CSC4005-Assignment-4 Report

Name: HuangPengxiang

Student ID: 119010108

Objective: This Assignment requires to four parralle version of heat distribution program using MPI, Pthread, CUDA, and OpenMP. The bonus part is use MPI with OpenMP to implement parallel computation. Also, The program should be tested in the cluster in order to get the experimental data. The experiment analysis session and conclusion session which concludes the comparison result is necessary to be included in this report as well.

Introduction

- In physics, the heat-distribution problem is the problem generally based on natural heat transfer phenomena such as conduction, convection, and irradiation. A homogeneous distribution of heat gains is first of all determined by spreading both solar energy access and heat storage within the building form. In this Project, there is a fixed-size room and has four walls and fire souce place. The initial temperature of the wall is 36 degree, and the temperature of the heat source is 100 degree. The color of each grid in room size represent the temperature. The temperature is higher if the color is darker. This Project also use Jacobi and Sor algorithm to iterate the temperature for each grid until it is stable.
- However, If we apply sequential method to calculate it, the compuation time will be quite large with the size of

room increase. Theoretically, Sequential computation will calculate grid temperature used by the Jacobi or Sor algorithm if we pick a fixed body number for a specific image size. Therefore, This program aims to investigate different version of Parrallel computing to optimize the compuatation speed of Heat-Distribution calculation. In this experiment, I designed **Five parrallel computing version including MPI , Pthread, CUDA, OpenMP, MPI+OpenMP** and compare those version with each other to find out the best one in computing performance for different scenario.

- The basic problems and tasks are: given a fixed of room size and source temperature, and the program should perform the parrallel computing strictly following the rule of heat distribution law calculation for whatever the grid located and number of cores/threads. During the process, the data race of `stablized` variable should also be considered to get the correct result. Running the parrallel computation program in the cluster to get the experimental data and analyze the performance for specific situation such as different cores/thread and different input size.
- The rest of report is mainly containing Four part: **Running Guidance , The Design Approach, Performance Analysis , and Conclusion**. The Running Guidance shows you The files i include to finish the project and how to run it on the server. It Also includes a script file to show you how I get my experimental data on the server. The Design Approach mainly tells How I design my five version of parralle computation for specific motivation and the methods to implement those code. The Performance Analysis is the comparision of performance between those approach implementing the heat distribution computation. I set different variable for those program to observe in what scenario which one will perform better. The Conclusion part is focused on the result of analysis, and give the table of analysis results between those heat distribution computing program.

Running Guidance

My program inlcude the file **MPI_Version, Pthread_Version, CUDA_Version, OpenMP_Version, MPI+OpenMP_Version**, and **Appendix** which contains the experimental data and some analysis pictures.

There are two major parts for you to run my program: `build` and `test`.

To build those five versions of parallel computation, basically you should follow the instruction below:

For MPI, Pthread, OpenMP, MPI+OpenMP:

```
$ cd /* The location of one of these version */
$ mkdir build && cd build
$ cmake .. -DCMAKE_BUILD_TYPE=Debug
$ cmake --build . -j4
```

For CUDA:

```
$ mkdir build && cd build
$ source scl_source enable devtoolset-10
$ CC=gcc CXX=g++ cmake ..
$ make -j12
```

To test my program and see the graphic ouput, you need to follow the instructions below:

For MPI Version

if you want to test my program in your own computer, you may need to

```
mpirun -np num_of_cores ./csc_4005_imgui
#such like mpirun -np 3 ./csc_4005_imgui
```

if you want to test my program in cluster, I also designed a running script which located in /Project4/Appendix/ , you can found a script named mpi_4.sh and use it to test my program. You need to use `sbatch` command to submit my program to cluster, and use `xvfb-run` to see the terminal output. However, in this way, you can not see my graphic output. Since salloc command somehow didn't work on cluster for my computer, I only use sbatch to check the terminal output.

```
# The Way I Get My Experimental Data
$ cd /* where mpi_4 located */
$ sbatch mpi_4.sh
$ cat slurm.out
# You may need to type control + c to end the program since it is the dead loop
```

For Pthread Version && OpenMP Version

You may need to add one more argument to test my program, I ask user to input the number of thread before running it. You could aslo use `xvfb-run` to close the graphic output.

```
./csc_4005_imgui 16 # means the number of thread in my progam will be 16
```

For CUDA Version

You also need to add one more argument to test my program, I ask user to input the number of thread before running it. You could aslo use xvfb-run to close the graphic output.

```
# you have to salloc for CUDA, or you may unable to run my program.
$ salloc -n1 -t10
$ srun /csc4005_imgui 16 # means the number of thread in program will be 16
```

For MPI+OpenMP Version

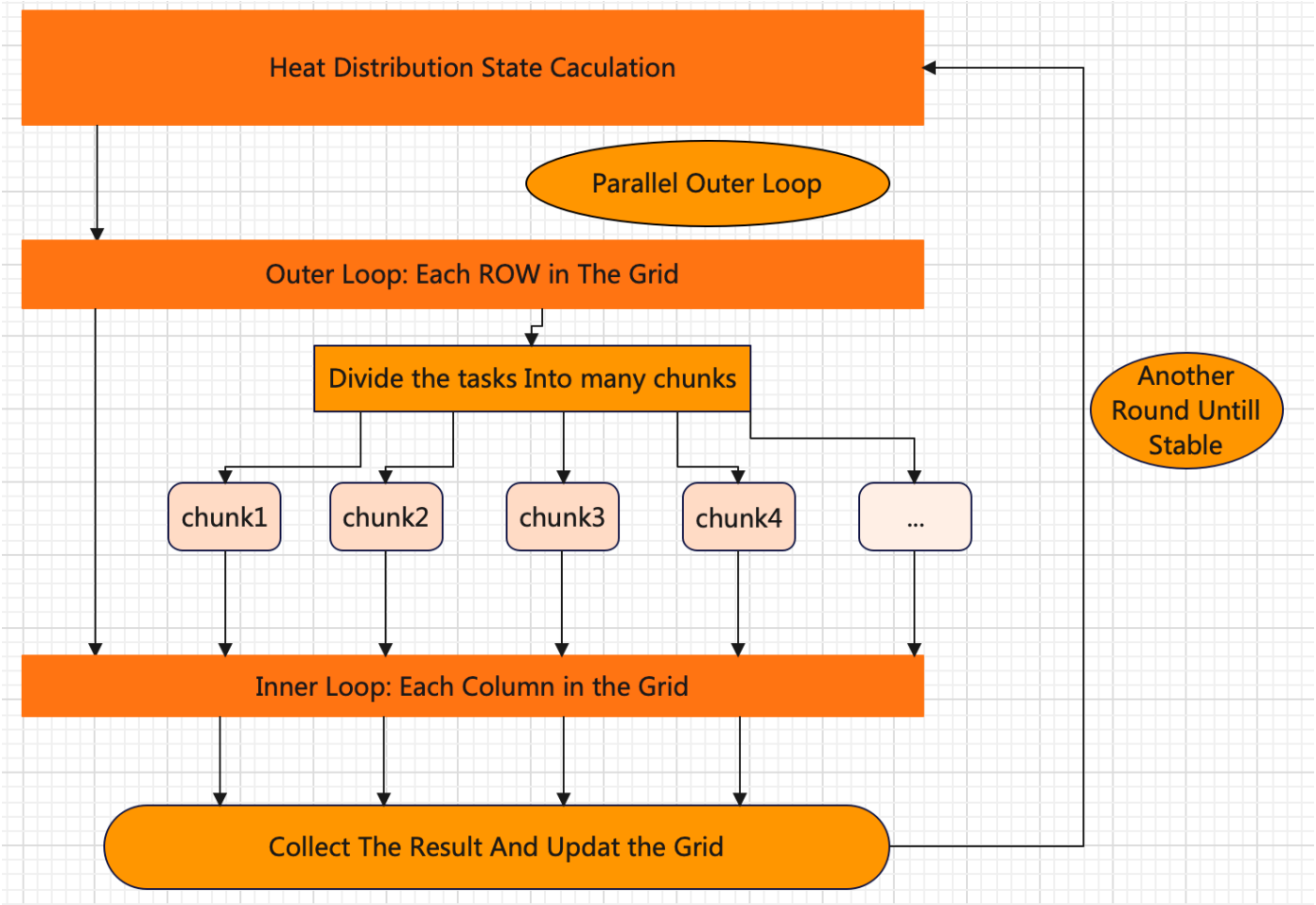
You should add one more argument after normal `mpirun` . I ask user to input the number of thread and number of cores in the program, you may also use xvfb-run to close the graphic output.

```
$ mpirun -np num_of_cores ./csc_4005_imgui num_of_thread_each_core
# such like mpirun -np 3 ./csc_4005_imgui 8 means there are 3 cores totally and each core will
use 8 thread to parallel computation
```

Design Approach

Parallel Computing Design

The main task is to calculate each grid's temperature used by Jacobi and Sor algorithm and update the whole room state until it reach stable. If we assume there are $N \times N$ grid in the fixed room size, and we need two `for` loop to calculate all the grid for each round. It means that we have **two for loop** for the whole sequential computation. In this Project, I choose the **outer loop** to parallel since it has less sheduling overhead than inner loop. And Since **it is embarrassingly parallel computation strategy** (**data independent** from each other), theoratically, It will be a **cost optimal** model, which the performance will propotional change with the problem size change.



Code Design

From the previous fogure, each version design will strictly follow the task distribution like this: divide each row tasks and each processor/thread calculate its own column and gather all the result together in the main thread or main core to update and draw the final figure. And We choose outerloop mainly because the theoretically the inner loop will have more shecduling overhead than outerloop, since every time jump to the outerloop it need to reallocte the same resource for each rank/thread. , it may slow down all the compuation and won't improve the parallel efficiency compared with outer one.

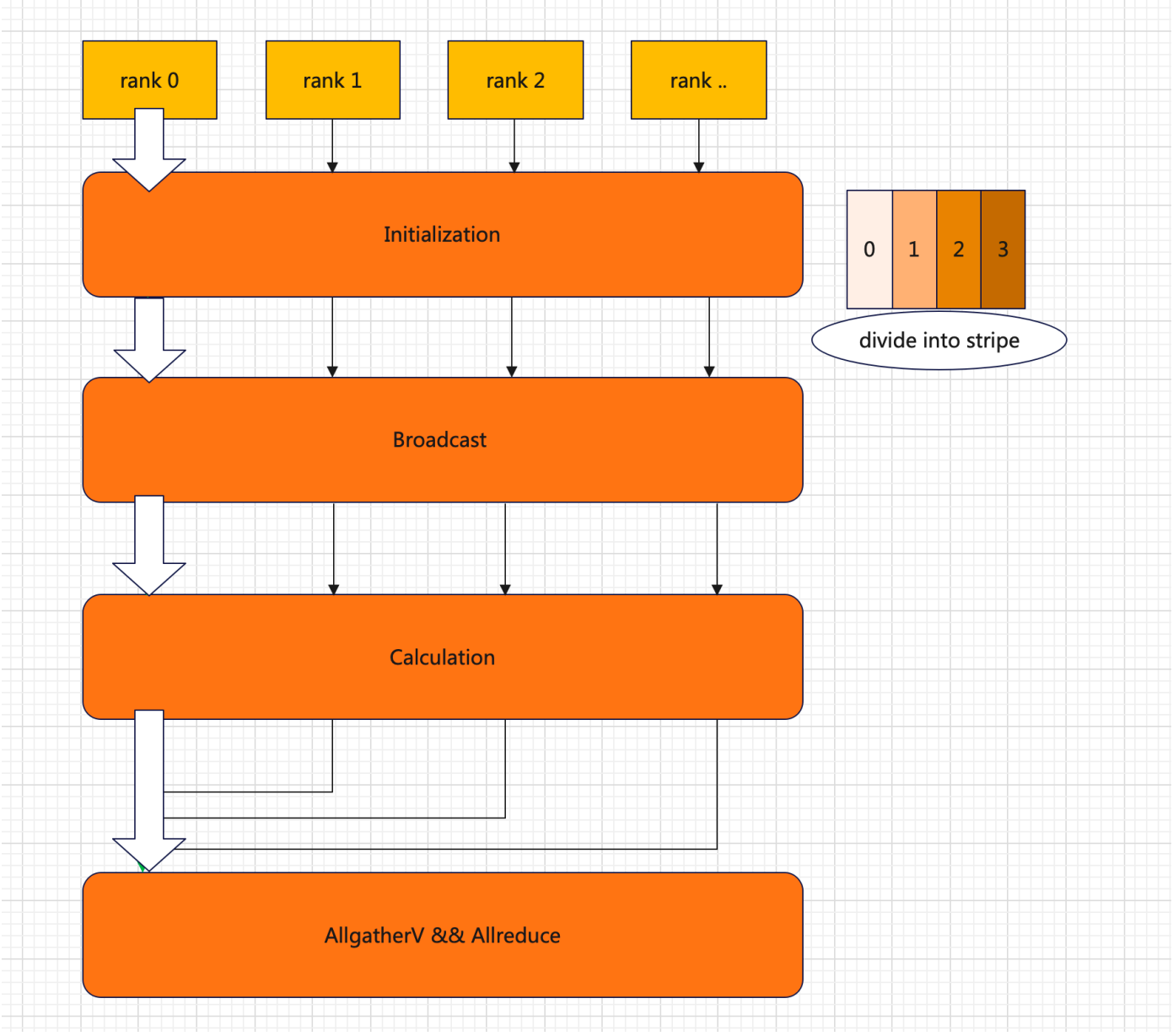
Also, it have the chance to occur the **data race** on the bool variable `stabilize`. The data race will occur when two different thread manipulate the same variable. Basically, two thread have the access the manipulate their shared memory, but when the manipulate it at the same time, they need to read and write the resource at the same place, which will occur data race and compitition. It will slow down the paralle efficiency if we add `critial` section to avoid the data race. So my solution is malloc different resouce for those thread, and each thread can access its own stable variable. And after the calculation is finished, we gather those variable used by `&` operation. It will have the same

result compared with the sequential one and also can avoid the data race. At the same time, the data race varaible is only one, so we can apply this method, which won't cose too much memory space.

