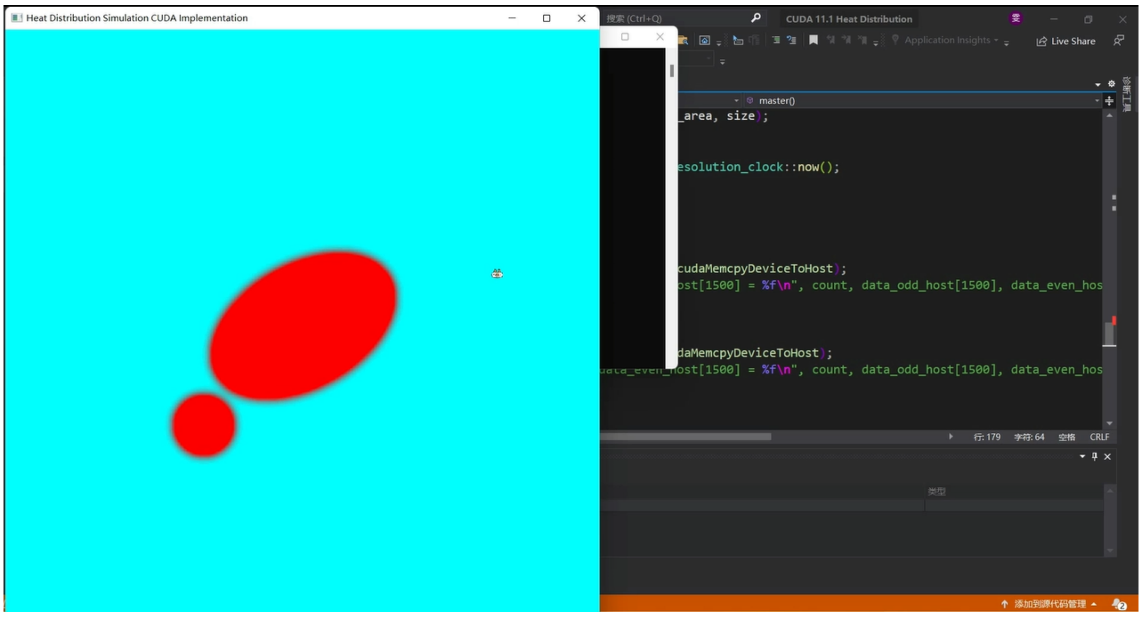
Here are some screenshots of GUI version compile and run of data size 1000:


csc4005 [Running] - Oracle VM VirtualBox
File Machine View Input Devices Help
Applications Places Heat Distribution Simulation Sequential Implementation
Wed 03:44
Heat Distribution Simulation Sequential Implementation
show.sh - project4 - Visual Studio Code
pi_openmp.cpp
show.sh
Preview README.md
README.md
sequential.cpp
printf
No results
DEBUG CONSOLE
TERMINAL
bash
Error
No results
Ln 22, Col 15 Spaces: 4 UTF-8 LF Shell Script
show.sh - project4 - Visual Studio C...
Heat Distribution Simulation Sequen...
Right Ctrl

```
ntial.cpp -o seqg -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -lm -DGUI -O2 -s

i_sendrecv.cpp -o mpi_sendrecvg -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -l
mpi_sendrecvg 1000

i_broadcast.cpp -o mpi_broadcastg -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL
mpi_broadcastg 1000

ad.cpp -o pthreadg -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -lm -lpthread -
 4

o.cpp -o openmpg -fopenmp -I/usr/include -L/usr/local/lib -L/usr/lib -lglut -lGLU -lGL -lm -O2
4
```

```
project4]$ bash show.sh
```

Figure 10: Screenshot 1


Heat Distribution Simulation CUDA Implementation
搜索 (Ctrl+Q)
CUDA 11.1 Heat Distribution
Application Insights
Live Share
master()
行: 179 字符: 64 空格 CRLF
类型
添加到源代码管理

```
_area, size);

esolution_clock::now();

cudaMemcpyDeviceToHost);
ost[1500] = %f\n", count, data_odd_host[1500], data_even_hos

daMemcpyDeviceToHost);
data_even_host[1500] = %f\n", count, data_odd_host[1500], data_even_hos
```

