香港中文大學(深圳)
The Chinese University of Hong Kong, Shenzhen

Heat-Distribution Simulation

Assignment IV
CSC4005: Parallel Programming

Name: *Derong Jin*
Student ID: *120090562*

Date: December 5, 2022

1 Introduction

In the field of heat simulation, jacobi integration is widely used because of it intuitively simulate the heat transfer process and find out a stand point in the end. This algorithm is used when laplacian operator is well defined, e.g. manifolds, meshes, and grids. This operator measures how the value of a point differ from its neigbourhoods. In 2d grid system in this project, the laplacian gives:

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

The jacobi integration is using this laplacian to replace the value of $h_{i,j}$ repeatedly unitl converge. This procedure takes $O(N)$ times obiviously, where N is the number of vertices in the system.

In this project, the object is speed up the this method via parallel approaches including using MPI, Pthread, OpenMP, CUDA, and hybrid method of MPI + OpenMP. And I will compare on their runtime and analyze on the results.

2 Method

2.1 Basic Framework

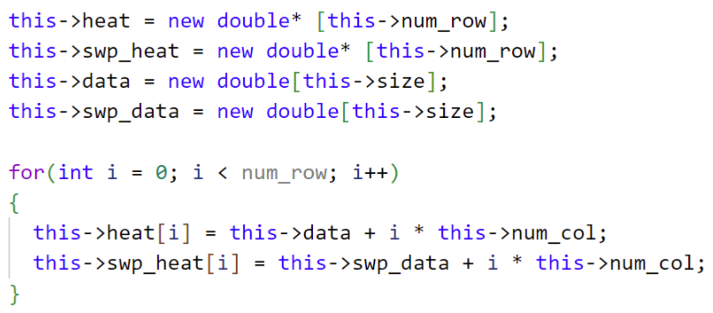
From earlier introduction, it is shown that Jacobi integration is a favorable method to solve the heat-distribution problem on a connected grid system. Meanwhile, the purpose of this project is to implement this method and parallelize it using different parallelization interface. To conduct a fair comparison between these parallelization methods, as well as reduce the redundancy in program, a framework that encloses reusable steps is preferred. One possible way to develop such framework is seperate the calculation into steps and implement reusable parts in an parent class. When implementing a parallel program, it can be done by first inheritate the parent class and use pre-implemented function. This design simplize the implementation of each parallel program, and thus reduce the chance of logical flaws to happen.

In this problem, the calculation is derived in following steps:

1. initialize the heat state of the room, (initialize the positioin of the fire)
2. apply Jacobi Intergal on every grid vertex and store the result into a buffer of the same size.
3. set the heat value to the updated one in the buffer.
4. repeat from step 2 until converge or the number of iteration exceeds a given value.