MPI Design

To design parallel computation via MPI, it mainly compose of 5 parts:

- **Creation and Partition:** create the state and grid for each rank, and partition the problem using **stripe partition**
- **Bcast:** broadcast the state and grid information to each rank make sure they are the same
- **Calculation:** calculate the grid using two algorithm for each rank
- **Allgather && Allreduce:** gather all the result into rank 0, and reduce the stable result into rank 0.
- **Update:** update the room state and check its stablized



-- Bcast:

The information I bcast to make sure each rank have the same state and grid to access for each same roud: All variable in the state struct and all the temperature variable in the grid.

```
/* board cast the information to calcualte */
MPI_Bcast(&current_state.border_temp, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Bcast(&current_state.source_temp, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Bcast(&current_state.block_size, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Bcast(&current_state.tolerance, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);
MPI_Bcast(&current_state.sor_constant, 1, MPI_FLOAT, 0, MPI_COMM_WORLD);

MPI_Bcast(&current_state.room_size, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&current_state.source_x, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&current_state.source_y, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&current_state.algo, 1, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(&grid.get_current_buffer()[0], size, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

-- AllGather:

I allgather the grid temperature information in the `hdist.hpp`. And there is a little different between the algorithm Jacobi and Sor. since Sor has add one more argument `for k in {0,1}`, which means Sor algorithm will change a buffer when it finish one iteration. So we may need to Bcast again when we change a buffer. However, algorithm Jacobi only change the buffer when it finish the calculation. Hence, we don't need to Bcast the buffer when we finish the calculation.

```
/* for the Jacobi */
MPI_Allgatherv(&grid.get_current_buffer()[displ[rank]], recvcounts[rank] , MPI_DOUBLE,
&grid.get_current_buffer()[0],recvcounts,displ,MPI_DOUBLE,MPI_COMM_WORLD );

/* for the Sor */
MPI_Allgatherv(&grid.get_current_buffer()[displ[rank]], recvcounts[rank] , MPI_DOUBLE,
&grid.get_current_buffer()[0],recvcounts,displ,MPI_DOUBLE,MPI_COMM_WORLD );
MPI_Bcast(&grid.get_current_buffer()[0], state.room_size * state.room_size , MPI_DOUBLE, 0,
MPI_COMM_WORLD);
```

-- Allreduce:

The variable need to reduce is the `stabilized`. Firstly malloc its own stable in each rank and let they calculate, after finish the calculation, the root will responsible to use `&` reduce all the result in each rank. And broadcast it untill the room is stable.

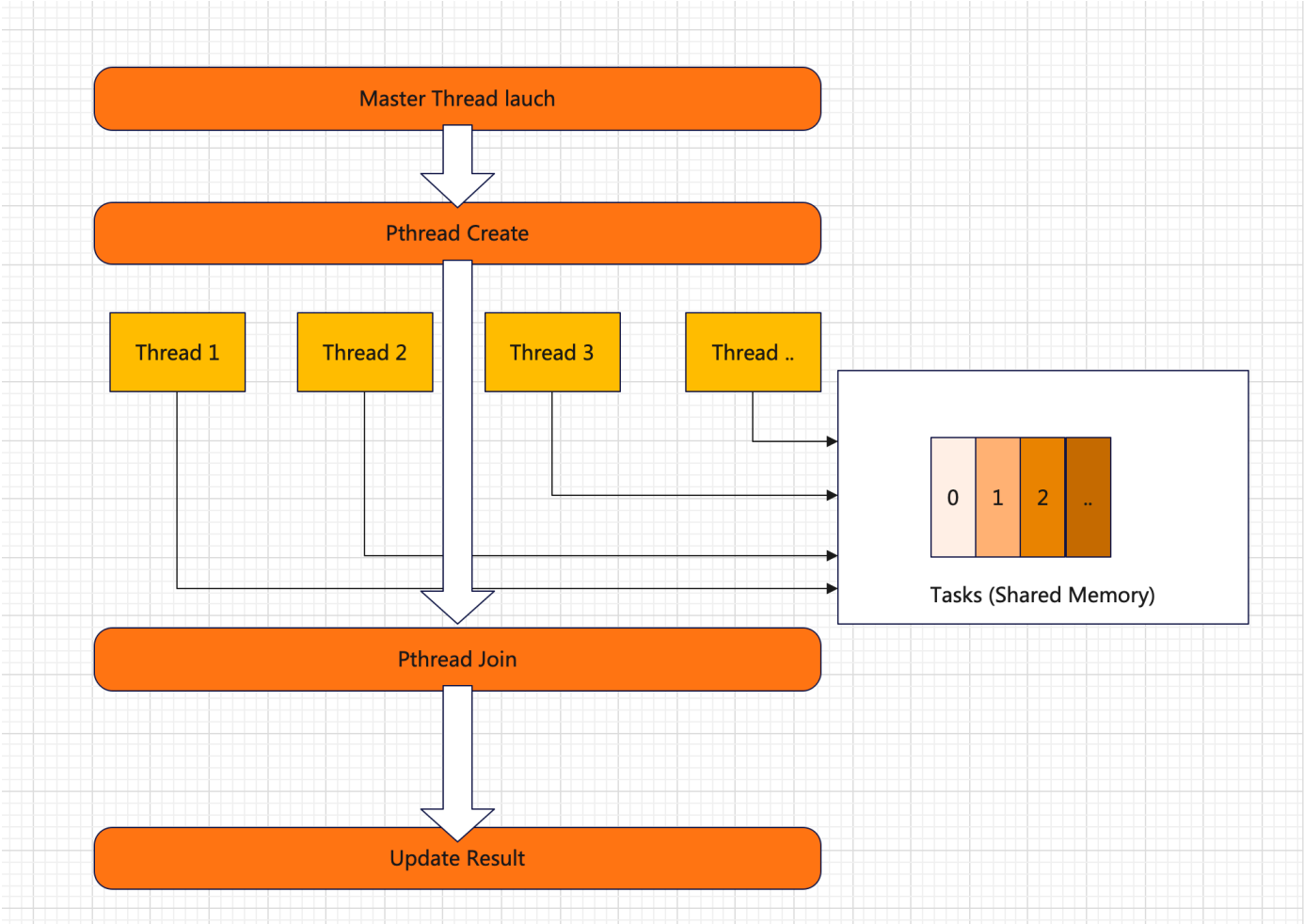
```
/* used to collect all the bool finish in every rank */
MPI_Allreduce(&root_finish, &all_finished, 1, MPI_INT, MPI_BAND, MPI_COMM_WORLD );
```

Pthread Design

For implementation via Pthread, I aslo divide the bodies into different chunks. But unlike MPI Version, since Ptrhead can share the memory bettween every thread. in this version, there is no need to boardcast the information from root to each part and send and receive the information between each part. some variables and structures would be first initialized in the shared-memory region, for example, current_states , finished , and grid . The master thread is in charge of updating these variables.

To design parallel computation via Pthread, it mainly compose of 5 parts:

- Pthread Creation
- Pthread Calculation in Shared Memory
- Pthread Join



-- Pthread Creation:

create the thread and initialization, also create the local stable varible to avoid data race.

```
/* create the threads */
for (int j = 0; j < threadnum; j++) {
  pthread_create(&threads[j],&attr,Pthread_calculate,(void*)&thread_paras[j]);
}

/* create the local stable to avoid the data race */
bool * each_stable = (bool *)  malloc(sizeof(bool) * threadnum);

for (int j = 0; j < threadnum; ++j){
  each_stable[j] = stabilized;
}
```

-- Pthread Calculation:

I create the struct for each thread to recieve the parameter and i set a pointer point to the state and the grid to implement the memory sharing. And each thread wiil do the normal calculation just as the sequential one.

```
struct threadpara *recv_para = (struct threadpara*)t;
int myid = (*recv_para).thread_id;
int threadnum = (*recv_para).thread_num;
State *state = (*recv_para).state;
Grid *grid = (*recv_para).grid;
bool *stabilized = (*recv_para).stabilized;
```

Remark that, for each iteration of updating of the elements during the computation, there is no data dependency between the elements. That's because there are two buffers involved and read and write operations are separated. After each pthreads' computation comes to end, threads would be synchronized and only the master thread would perform the grid.switch_buffer() operation.

```
/* only switch once */
if(myid == 0) (*grid).switch_buffer();
```

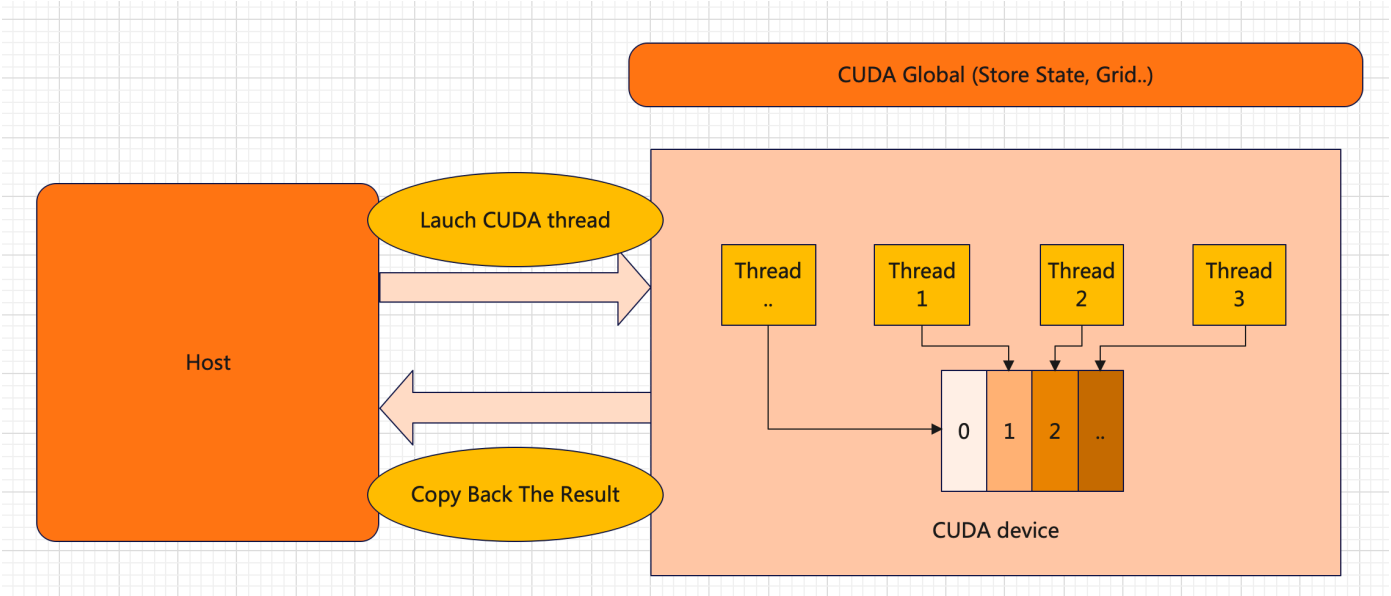
-- Pthread Join:

Join all the result into the root one and join all the stable variable into the main thread.

```
/* join the all the threads*/
for (int i = 0; i < threadnum; i++){
    pthread_join(threads[i],NULL);
}

/* join each local stable */
for (int j = 0; j < threadnum; ++j){
    stabilized &= each_stable[j];
}
```

CUDA Design

Unlike Pthread and MPI, CUDA implement the parallel computation in GPU instead of CPU. And basically, CUDA have very high speed calcualtion but CUDA has very limited fucntion to use since GPU do not support high level calculation from a hardware point of view.



To design parallel computation via CUDA, it mainly compose of 3 parts:

- Initialisation: Launch the kernel and send the variables need to calculated from host to device.
- Caculation: GPU parallel calculate the result
- Send back: send the information back and update the heat distribution room.

-- Initialisation:

I used `cudaMallocManaged` function to malloc the array to store the information like state room size, state temperature, that is mainly becuase, first the CUDA can not use std::vector directly, so I can not directly use the original function in hidist.hpp . second, `cudaMallocManaged` command will malloc the space where is accessible for both host and device. after that, I lauch the kernel function and pass the arguemnt in it to directly use.