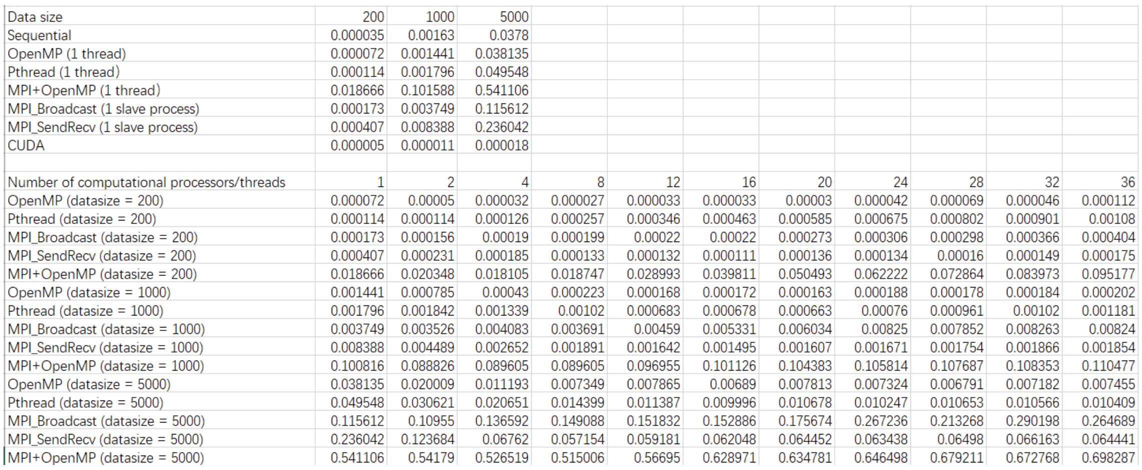
Figure 11: Screenshot 2

See Record.mp4 for the complete video.

3.3 Raw data

If convergence check is enabled, the number of iteration used by programs with different data sizes is different, so average computation time in each iteration is the most appropriate measurement.

The attached code enables fixed 300 iterations in order to control the running time of the display.