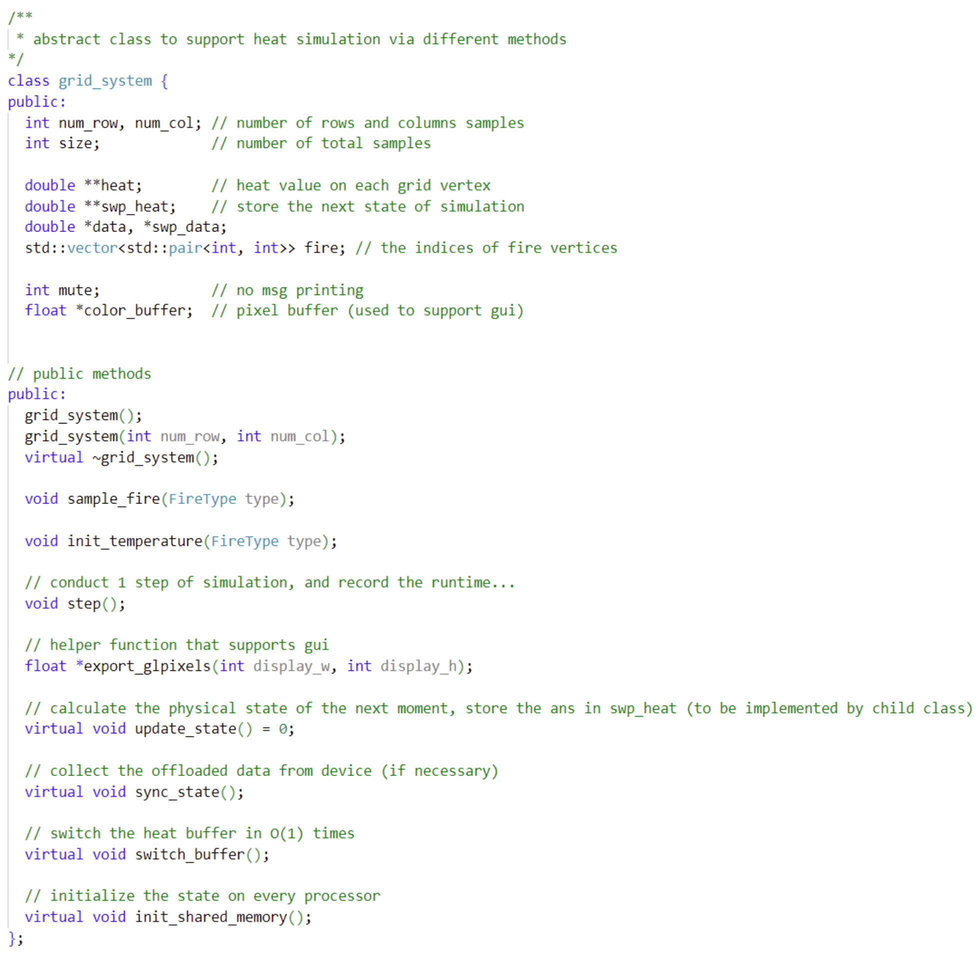
Notice that step 3 can be done in $O(1)$ time by swapping the pointers of buffers instead of the values within them. Therefore the bottleneck of this calculation is the intergal process, which takes $O(N)$ times sequentially.

In order to make this process easier to both parallize and implement, the framework allow data-accessing in one- and two-dimension manner by making the buffer row dominant and continuous:



```cpp
this->heat = new double* [this->num_row];
this->swp_heat = new double* [this->num_row];
this->data = new double[this->size];
this->swp_data = new double[this->size];

for(int i = 0; i < num_row; i++)
{
  this->heat[i] = this->data + i * this->num_col;
  this->swp_heat[i] = this->swp_data + i * this->num_col;
}
```

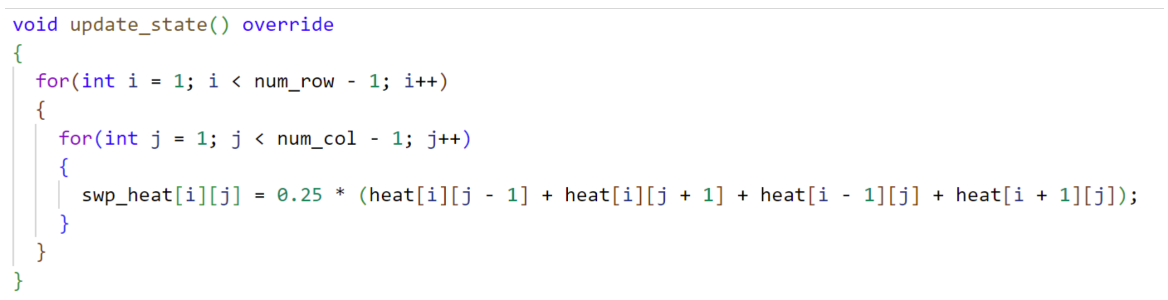
Finally, we shaw introduce C++ defination of this framework abstract class:



```cpp
/**
 * abstract class to support heat simulation via different methods
*/
class grid_system {
public:
  int num_row, num_col; // number of rows and columns samples
  int size;             // number of total samples

  double **heat;        // heat value on each grid vertex
  double **swp_heat;    // store the next state of simulation
  double *data, *swp_data;
  std::vector<std::pair<int, int>> fire; // the indices of fire vertices

  int mute;             // no msg printing
  float *color_buffer;  // pixel buffer (used to support gui)


// public methods
public:
  grid_system();
  grid_system(int num_row, int num_col);
  virtual ~grid_system();

  void sample_fire(FireType type);

  void init_temperature(FireType type);

  // conduct 1 step of simulation, and record the runtime...
  void step();

  // helper function that supports gui
  float *export_glpixels(int display_w, int display_h);

  // calculate the physical state of the next moment, store the ans in swp_heat (to be implemented by child class)
  virtual void update_state() = 0;

  // collect the offloaded data from device (if necessary)
  virtual void sync_state();

  // switch the heat buffer in O(1) times
  virtual void switch_buffer();

  // initialize the state on every processor
  virtual void init_shared_memory();
};
```