```
/* malloc the grid and the state for cuda device to use */
int size = current_state.room_size * current_state.room_size;
cudaMallocManaged(&device_data0,sizeof(double) * size);
cudaMallocManaged(&device_data1,sizeof(double) * size);
cudaMallocManaged(&CUDA_finish, sizeof(int)    *  1  );

 /* copy the data from host to device */
for (int i = 0; i < grid.data0.size(); i++){
  device_data0[i] = grid.data0[i];
  device_data1[i] = grid.data1[i];
}
(*CUDA_finish) = 0;

/* lauch the kernel */
cudaError_t cudaStatus;
mykernel<<<1, threadnum>>>(current_state.room_size, current_state.source_x,
current_state.source_y, current_state.source_temp,
current_state.border_temp,current_state.tolerance, current_state.sor_constant, Algo ,
current_buffer, threadnum, device_data0, device_data1, CUDA_finish );
```

use `getLastError()` in cuda to return error if cuda malloc or kernel lauch falied.

```
 MallocStatus =  cudaGetLastError();
if (MallocStatus != cudaSuccess){
  fprintf(stderr, "malloc falied! : %s\n",
          cudaGetErrorString(MallocStatus));
  return 0;
}

cudaStatus = cudaGetLastError();
if (cudaStatus != cudaSuccess) {
  fprintf(stderr, "mykernel launch failed: %s\n",
          cudaGetErrorString(cudaStatus));
  return 0;
}
```

-- Calculation:

since CUDA can not use `std::vector` directly, then I rewrite the funtion for CUDA to update the grid temerature based on array .

```
* rewrite the calculation for each single grid for cuda to use */
__device__ CU_UpdateResult CU_update_single(size_t i, size_t j, int room_size, int source_x,
int source_y ,float source_temp, float border_temp, float tolerance, float sor_constant,
                                            int Algo, int current_buffer, double * data0, double
* data1){
  CU_UpdateResult result{};

  if(i == 0 || j == 0 || i == room_size -1 || j == room_size -1){
      result.temp = border_temp;
  }
  else if (i == source_x && j == source_y){
      result.temp = source_temp;
  }
  else{
      double  sum;
      if (current_buffer == 0)  sum = data0[(i+1) * room_size + j] + data0[(i-1)*room_size + j]
+ data0[i*room_size +j+1] + data0[i*room_size +j-1];
      else  sum = data1[(i+1) * room_size + j] + data1[(i-1)*room_size + j] + data1[i*room_size
+j+1] + data1[i*room_size +j-1];
      switch(Algo){
          case 0: // Jacobi
              result.temp = 0.25 * sum;
              break;
          case 1:
              if (current_buffer == 0) result.temp = data0[i*room_size + j] + (1.0 /
sor_constant) * (sum - 4.0* data0[i*room_size + j]);
              else result.temp = data1[i*room_size + j] + (1.0 / sor_constant) * (sum - 4.0*
data1[i*room_size + j]);
              break;
      }
  }
  if (current_buffer == 0 ) result.stable = fabs(data0[i*room_size + j] - result.temp) <
tolerance;
  else result.stable = fabs(data1[i*room_size + j] - result.temp) < tolerance;
  return result;
}
```

OpenMP Design

The OpenMP design is almost the same as Pthread version, since they are all shared by one core memory. The design of them are very simialr. And OpenMP is also very esay to implement since it will automatically allocate thread for you, the only thing you need is just to write

```
#pragma omp parallel num_threads(threadnum)
{
int myid = omp_get_thread_num();
int mylen = state.room_size / threadnum;
int remain = state.room_size % (threadnum);
if (myid < remain) mylen++;

size_t min;
size_t max;

if (myid < remain) min = mylen * myid;
```

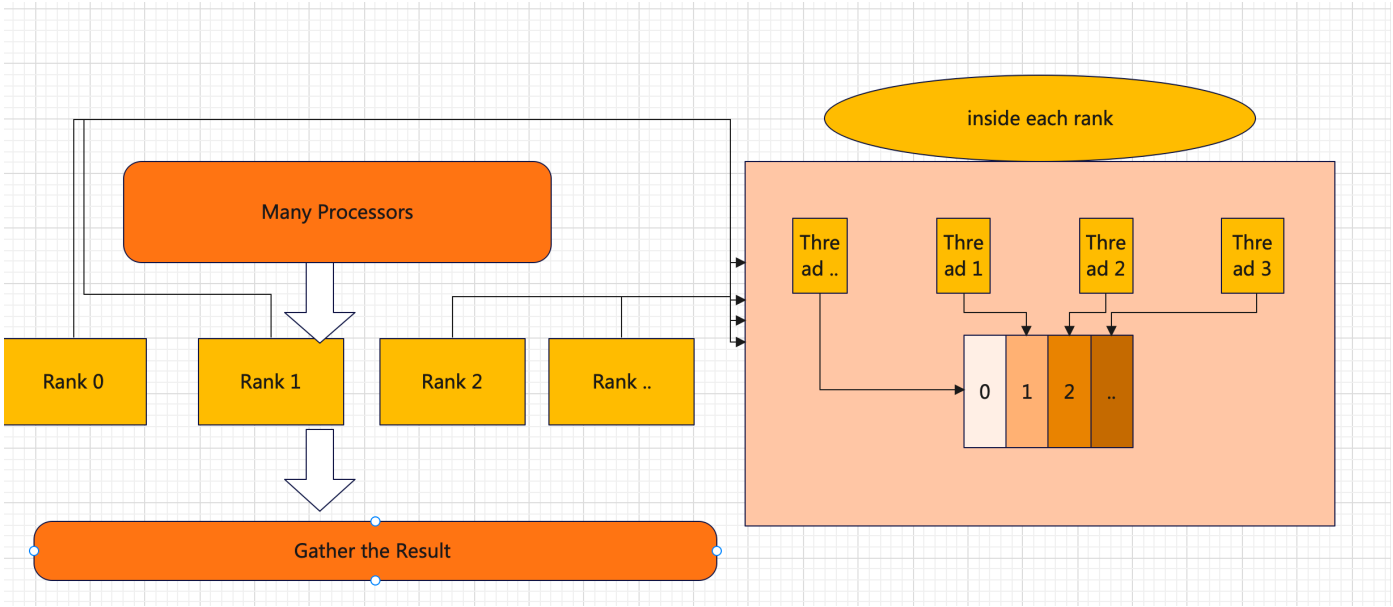
```
else min = remain * (mylen+1) + (myid - remain) * mylen;
max = min + mylen;