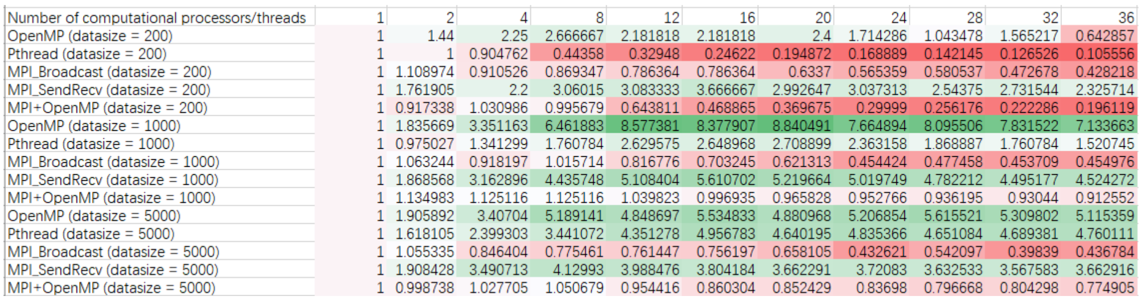


Data size	200	1000	5000								
Sequential	0.000035	0.00163	0.0378								
OpenMP (1 thread)	0.000072	0.001441	0.038135								
Pthread (1 thread)	0.000114	0.001796	0.049548								
MPI+OpenMP (1 thread)	0.018666	0.101588	0.541106								
MPI_Broadcast (1 slave process)	0.000173	0.003749	0.115612								
MPI_SendRecv (1 slave process)	0.000407	0.008388	0.236042								
CUDA	0.000005	0.000011	0.000018								
Number of computational processors/threads	1	2	4	8	12	16	20	24	28	32	36
OpenMP (datasize = 200)	0.000072	0.00005	0.000032	0.000027	0.000033	0.000033	0.00003	0.000042	0.000069	0.000046	0.000112
Pthread (datasize = 200)	0.000114	0.000114	0.000126	0.000257	0.000346	0.000463	0.000585	0.000675	0.000802	0.000901	0.00108
MPI_Broadcast (datasize = 200)	0.000173	0.000156	0.00019	0.000199	0.00022	0.00022	0.000273	0.000306	0.000298	0.000366	0.000404
MPI_SendRecv (datasize = 200)	0.000407	0.000231	0.000185	0.000133	0.000132	0.000111	0.000136	0.000134	0.00016	0.000149	0.000175
MPI+OpenMP (datasize = 200)	0.018666	0.020348	0.018105	0.018747	0.028993	0.039811	0.050493	0.062222	0.072864	0.083973	0.095177
OpenMP (datasize = 1000)	0.001441	0.000785	0.00043	0.000223	0.000168	0.000172	0.000163	0.000188	0.000178	0.000184	0.000202
Pthread (datasize = 1000)	0.001796	0.001842	0.001339	0.00102	0.000683	0.000678	0.000663	0.00076	0.000961	0.00102	0.001181
MPI_Broadcast (datasize = 1000)	0.003749	0.003526	0.004083	0.003691	0.00459	0.005331	0.006034	0.00825	0.007852	0.008263	0.00824
MPI_SendRecv (datasize = 1000)	0.008388	0.004489	0.002652	0.001891	0.001642	0.001495	0.001607	0.001671	0.001754	0.001866	0.001854
MPI+OpenMP (datasize = 1000)	0.100816	0.088826	0.089605	0.089605	0.096955	0.101126	0.104383	0.105814	0.107687	0.108353	0.110477
OpenMP (datasize = 5000)	0.038135	0.020009	0.011193	0.007349	0.007865	0.00689	0.007813	0.007324	0.006791	0.007182	0.007455
Pthread (datasize = 5000)	0.049548	0.030621	0.020651	0.014399	0.011387	0.009996	0.010678	0.010247	0.010653	0.010566	0.010409
MPI_Broadcast (datasize = 5000)	0.115612	0.10955	0.136592	0.149088	0.151832	0.152886	0.175674	0.267236	0.213268	0.290198	0.264689
MPI_SendRecv (datasize = 5000)	0.236042	0.123684	0.06762	0.057154	0.059181	0.062048	0.064452	0.063438	0.06498	0.066163	0.064441
MPI+OpenMP (datasize = 5000)	0.541106	0.54179	0.526519	0.515006	0.56695	0.628971	0.634781	0.646498	0.679211	0.672768	0.698287

Figure 12: Raw data - Average computation time on different configurations

3.4 Data analysis

The following figure shows the speedup (sequential execution time (parallel version using 1 processor) / parallel execution time) under different configurations (datasize and number of computational cores/threads).



Number of computational processors/threads	1	2	4	8	12	16	20	24	28	32	36
OpenMP (datasize = 200)	1	1.44	2.25	2.666667	2.181818	2.181818	2.4	1.714286	1.043478	1.565217	0.642857
Pthread (datasize = 200)	1	1	0.904762	0.44358	0.32948	0.24622	0.194872	0.168889	0.142145	0.126526	0.105556
MPI_Broadcast (datasize = 200)	1	1.108974	0.910526	0.869347	0.786364	0.786364	0.6337	0.565359	0.580537	0.472678	0.428218
MPI_SendRecv (datasize = 200)	1	1.761905	2.2	3.06015	3.083333	3.666667	2.992647	3.037313	2.54375	2.731544	2.325714
MPI+OpenMP (datasize = 200)	1	0.917338	1.030986	0.995679	0.643811	0.468865	0.369675	0.29999	0.256176	0.222286	0.196119
OpenMP (datasize = 1000)	1	1.835669	3.351163	6.461883	8.577381	8.377907	8.840491	7.664894	8.095506	7.831522	7.133663
Pthread (datasize = 1000)	1	0.975027	1.341299	1.760784	2.629575	2.648968	2.708899	2.363158	1.868887	1.760784	1.520745
MPI_Broadcast (datasize = 1000)	1	1.063244	0.918197	1.015714	0.816776	0.703245	0.621313	0.454424	0.477458	0.453709	0.454976
MPI_SendRecv (datasize = 1000)	1	1.868568	3.162896	4.435748	5.108404	5.610702	5.219664	5.019749	4.782212	4.495177	4.524272
MPI+OpenMP (datasize = 1000)	1	1.134983	1.125116	1.125116	1.039823	0.996935	0.965828	0.952766	0.936195	0.93044	0.912552
OpenMP (datasize = 5000)	1	1.905892	3.40704	5.189141	4.848697	5.534833	4.880968	5.206854	5.615521	5.309802	5.115359
Pthread (datasize = 5000)	1	1.618105	2.399303	3.441072	4.351278	4.956783	4.640195	4.835366	4.651084	4.689381	4.760111
MPI_Broadcast (datasize = 5000)	1	1.055335	0.846404	0.775461	0.761447	0.756197	0.658105	0.432621	0.542097	0.39839	0.436784
MPI_SendRecv (datasize = 5000)	1	1.908428	3.490713	4.12993	3.988476	3.804184	3.662291	3.72083	3.632533	3.567583	3.662916
MPI+OpenMP (datasize = 5000)	1	0.998738	1.027705	1.050679	0.954416	0.860304	0.852429	0.83698	0.796668	0.804298	0.774905