for (size_t i = min; i < max; ++i) {
    for (size_t j = 0; j < state.room_size; ++j) {
        auto result = update_single(i, j, grid, state);
        each_stable[myid] &= result.stable;
        grid[{alt, i, j}] = result.temp;
    }
}
#pragma omp barrier
```

How I avoid data race:

I malloc the local stable part for each thread to use, and each thread only have the access to manipulate its own `stable` variable. and after calculation, the main thread will use `&` to gather all the thread stable together and return the final result.

MPI + OpenMP Design

The bonus version combines the MPI approach with OpenMP approach. Based on the MPI program I wrote, OpenMP threads are added to further parallel the workload in different processors. For example, each processor in MPI program originally use a for loop to iterate the points in the heat distribution assigned to it. Then, combining OpenMP approach is to add parallel for clause to incrementally parallel it.



Experiment Design

Based on the implementation of the above parallel program, several experiments are designed to evaluate the performance under different configurations.

The evaluation concentrates on the following dimensions:

(1) number of cores/threads

(2) problem size

(3) Comparison between parallel versions and sequential version

(4) Comparison among different paralle versions performance and speed up

(5) comparison the speedup of different algorithms.

(6) inner loop and outer loop comparision

All MPI, Pthread, CUDA, OpenMP, and MPI + OpenMP are evaluated by the above criteria. The design detail would be explained as below:

- Design (1) - Relationship between performance and number of cores/threads
  This experiment uses the change of speedup to represent the performance change under different configurations of cores/threads. The cores/threads ranges in 1 (sequential), 2, 4, 8, 16, 32, 64, 128. And the speedup is the division of duration time compared with sequential program. The duration time we used to calculate the speedup is the mean of 20 times of iteration (in order to reduce the accidental error).
- Design (2) - Relationship between performance and problem sizes
  Unlike design (1), the variable in this experiment design is chosen to be the problem size. We fix the number of cores/ threads and test the parallel program on problem sizes equal to 100, 200, 400, 800, 1600, 3200, and 6400. The evaluation criterion is the change of speedup compared with the sequential version.
- Design (3) - Comparison between parallel versions and sequential versions
  In this experiment, the concentration changes to the comparison between parallel computing and sequential computing. We fixed the problem size and number of cores/threads, and then investigate under what configuration, parallel computing will outperform sequential computing as well as how much it outperforms.
- Design (4) - Comparison among different parallel versions: This experiment will investigate the best parallel program when the configurations are the same. In this experiment, we use the average duration of time of one iteration instead of the change of speedup to judge the performance. After analyzing the result, this experiment will give an upper level of suggestion on adopting the suitable parallel method when solving heat distribution problem under different configurations.
- Design (5) - Comparison the speedup of different algorithms. Since this program provides two algorithms to simulate the heat distribution, we try to investigate which algorithm is more suitable to be paralleled. To make it comparable, we fix the problem size and number of threads/ cores. The evaluation not only focuses on the average duration time of one iteration, but also counts the tendency of the change of performance to judge whether one algorithm has a good and stable enhancement of performance.
- Design (6) - Comparision of parallel inner loop and outer loop. we fixed the best one performance in outer loop in some fixed size and thread/core number and sam algorithm, and then we choose inner loop to parallel computing it. then we can compare the performance difference between those.

Declaration:

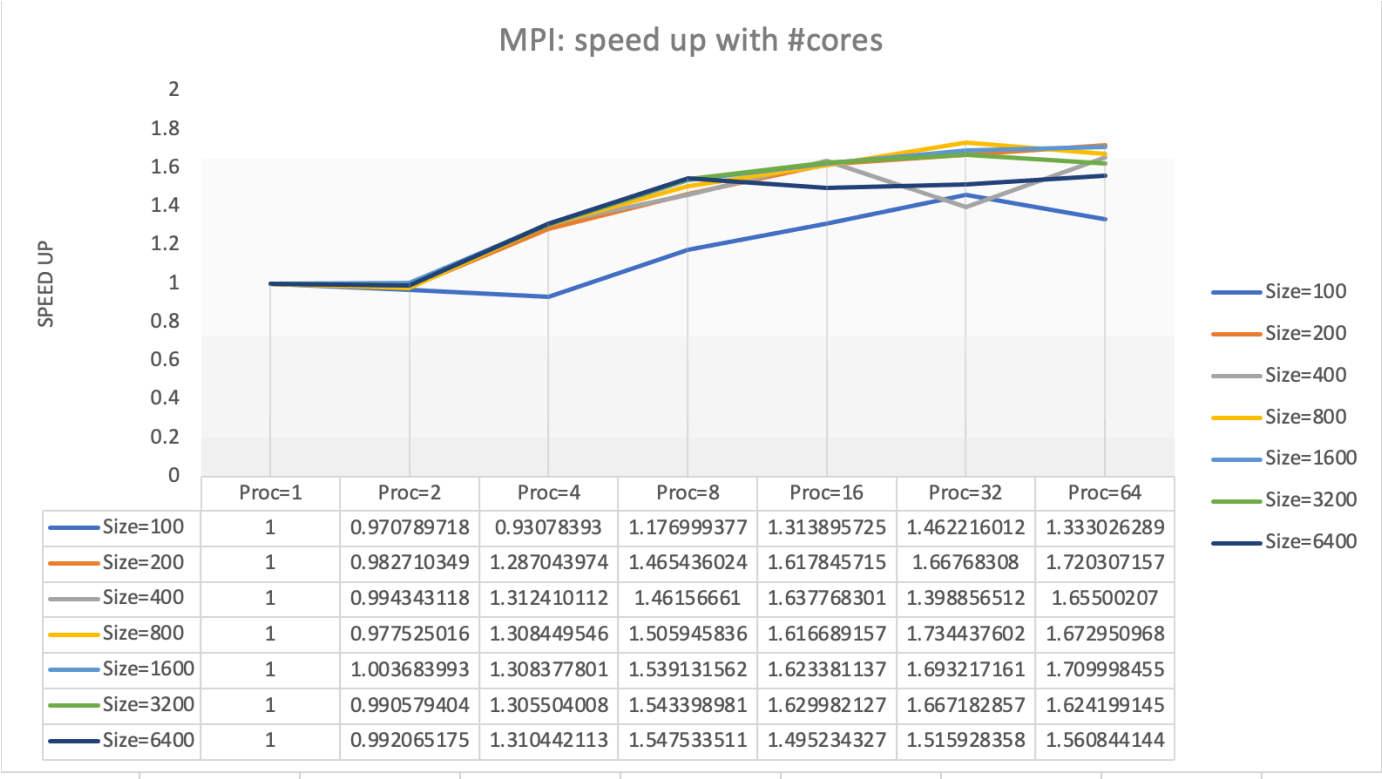
In all the experiments, I measure all the communication time and computation time each round except for drawing the graph. This is because there are many kinds of overhead (memory copy of CUDA and some collective communication) which is caused in the master threads. To make them comparable, I measure all the time except for drawing the graph.

Performance Analysis

Analysis 1: Relationship between performance and number of core/threads

-- Result 1: MPI Version:

The following figure demostrate the change of MPI speed up with number of cores increase.



	Proc=1	Proc=2	Proc=4	Proc=8	Proc=16	Proc=32	Proc=64
Size=100	1	0.970789718	0.93078393	1.176999377	1.313895725	1.462216012	1.333026289
Size=200	1	0.982710349	1.287043974	1.465436024	1.617845715	1.66768308	1.720307157
Size=400	1	0.994343118	1.312410112	1.46156661	1.637768301	1.398856512	1.65500207
Size=800	1	0.977525016	1.308449546	1.505945836	1.616689157	1.734437602	1.672950968
Size=1600	1	1.003683993	1.308377801	1.539131562	1.623381137	1.693217161	1.709998455
Size=3200	1	0.990579404	1.305504008	1.543398981	1.629982127	1.667182857	1.624199145
Size=6400	1	0.992065175	1.310442113	1.547533511	1.495234327	1.515928358	1.560844144

The results shows that, as the number of processors increases, most of case, the speedup would first increase. After a certain point (around 32 cores and 64 cores), the speedup would decrease. The tendency of performance decrease after certain point is dramatically and would give a worse performance (speedup less than 1) when the number of cores is relatively large (After 32 cores).

Also, when the size is too small, such as `size = 100`, the perfomance will instead decrease when the number of cores increase to 4.

-- Discussion 1: MPI Version:

Theoratically, The parallel computing of MPI can be devided into two parts t_{comm} and $t_{compute}$.

Then with the Amdal's law, the theoratically analysis can be the following part:

$$t = t_{comm} + t_{commpute} = P(t_{startup} + \frac{n}{p}t_{data}) + \frac{1}{p}t_{compute}$$

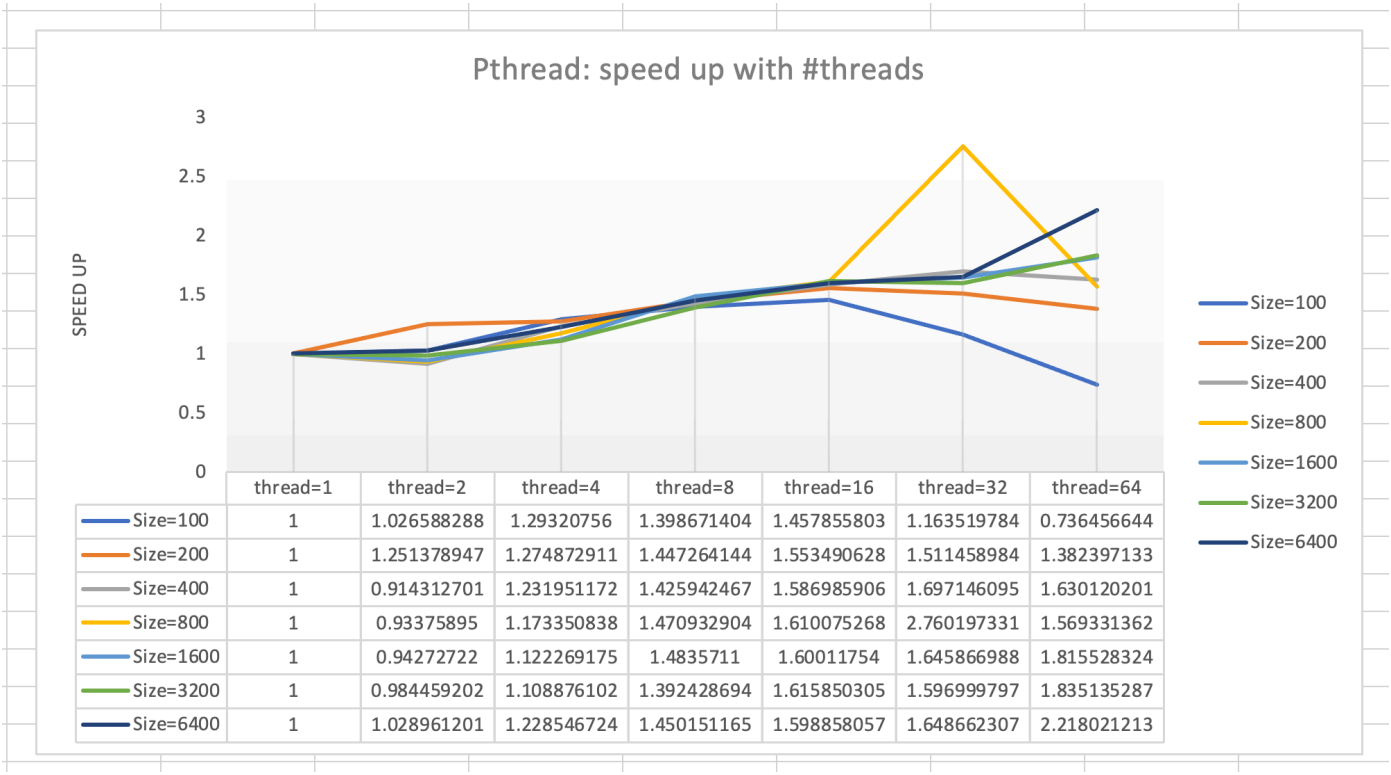
$$speedup = \frac{1}{\frac{pt_{startup}}{t_{compute}} + \frac{t_{data}}{\frac{1}{p}t_{compute}} + \frac{1}{p}}$$

Notice that, when the $t_{compute}$ is relatively small compared than startup time and commnucation time **in the small size**. the speedup would be limited and even drop when the number of processors increases. Back to the problem, In the heat distribution, the time of updating one point is small, it means that according to the formulation above, $t_{startup}/t_{joc} and t_{data}/t_{compute}$ are not small. So $p \times t_{startup}/t_{compute}$ will be gradually significant and the speedup will drop early. This phenomenon can be also explained directly by the Amdahl's law, which posits that the speedup is limited by the serial section. Meanwhile, the communication overhead would gradually replace the time-saving brought by parallel computing, so the performance would drop.

Hence, The increasing cores will not simply increase the Performance under some situdation such as small size large number of core. But most of the time, The increasing cores will result in increasing time generally in MPI Version. But it **more likely to occur some unexpected result** cause the Performance of MPI also related to the net work latency and some commnucation between node to node, which may have bigger chance to have large variance.

-- Result 2: Pthread Version:

The relationship between cores and speed up result is showing below:



	thread=1	thread=2	thread=4	thread=8	thread=16	thread=32	thread=64
Size=100	1	1.026588288	1.29320756	1.398671404	1.457855803	1.163519784	0.736456644
Size=200	1	1.251378947	1.274872911	1.447264144	1.553490628	1.511458984	1.382397133
Size=400	1	0.914312701	1.231951172	1.425942467	1.586985906	1.697146095	1.630120201
Size=800	1	0.93375895	1.173350838	1.470932904	1.610075268	2.760197331	1.569331362
Size=1600	1	0.94272722	1.122269175	1.4835711	1.60011754	1.645866988	1.815528324
Size=3200	1	0.984459202	1.108876102	1.392428694	1.615850305	1.596999797	1.835135287
Size=6400	1	1.028961201	1.228546724	1.450151165	1.598858057	1.648662307	2.218021213

From the result figure, It shows that when the problem size is not small, the performance of Pthread parallel computation will decrease first when the thread reach to 2, and then increase as the number of thread increase. when the thread number reach 16 or 32, the speed up will reach its max value. But when the thread number is over 32, the the performance will dramatically decrease.

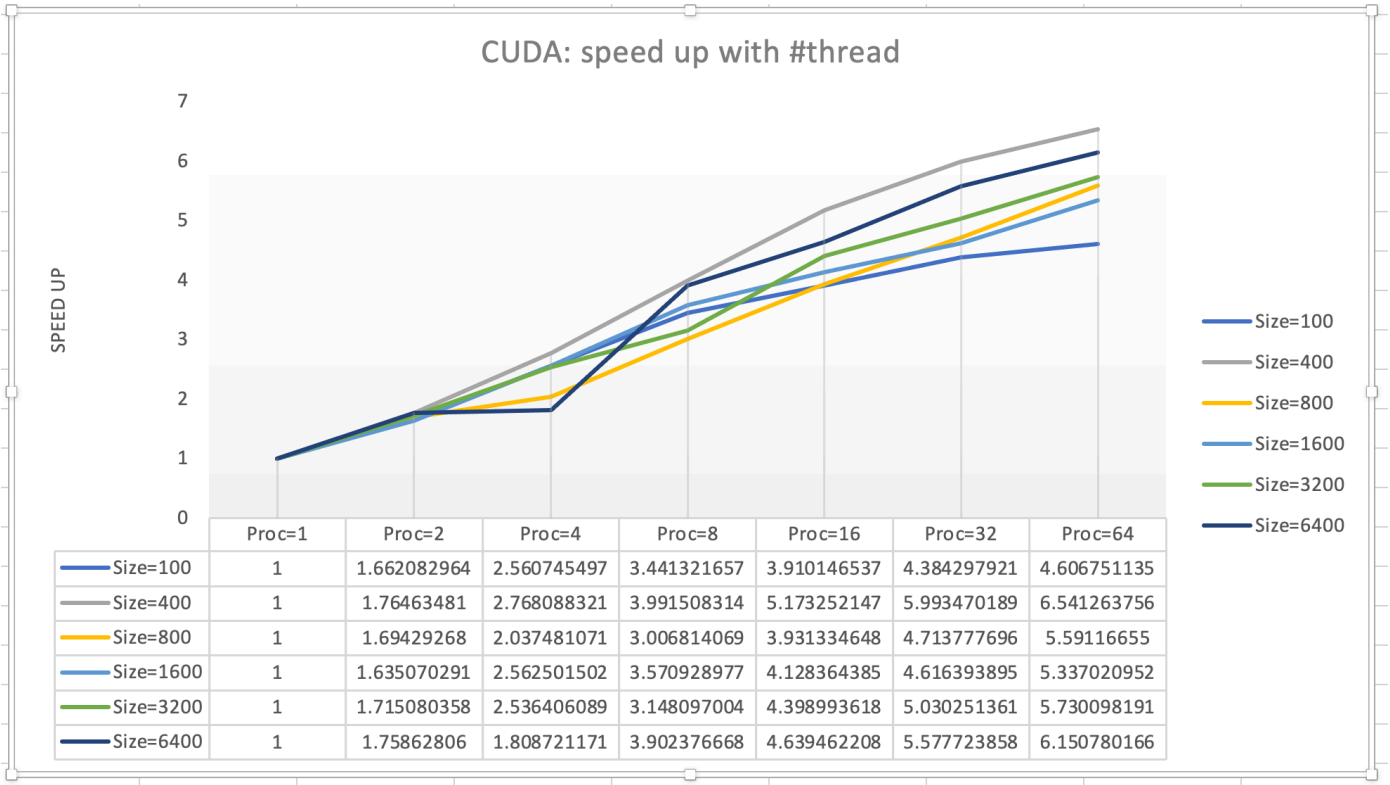
-- Discussion 2: Pthread Version:

The performance increases at the beginning when the thread number is greater than 2 can be explained by the timing saving parallelization brings. Then, the tendency's slowing down and cessation can be explained by the problem of over spawn, increasing time consumed in the threads' synchronization overhead and fork/join overhead. Especially, when the threads are over-spawn, parallelization would not improve the performance since the threads running simultaneously are limited. At the same time increasing the number of threads after that would cost extra overhead.

The conclusion basically meets the statement of Amdahl's law, limited by the serial fraction (In program, like some preparation work done by root process), the improvement of performance that increasing threads brings is bounded. It means there must be a critical point for the improvement to reach ceasation. Also compared with another problem: Odd- even sort transposition sort, whose performance may decrease as the number of processes is relative large, parallel Mandelbrot set computation's performance will cease but not decrease. This is because there are little communication between threads in this problem (all threads perform its work independently). Thus the communication overhead would be significant when the number of threads increase.

-- Result 3: CUDA Version:

The result is showing below:



	Proc=1	Proc=2	Proc=4	Proc=8	Proc=16	Proc=32	Proc=64
Size=100	1	1.662082964	2.560745497	3.441321657	3.910146537	4.384297921	4.606751135
Size=400	1	1.76463481	2.768088321	3.991508314	5.173252147	5.993470189	6.541263756
Size=800	1	1.69429268	2.037481071	3.006814069	3.931334648	4.713777696	5.59116655
Size=1600	1	1.635070291	2.562501502	3.570928977	4.128364385	4.616393895	5.337020952
Size=3200	1	1.715080358	2.536406089	3.148097004	4.398993618	5.030251361	5.730098191
Size=6400	1	1.75862806	1.808721171	3.902376668	4.639462208	5.577723858	6.150780166

From the figure, it shows that when the number of thread increases, the performance of the CUDA program will gradually slow down slightly. In general tendency, the performance increase stability. This result illustrates CUDA's great scalability, which means that CUDA will increase stably when the number of the thread increase. Otherwise like MPI and Pthread, there are some curve decrease in the middle, but CUDA increase with the stable speed.

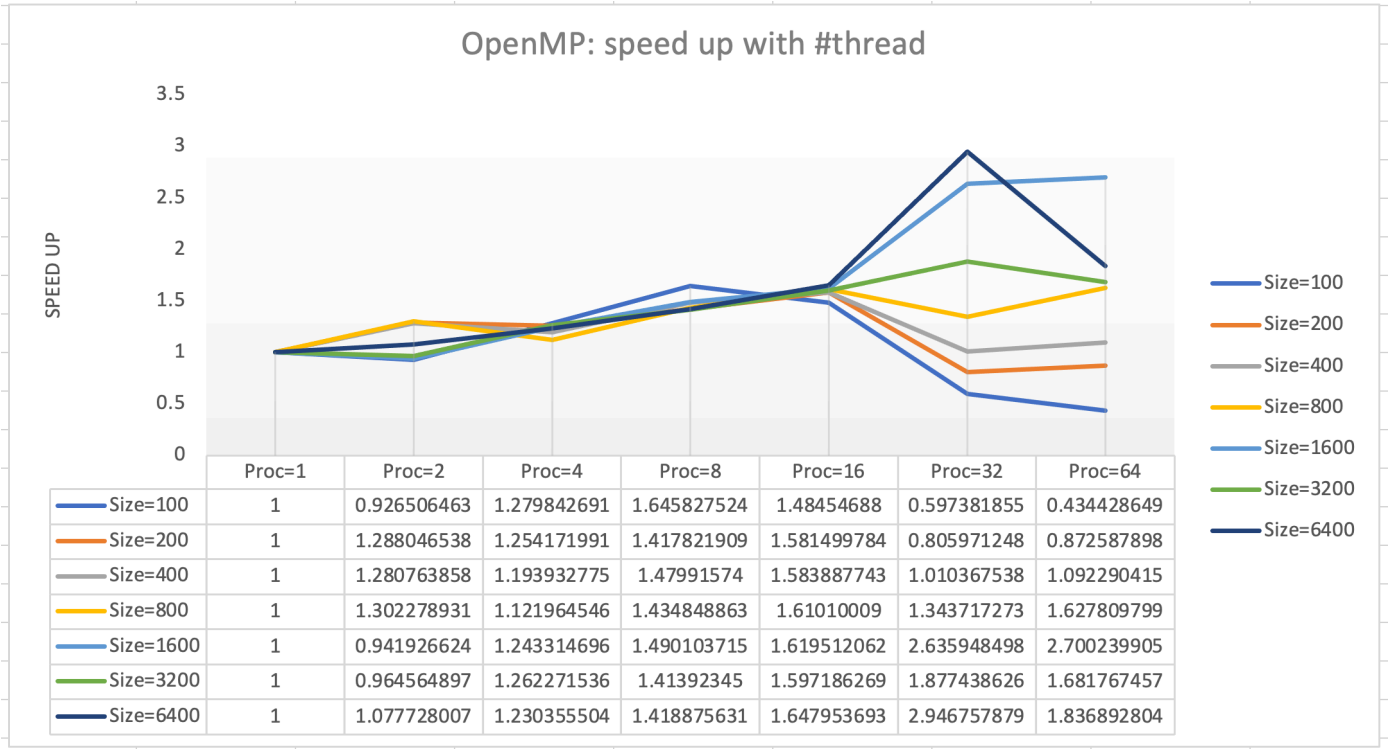
-- Discussion 3: CUDA Version:

For CUDA, The speed will consistently increase and it won't reach the threashhold. That is because CUDA use GPU to calculate, and GPU has many cores while CPU have only 32 in cluster. So when we increase the number of threads, the speed will consistently increase at this time.

The reason is that the threads in CUDA are very lightweight, which means that forking a new CUDA thread results in a very light cost.