Figure 13: Speedup on different configurations

Sequential vs openMP vs Pthread vs MPI Broadcast vs MPI Send-Recv vs MPI+OpenMP vs CUDA: When the data size is small, the overhead produced by parallel programming is large, and as the data size increases, the proportion of overhead of multi-threading decreases, but the overhead of multi-processing still accounts for a large proportion. This is because in the case of multi-threading, the memory is shared, and the only overhead is the workitem allocation. In contrast, in the case of multi-processing, the overhead is not only the workitem allocation, but also the data copy between processes, and the latter is proportional to the data size.

In CUDA logic, each body corresponds to a thread, but in fact there are a total of 32 physical threads in each block. But even so, the number of threads is large, the degree of parallelism is high, and the speed is unsurpassed.

MPI+OpenMP version is abnormally slow, in this logical framework, whether OpenMP runs alone or MPI runs alone, the speed is acceptable, but once put them together the program runs particularly slow. This combination logic is the same as my last assignment, and the result of the last assignment is normal. I've tried to control the variables as much as I can, but I still can't figure out why.

openMP vs Pthread: In terms of the overall trend, even though both OpenMP version and Pthread version are multi-threading programs, the overhead of the latter is larger than the former. This is because the dynamic thread pool implementation method adopted by Pthread version is: multiple threads compete for the read and write rights of the *progress* variable, and after the competition, use *progress* to read the work item from the work item list, and then increment *progress*. This implementation allows threads to automatically compete for the next work item after completing a work item until all work items have been computed. This avoids frequent communication between threads and the main process. But the procedure of competition uses a mutex lock. When a thread is acquiring a work item, other threads that complete the work item will be blocked. However, OpenMP version does not have this blocking problem, and the overhead will be smaller than Pthread version.

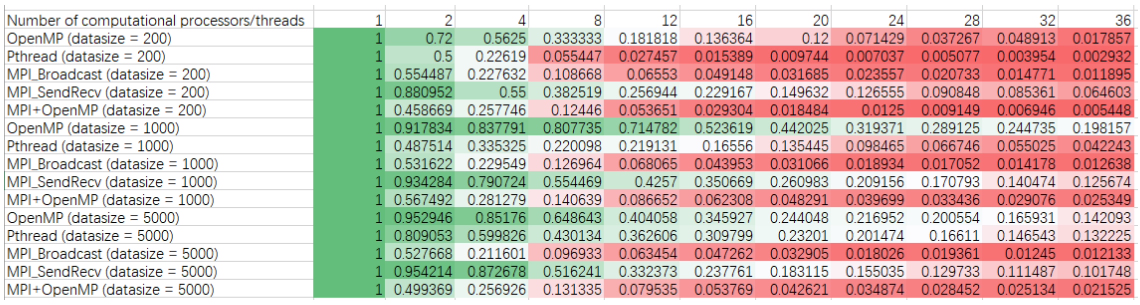
MPI Broadcast vs MPI Send-Recv: In both versions, the overhead of workitems distribution and the result sending back are the same, the only difference is the old data copy. When the number of processes is small, the overhead of MPI Broadcast version is smaller than that of the MPI Send-Recv version because the former's main process only needs to transfer data to the child processes once per iteration. When the number of processes is large, the MPI Broadcast version has to transfer a complete data to each additional child process, thus the overhead is rising rapidly, while the total amount of data the MPI Send-Recv version's main process transfers to its child processes is unchanged, and the overhead is relatively small. Taken together, because the amount of computation for each data point is not heavy, and the degree of acceleration is limited, the overhead of the data copy is obvious, the more processes the MPI Broadcast version usees, the slower. In contrast, the MPI Send-Recv version can show the acceleration effect as the number of used processes increases when the data size is large.