And also, the maximum number of threads per block in GPU. The device used in this experiment is NVIDIA GeForce RTX 2080 Ti which supports maximal 1024 threads. Therefore, the increasing threads can be parallel computed in GPU and decrease the time of parallel computing section according to Amdahl's law.

-- Result 4: OpenMP Version:

The result figure is showing below:



	Proc=1	Proc=2	Proc=4	Proc=8	Proc=16	Proc=32	Proc=64
Size=100	1	0.926506463	1.279842691	1.645827524	1.48454688	0.597381855	0.434428649
Size=200	1	1.288046538	1.254171991	1.417821909	1.581499784	0.805971248	0.872587898
Size=400	1	1.280763858	1.193932775	1.47991574	1.583887743	1.010367538	1.092290415
Size=800	1	1.302278931	1.121964546	1.434848863	1.61010009	1.343717273	1.627809799
Size=1600	1	0.941926624	1.243314696	1.490103715	1.619512062	2.635948498	2.700239905
Size=3200	1	0.964564897	1.262271536	1.41392345	1.597186269	1.877438626	1.681767457
Size=6400	1	1.077728007	1.230355504	1.418875631	1.647953693	2.946757879	1.836892804

It shows that OpenMP Version will generally increase when the thread number is smaller than 64. When the thread number is reach 64, there occur a huge difference among size. when the size is small like 100, 200, 400, the performance will dramatically decrease. However, when the size is lagrge, the performance will keep increasing as the thread number increase.

-- Discusssion 4: OpenMP Version:

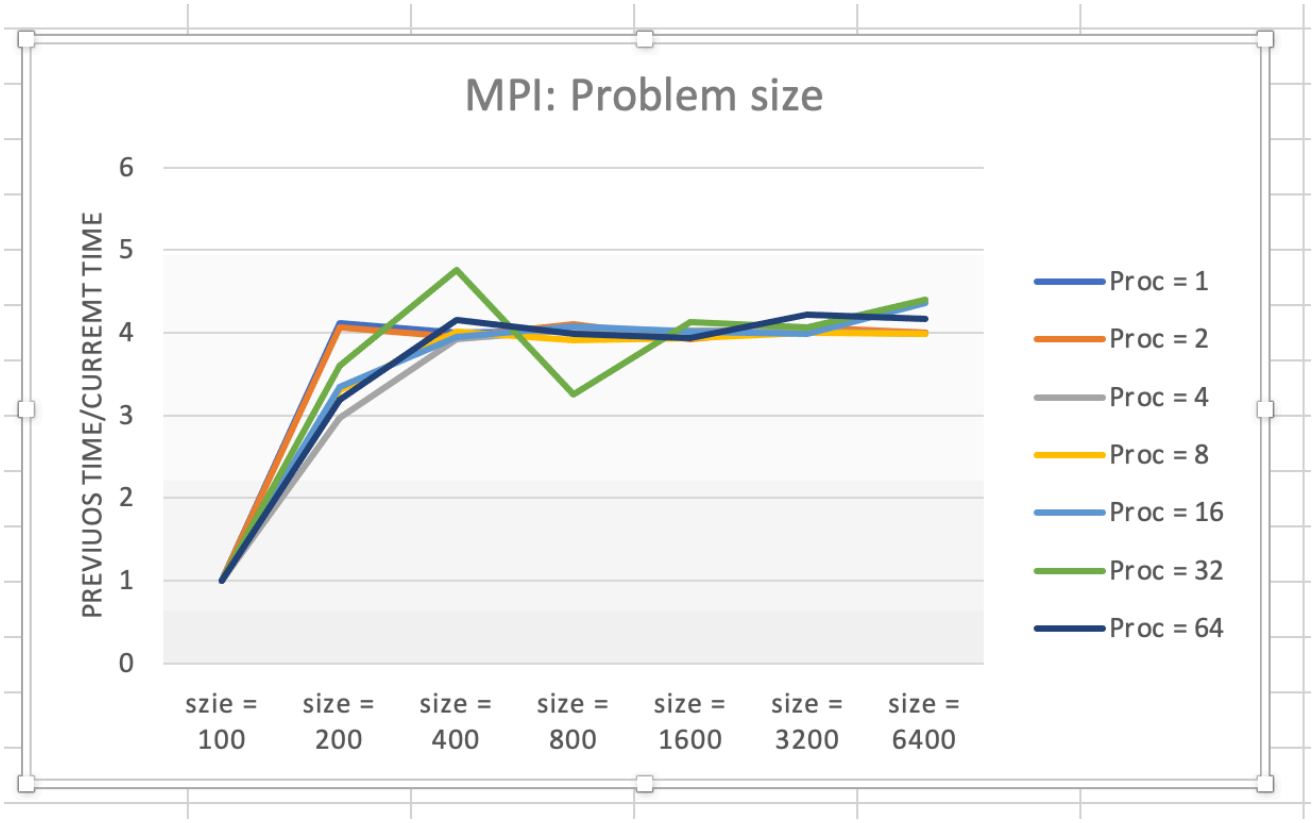
Fo OpenMP, when the number of the cores/thread increase, the speed will increase first and when they meet number of threads reach 32. The speed will decrease afrer that. that is because for each core in the cluster, it only have 32 cpu to execute the program. Which means if there are number of the threads which is greater than 32, there are only 32 threads execute program simultaneously. It is easily to occur **overspawn**. Hence, When the size is too small, the proragm may don't need those thread to calculate, but those threads were created and wait other threads to execute, which will consume a lot of time, and it may cause the parallel version performs worse than sequential one. But For the large size, The OpenMP will also increase the speed as the number of threads increase since in this case, the problem size will occupied the dominated location.

Analysis 2: Relationship between problem size and performance

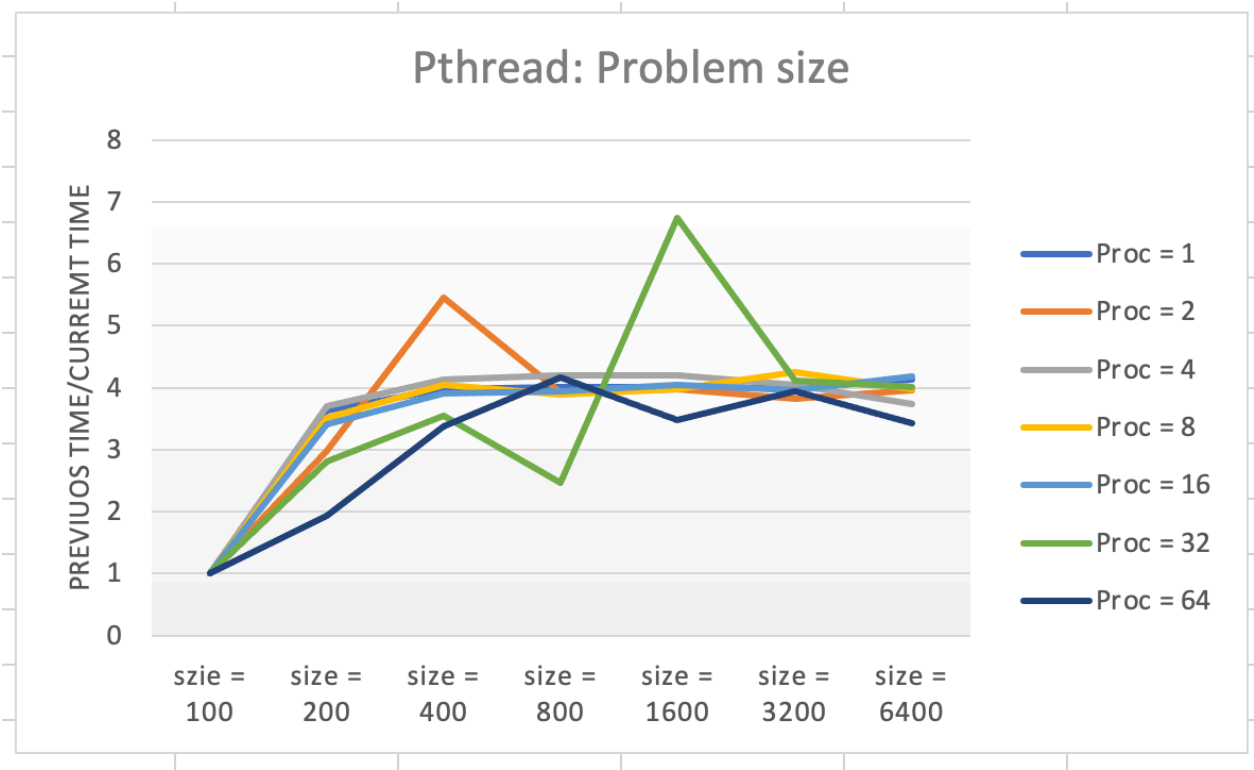
In the heat distriction problem, it should be a **cost optimal** question. **it belongs to embarassingly paralle computation**. which means that, the performance will increase in the propotional speed when the size increase. such as when the size increase 4 times, theoratically, the time will cost 4 times than previous one. In order to test that, I set the y-cordinate value to be the $t_{previous}/t_{current}$. as the size increase 4 times, theoratically the value should near 4.

-- Result:

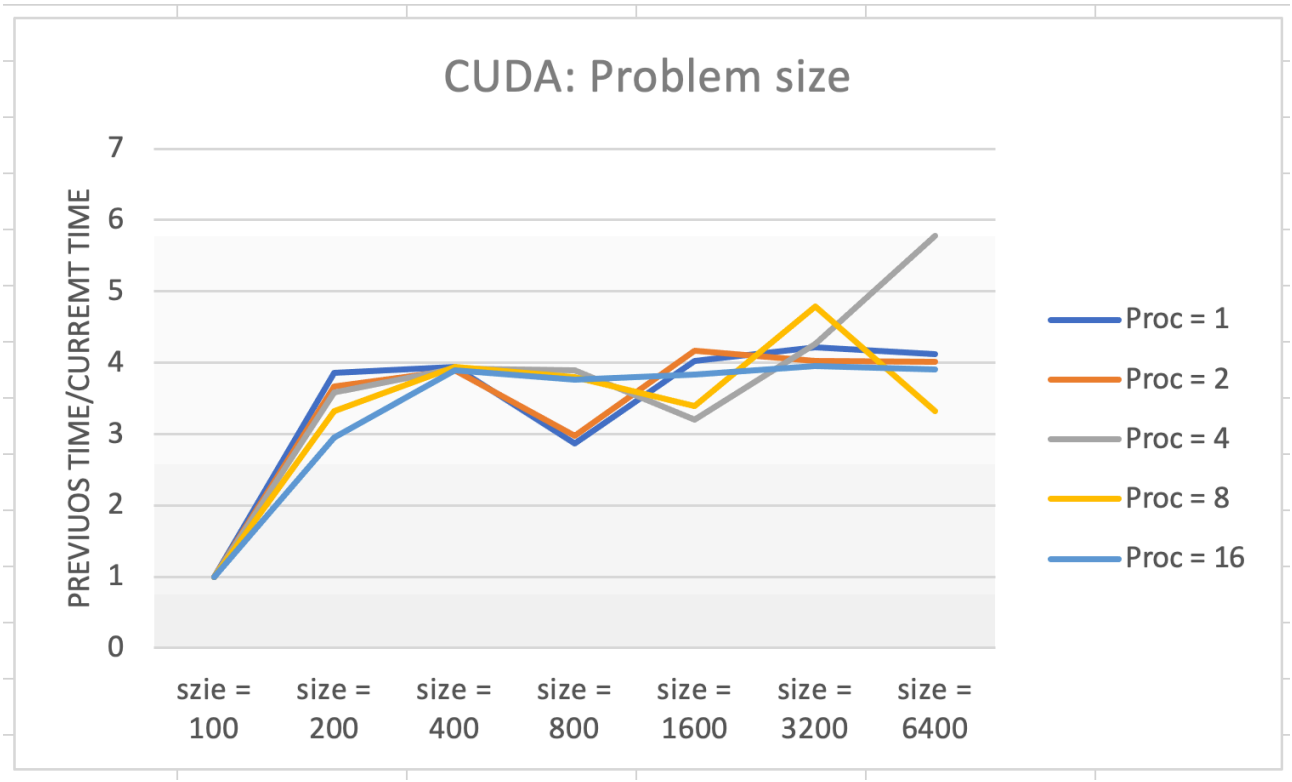
The result is showing below:



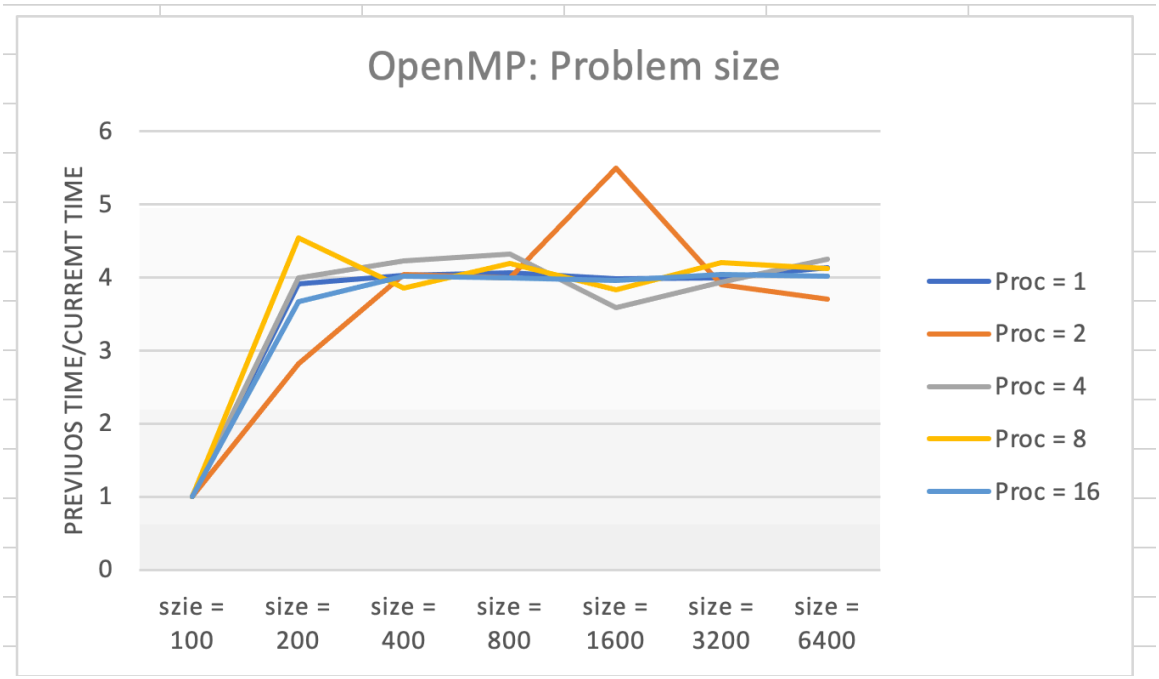
From the figure we can find that, each curve is near 4 when the size number increase 4 times. which will demonstrate the cost optimal problem size will cost propotional speed.



And Pthread have more variance than MPI Version, especially when the thread number is 32. when the size is equal to 800, the performance is worse which will cause the value in size 1600 is extremely high(previou one / current one).



The CUDA version also demonstrates the problem is a embarrassingly parallel computation since the value of it is constantly in the 4.



OpenMP has the similar effect with the other three, which all shows the same result.

-- Discussion:

The figures showing previous all demonstrates the same thing. Heat distribution is a cost optimal one, which means it is the embarassingly program. It will increase the time propotional when the problem size increase. That is becasue for each Algorithm, the heat distribution is **data independent.** which means that the calculation can be executed at the same time for each thread/core. Hence, This question is a cost optimal one.

Analysis 3: Comparision of four parallel version

-- Result:

Four Version Result:

MPI Version:

MPI							
	Size						
#Proc	100	200	400	800	1600	3200	6400
1	1700411	6993981	27950994	112653977	453439868	1816985797	7273334199
2	1751575	7117032	28110009	115244086	451775530	1834265672	7331508432
4	1826859	5434143	21297454	86097303	346566464	1391788754	5550290338
8	1444700	4772628	19123996	74806128	294607608	1177262535	4699952632
16	1294175	4323021	17066513	69681903	279318182	1114727436	4864344049
32	1162900	4193831	19981316	64951300	267797822	1089853935	4797940590
64	1275602	4065542	16888797	67338481	265169753	1118696437	4659872178

Pthread Version:

Pthread							
	Size						
#thread	100	200	400	800	1600	3200	6400
1	1885044	6867196	27375272	109685856	440875411	1760019711	7282023389
2	1836222	5487703	29940820	117466993	467659575	1787803606	7077063139
4	1457650	5386573	22221069	93480869	392842841	1587210426	5927347531
8	1347739	4744950	19198020	74568905	297171744	1263992705	5021561591
16	1293025	4420494	17249852	68124676	275526891	1089222006	4554515242
32	1620122	4543422	16130180	39738411	267868190	1102078857	4416928414
64	2559613	4967600	16793407	69893369	242835876	959068099	3283117108

CUDA Version:

CUDA							
	Size						
#thread	100	200	400	800	1600	3200	6400
1	4076187	15710835	61877719	177067973	713078286	3007366994	12388307872
2	2452457	9002547	35065453	104508492	436114759	1753484599	7044302406
4	1591797	5698265	22353954	86905334	278274290	1185680403	6849208199
8	1184483	3930523	15502340	58888900	199689854	955296800	3174554618
16	1042464	3074791	11961087	45040168	172726586	683648865	2670203424
32	929724	10305743	10324189	37563921	154466517	597856206	2221032842
64	884829	9663234	9459597	31669236	133609797	524836904	2014103502

OpenMP Version:

OpenMP							
	Size						
#thread	100	200	400	800	1600	3200	6400
1	1723504	6743368	27126672	110110451	438047072	1747724929	7232954484
2	1860218	5235345	21180073	84552125	465054348	1811930887	6711298620
4	1346653	5376749	22720435	98140758	352321961	1384587135	5878751680
8	1047196	4756146	18329876	76740104	293970861	1236081719	5097666296
16	1160963	4263907	17126638	68387333	270480895	1094252413	4389052020
32	2885096	8366760	26848321	81944657	166181954	930909221	2454546583
64	3967289	7728010	24834670	67643315	162225242	1039219139	3937602928

And The Final Comparasion is showing below:

Compare	100	200	400	800	1600	3200	6400
1	1700411	6743368	27126672	109685856	438047072	1760019711	7273334199
2	1836222	5487703	29940820	117466993	467659575	1787803606	6711298620
4	1457650	5386573	22221069	93480869	392842841	1587210426	5878751680
8	1347739	4744950	19198020	74568905	297171744	1263992705	5097666296
16	1293025	4420494	17249852	68124676	275526891	1089222006	4389052020
32	929724	10305743	16130180	39738411	267868190	1102078857	2454546583
64	884829	9663234	9459597	31669236	133609797	524836904	2014103502

MPI
Pthread
CUDA
OpenMP

From the chart, it shows the best parallel approach under the certain configuration (fixed number of cores/threads and fixed problem size). The yellow block indicates the best approach is MPI. The orange approach indicates the best approach is Pthread. The blue block indicates the best approach is OpenMP. The green block means the best approach is CUDA.
The result shows that when problem size is relatively small or the number of cores/threads is small, MPI takes the most advantage among all the approaches. When the problem size is middle(800-1600) and large(>1600), Pthread outperforms others. And when the problem size is extremly lagrge, the OpenMP performs better than others. What's more, when it scales to a greater number of threads, CUDA becomes the best one, and when the thread/core number is very large, CUDA outperform than others.

-- Discussion:

- The advantage of MPI can be explained by the communication overhead is still not significant when the problem size is small. According to the time complexity formulation got in the analysis, if the time of message passing one data is close to the time of updating one point, then adding the problem size or number of cores would both make the communication overhead gradually significant compared with the time-saving brought by parallel computing. In this problem, That is mainly becasue MPI can have muti core to process and compute the grid while pthread and others only have one core to do the computation, which will cost much more time than MPI in the computation. At the same time, due to few elements input, MPI will also have less communication between each core. Thus, MPI could perform better than others when the input size is not very large.
- As for the CUDA, since it has lightweight threads and scalibility, when the number of threads increase to large size, it's performance will outperforms others. This result verifies the scalability of CUDA threads when problem size is between 800 and 3200, which further explains why when number of threads/cores is large, CUDA will take advantage among other three approaches. In this problem, since CUDA use GPU not CPU, then when we

use CUDA, we may need time to launch the kernel function and also need time to pass all the variable to the GPU, and after the calulation CUDA need to send the value back to the CPU, which will be time consuming. since GPU has many cores to do the calculation, so when the number of problem size increase, CUDA may perform better than other. But for more complicated computation, CUDA may perform worse than others since CUDA are unable not do the high-level calculation.

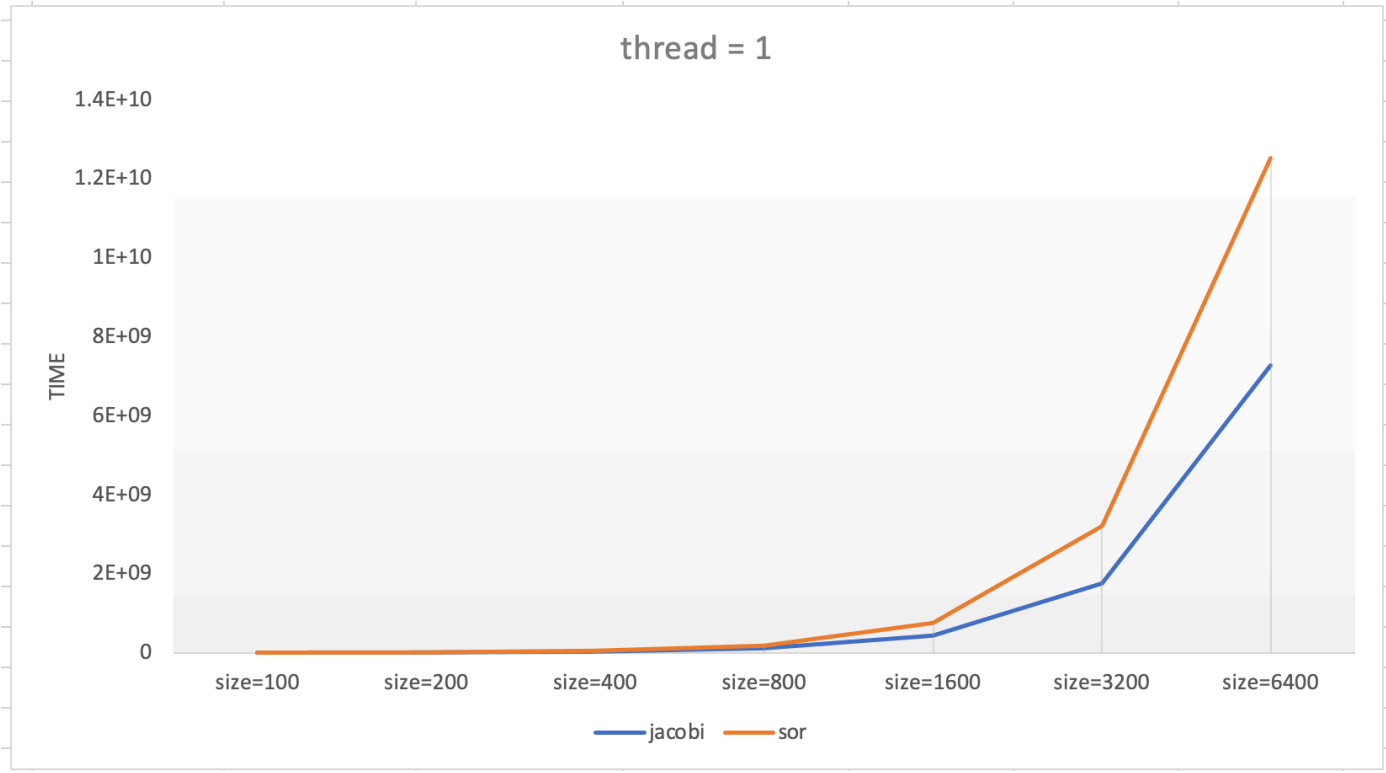
- As for the Pthread approach, when problem size is large, it avoids the significant communication overhead of transfer data. it is also free from the memory copy latency like CUDA which is also significant when problem size is large, and avoids the overhead to dynamically schedule the large size of problem like OpenMP. Therefore, Pthread outperforms when the problem size is large. However, when the number of threads increase (greater than 32 threads), **over-spawn** occurs so the perfromance of Pthread would cease to incease and finally be outperformed by other approaches. In this problem, When the problem size and number of core/thread is large, Pthread will performs better than MPI. Especially, When the size is very large(1600), MPI performs very poor compared with pthread. that is mainly becasue when the problem size increase, the communication beteween each core will aslo increase as well. So MPI will cost much time on communication, while pthread have the advantage of sharing memory. Pthread does not need to communicate between each thread since they can modify their own canvas in their scope. Moreover, MPI need much time on synchronisation since it need to synchronize the root rank and other rank. Ptrhead only need the join function to join every thread together, which cost less time in communication.
- For OpenMP, OpenMP performs as close as the Pthread, But OpenMP most of the time perform little worse than pthread in this project. The difference is that: Pthreads is a very low-level API for working with threads. On the other hand, OpenMP is much higher level, is more portable and doesn't limit you to using C. But lauchg more high-level API may have more chance to use more hidden function that we dont even know it laugh. So the low-level creation thread will not be explicit to us. It may consume more time to lauch openMP thread, so it may be little worse than Pthread. But when the size is extreme large, OpenMP can perform better than others, that may becase OpenMP have dynamically scheduling method to allocate each thread to divide the tasks more evenly, which will mainly reduce the sheduling overhead than Pthread.