The number of threads/processes vs. Execution time: It can be seen that when datasize is large enough, execution time decreases as the number of computational threads/processes being used increases, and this effect is apparent when there are only a few threads/processes being used. It shows the diminishing marginal utility. This is because when the number of threads/processes being used increases, the data scale each thread/process processes decreases, and the extent of reduction also decreases. The total overhead of message communication between threads/processes and the overhead of threads creation increases, which further reduces the marginal benefit and even makes it negative.

Datasize vs. Execution time and Speedup It can be seen that when the datasize is large, the reduction in execution time is significant. However when the datasize is small, the reduction in parallel versions' execution time is trivial, and the speedup is also unstable and finally decreasing when the number of cores increases. And as the data size becomes larger, the speedup ratio of multi-threading programs is significantly higher than that of multi-process programs.

This is because in multi-threaded versions' situation, the overhead of thread creations and message passing is fixed with the same number of threads, since the data is stored in the shared memory of threads. Therefore, when the datasize is small, the benefit can hardly cover the overhead if the number of thread is large, while when the datasize is large, the net benefit is more apparent.

In contrast, in multi-process version's situation, the overhead of message passing is proportional to data size with the same number of processes, since the data need to be passed to independent memory of each process. Therefore, the net benefit will not be grately influenced by data size.



Number of computational processors/threads	1	2	4	8	12	16	20	24	28	32	36
OpenMP (datasize = 200)	1	0.72	0.5625	0.333333	0.181818	0.136364	0.12	0.071429	0.037267	0.048913	0.017857
Pthread (datasize = 200)	1	0.5	0.22619	0.055447	0.027457	0.015389	0.009744	0.007037	0.005077	0.003954	0.002932
MPI_Broadcast (datasize = 200)	1	0.554487	0.227632	0.108668	0.06553	0.049148	0.031685	0.023557	0.020733	0.014771	0.011895
MPI_SendRecv (datasize = 200)	1	0.880952	0.55	0.382519	0.256944	0.229167	0.149632	0.126555	0.090848	0.085361	0.064603
MPI+OpenMP (datasize = 200)	1	0.458669	0.257746	0.12446	0.053651	0.029304	0.018484	0.0125	0.009149	0.006946	0.005448
OpenMP (datasize = 1000)	1	0.917834	0.837791	0.807735	0.714782	0.523619	0.442025	0.319371	0.289125	0.244735	0.198157
Pthread (datasize = 1000)	1	0.487514	0.335325	0.220098	0.219131	0.16556	0.135445	0.098465	0.066746	0.055025	0.042243
MPI_Broadcast (datasize = 1000)	1	0.531622	0.229549	0.126964	0.068065	0.043953	0.031066	0.018934	0.017052	0.014178	0.012638
MPI_SendRecv (datasize = 1000)	1	0.934284	0.790724	0.554469	0.4257	0.350669	0.260983	0.209156	0.170793	0.140474	0.125674
MPI+OpenMP (datasize = 1000)	1	0.567492	0.281279	0.140639	0.086652	0.062308	0.048291	0.039699	0.033436	0.029076	0.025349
OpenMP (datasize = 5000)	1	0.952946	0.85176	0.648643	0.404058	0.345927	0.244048	0.216952	0.200554	0.165931	0.142093
Pthread (datasize = 5000)	1	0.809053	0.599826	0.430134	0.362606	0.309799	0.23201	0.201474	0.16611	0.146543	0.132225
MPI_Broadcast (datasize = 5000)	1	0.527668	0.211601	0.096933	0.063454	0.047262	0.032905	0.018026	0.019361	0.01245	0.012133
MPI_SendRecv (datasize = 5000)	1	0.954214	0.872678	0.516241	0.332373	0.237761	0.183115	0.155035	0.129733	0.111487	0.101748
MPI+OpenMP (datasize = 5000)	1	0.499369	0.256926	0.131335	0.079535	0.053769	0.042621	0.034874	0.028452	0.025134	0.021525

Figure 14: Cost-efficiency on different configurations

The above figure shows the Cost-efficiency (speedup / number of processes/threads being used) under different number of cores/threads.

The number of cores vs. Cost-efficiency It can be seen that when datasize is large, the Cost-efficiency keeps decreasing as the number of processes/threads being used increases.

4 Conclusion

Because the amount of computation at each point is not large, and some points do not need to be calculated, this project is more suitable for using multi-threading with memory sharing (OpenMP, Pthread, CUDA).

The overhead of data copy in multi-processing(MPI) is too large compared to the degree of parallel acceleration, and even if more processes are used, the final speed is still slow than the sequential version.