Analysis 4: The Comparision of different algorithm

-- Result:

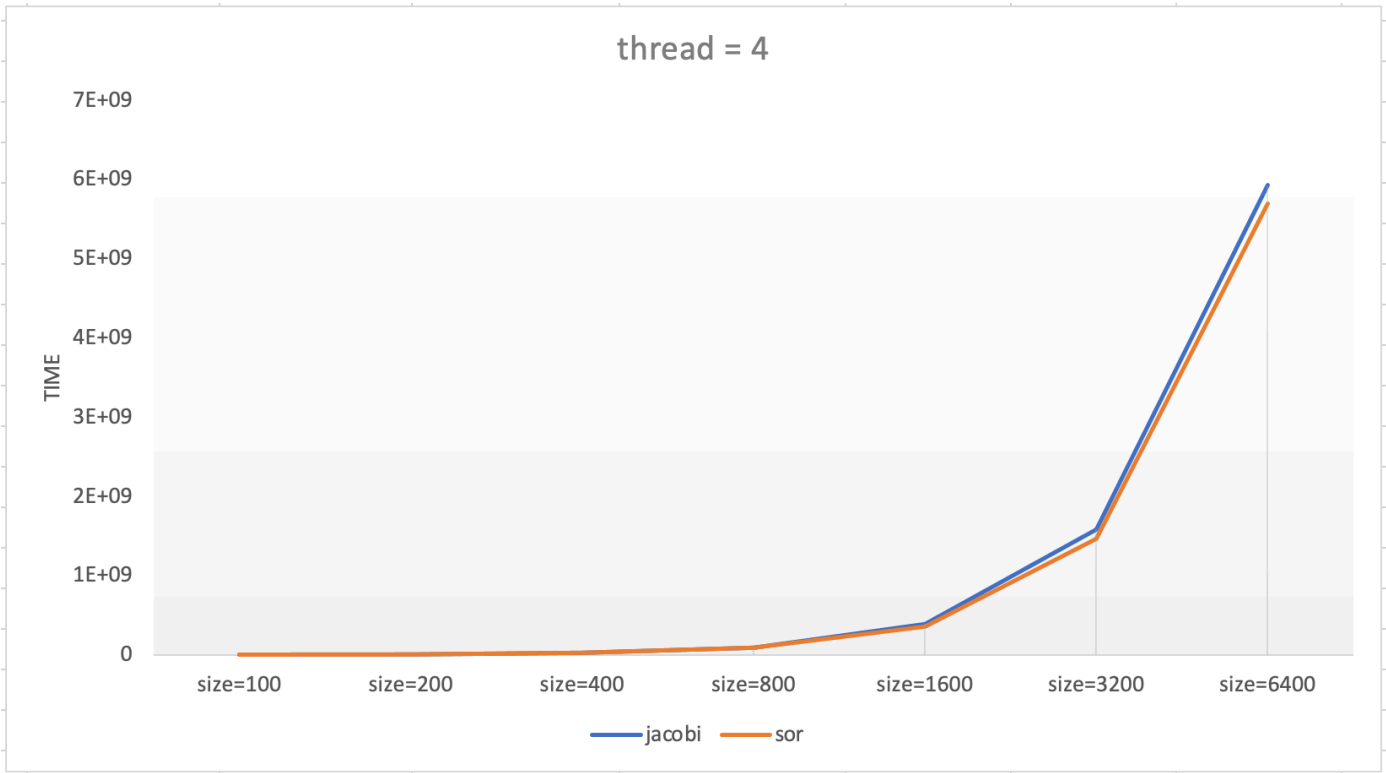
since the calcualtion logic in four version is simply the same for two algorithm. So here I only choose Pthread version and MPI version to compare those performance.

thread number is 1:

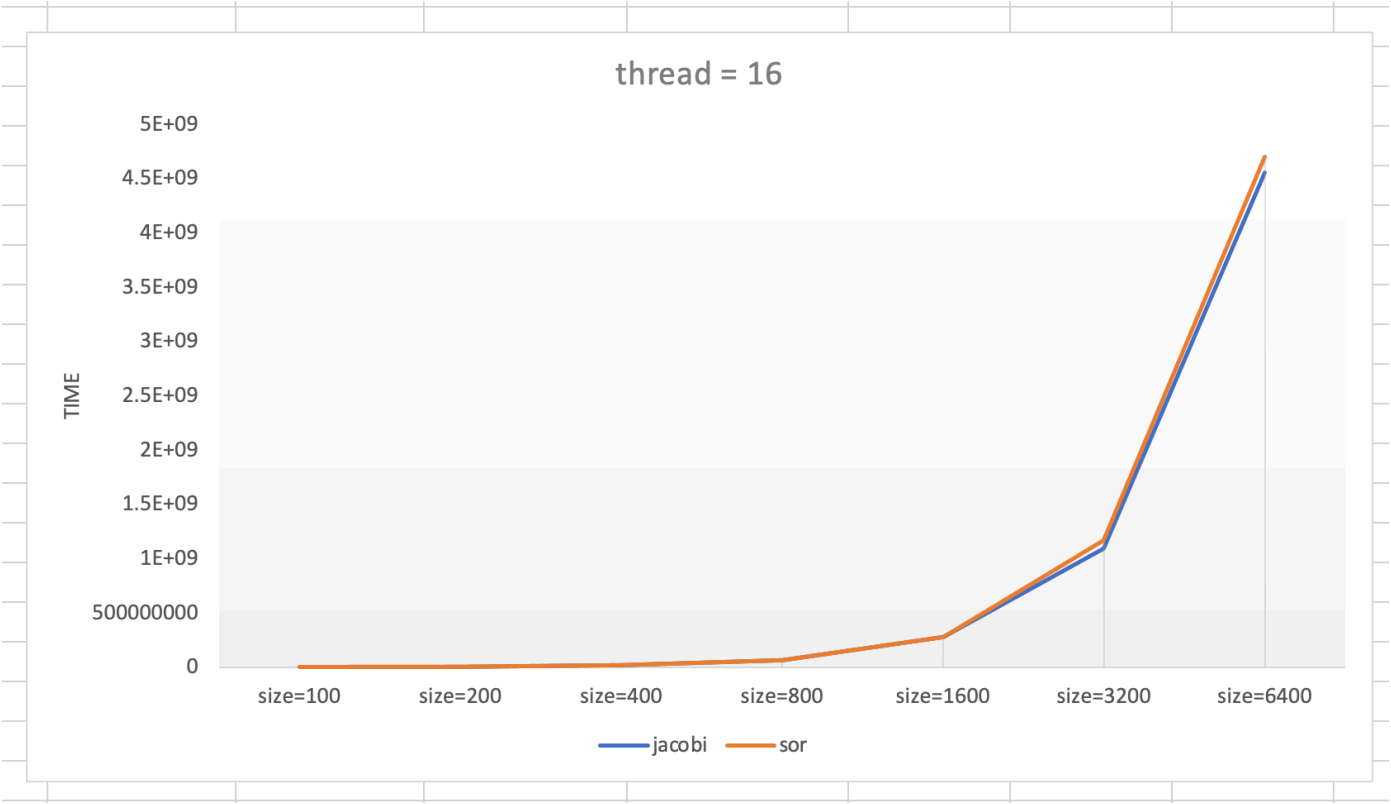


thread number is 4

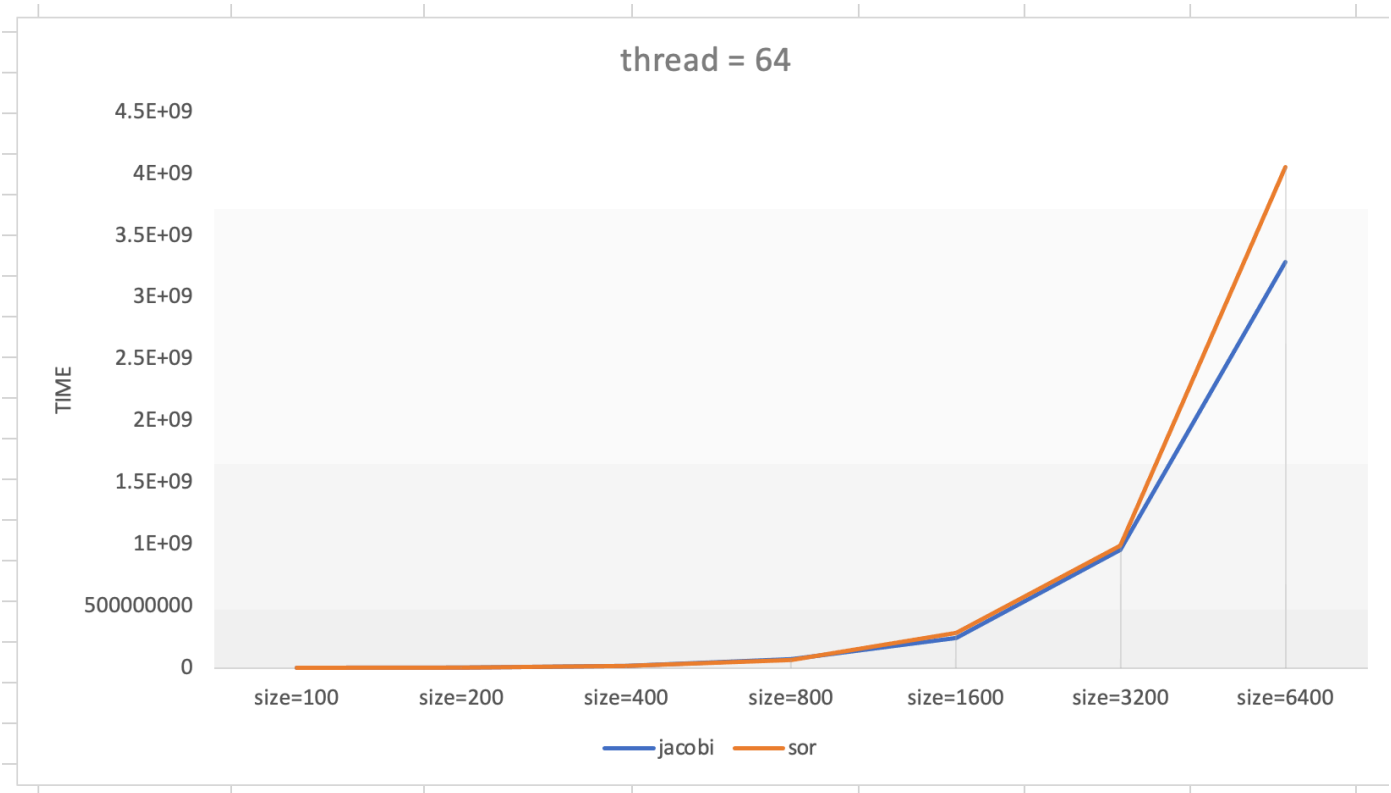
:



thread number is 16:



thread number is 64:



From the figure showing previous, we can found that when we apply single thread to calculate those two algorithm, the `Jacobi` one will perform better. As the thread number increase, the difference between those algorithm will reduce at the same time. And even when the thread number is 4 and when the size is large the `Sor` performs better than `Jaobi`. When the thread number reach 64, the `Jacobi` one will perform much better than `Sor`.

-- Discussion:

The reasons why Jacobi is the algorithm more suitable to be paralleled than Sor lies on two reasons.

First, For MPI program, it need Sor algorithm need to call `MPI_Allgatherv()` once and `MPI_Bcast` while Jocobi only need to calls `MPI_AllgatherV`, the code below shows the core idea of computation and communication:

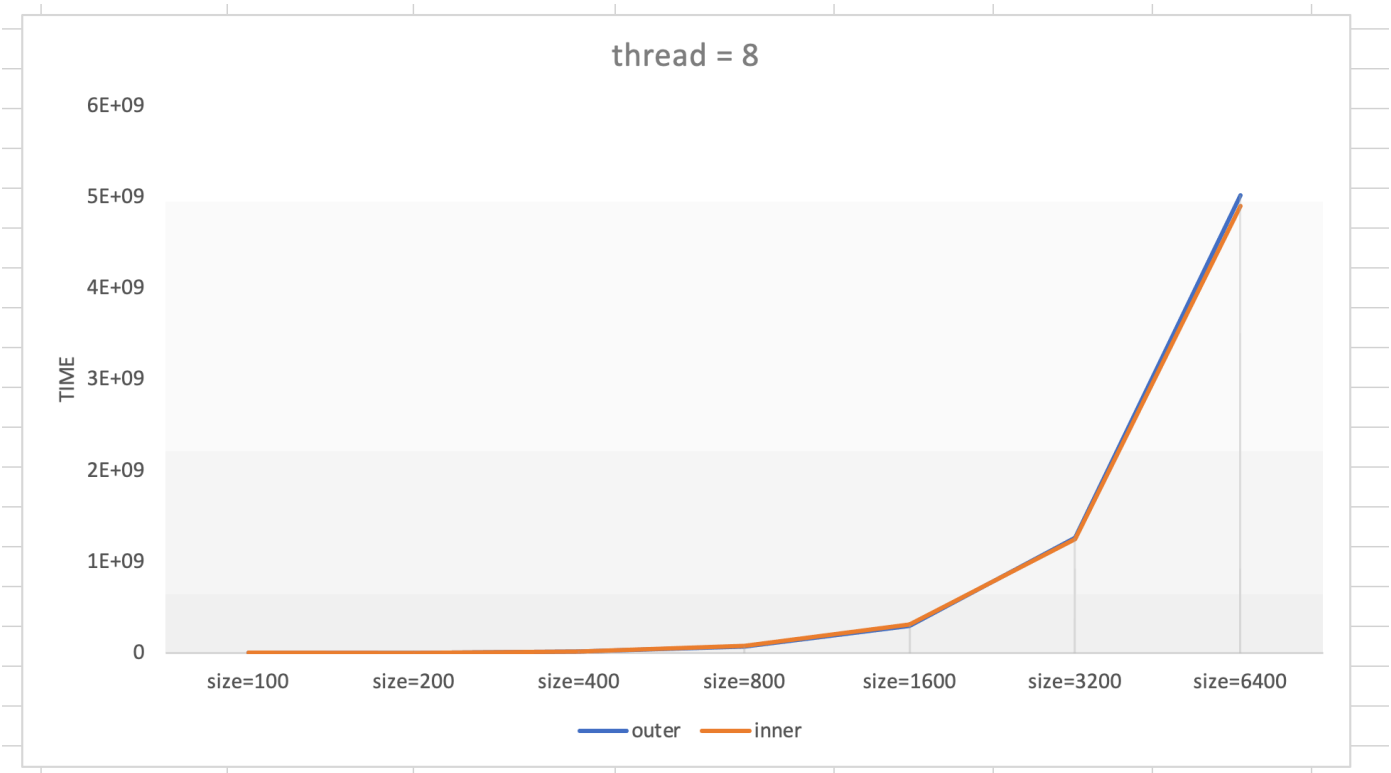
```
switch (state.algo) {
    case Algorithm::Jacobi:
        for (size_t i = min; i < max; ++i) {
            for (size_t j = 0; j < state.room_size; ++j) {
                auto result = update_single(i, j, grid, state);
                stabilized &= result.stable;
                grid[{alt, i, j}] = result.temp;
            }
        }
        grid.switch_buffer();
        MPI_Allgatherv(&grid.get_current_buffer()[displ[rank]], recvcounts[rank] , MPI_DOUBLE,
&grid.get_current_buffer()[0],recvcounts,displ,MPI_DOUBLE,MPI_COMM_WORLD );
        break;
    case Algorithm::Sor:
        for (auto k : {0, 1}) {
            for (size_t i = min; i < max; i++) {
                for (size_t j = 0; j < state.room_size; j++) {
                    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}];
                    }
                }
            }
            grid.switch_buffer();
            MPI_Allgatherv(&grid.get_current_buffer()[displ[rank]], recvcounts[rank] ,
MPI_DOUBLE, &grid.get_current_buffer()[0],recvcounts,displ,MPI_DOUBLE,MPI_COMM_WORLD );
            MPI_Bcast(&grid.get_current_buffer()[0], state.room_size * state.room_size ,
MPI_DOUBLE, 0, MPI_COMM_WORLD);
        }
}
```

Second, there are more times for memory access of Sor compared with Jacobi. In the code below, we can obviously see that Sor involves more times of shared memory access. Notice that when the problem size is large, the latency of accessing the shared memory is increasingly significant. So the disadvantage of Sor is gradually enlarged as the problem size increases.

Analysis 5: Compare parallize inner loop and outer loop

The reason I compare the outer loop is simply because theroticaly, outer loop parallization will have less sheduling overhead than inner loop, since it have no need to re allocate the reasource for another same calculation. To verify that, I choose Ptrhead to see the difference between inner loop and outer loop.

-- Result:



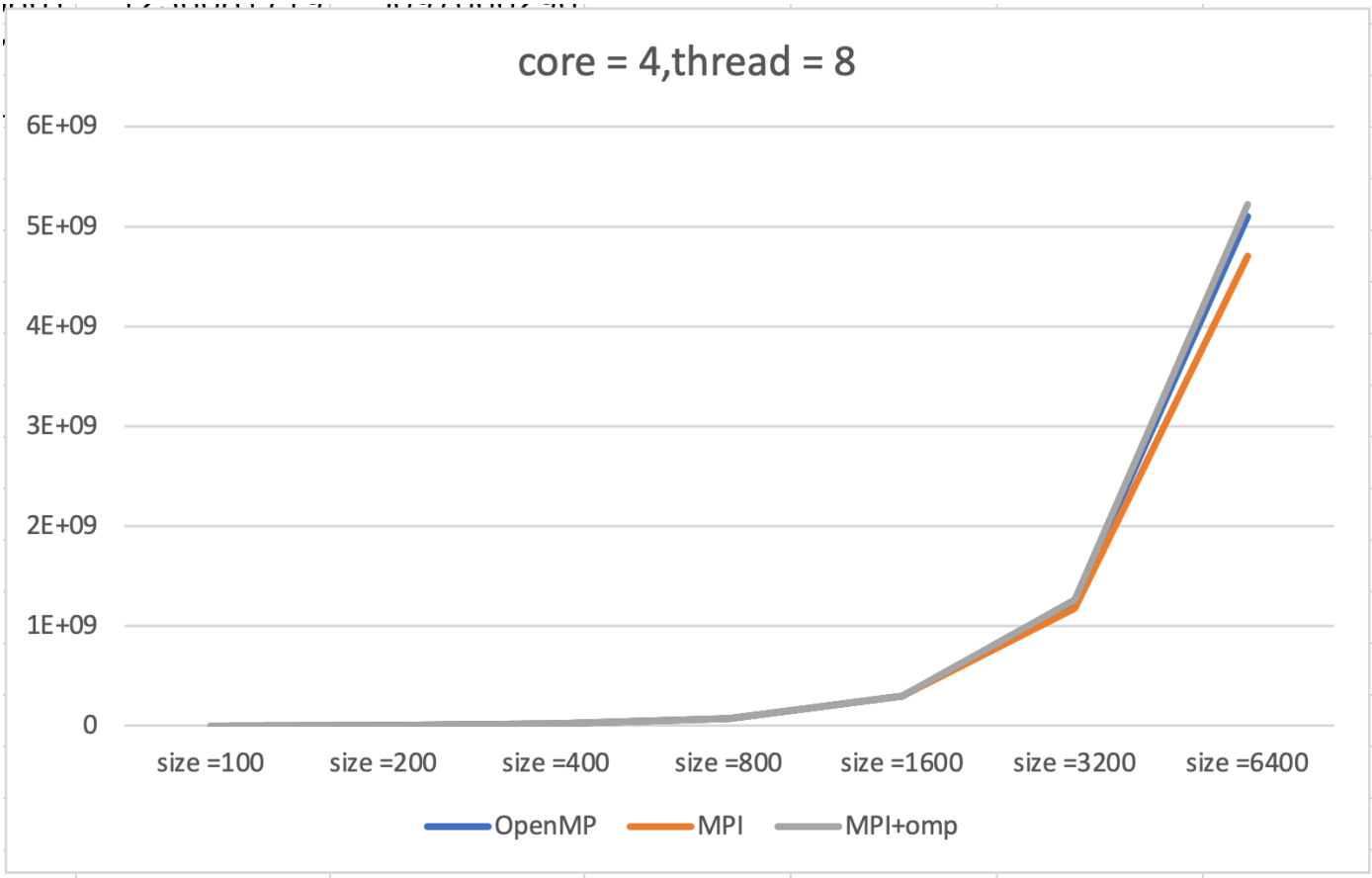
-- Discussion:

From the figure we can observe that, the inner loop and outer loop has slight difference of each problem size. Especially when the size is large, the difference is larger, that is beacuse the inner loop have more overhead in data sheduling and it will cost much time to allocate the resourse for calculation. hence it will take more time to allocate the resource.

Analysis 6: Bouns part: MPI+OpenMP vs OpenMP

-- Result:

The bonus part show the result of MPI + OpenMP performance.



-- Discussion:

We can observe that when we add more threads in the MPI version, It will not always increase the performance. instead, When the size is small, omp+mpi version performs not good as MPI. But When the size increase, the performance of MPI+OpenMP will increase and perform better than MPI and OpenMP. since a node has only 32 core. It can be understood that a node can open countless threads, but only 32 threads run at the same time, which is scheduled by the OS (in simple terms, imagine round robin, and each thread runs in time slice). The 32+ core can only be used in combination with mpI, which synchronizes data to process on multiple nodes, each of which is responsible for pulling multiple threads. Hence when the size is large, The OpenMP + MPI version could performs better than these two. However, when the size is too small, too many thread for one node will decrease the speed instead since it need time to matain the Synchronous.

Conclusion

- In general, Parralle computation will performs well nearly in every case than sequential one
- in general, applying more threads or cores for the heat distribution computation would increase the parallel computing performance and the increaing tendency is stable.
- The problem is cost optimal one, which have no data dependency.
- For small size and number of cores/threads, MPI will performs better than CUDA, Pthread and OpenMP computation. For large size and more cores/threads, Pthread has the better performace in computation than others. OpenMP has the cloest value to Pthread. CUDA performs better when the thread/core number is very large.
- The algorithm Jacobi is generally speaking better than Sor.
- The MPI+OpenMP Version will increase the performance when the problem size and thread number is small. otherwise, the single OpenMP or single MPI version is better.

Demo Output

Important Declaration:

I choose the size 200 to demonstrate my result other choose the 800. That is mainly because to reduce the vedio space. hope it won't affect my grade. many thanks.