2.2 Sequential Implementaion

The sequential version is the easiest implementation, simply inheritate the abstract class and implement the `update_state` method by going through every grid vertex and update the value of it with the average value of adjacant vertices is working.



```
void update_state() override
{
  for(int i = 1; i < num_row - 1; i++)
  {
    for(int j = 1; j < num_col - 1; j++)
    {
      swp_heat[i][j] = 0.25 * (heat[i][j - 1] + heat[i][j + 1] + heat[i - 1][j] + heat[i + 1][j]);
    }
  }
}
```

Figure 1: Core code of Jacobi Integration using sequential method

One update takes $O(N)$ times, where N is the number of grid vertices in the system.

2.3 Parallization With MPI

When parallizing such a solver, it is for sure every processor will only handle a part of calculation, the main concern now is how to partition and distribute the problem to processors. Since the sequential version takes $O(N)$ times to complete on state updation, it is too expensive to broadcast the whole state to each processor, which has no improvement due to the high communication latency.

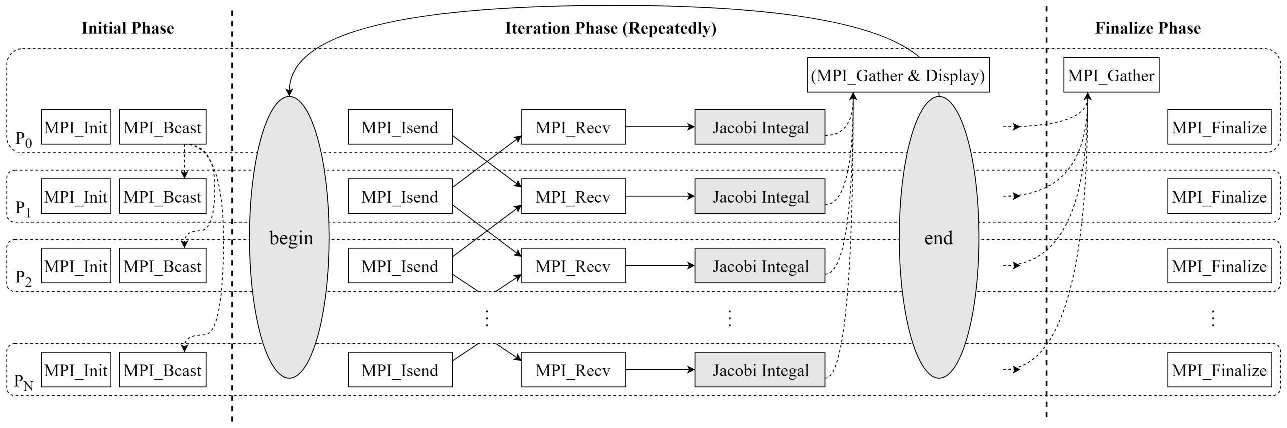
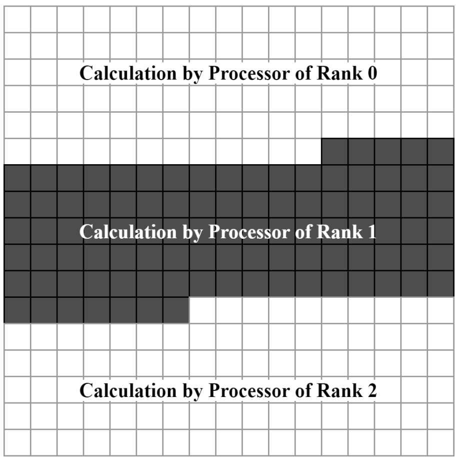
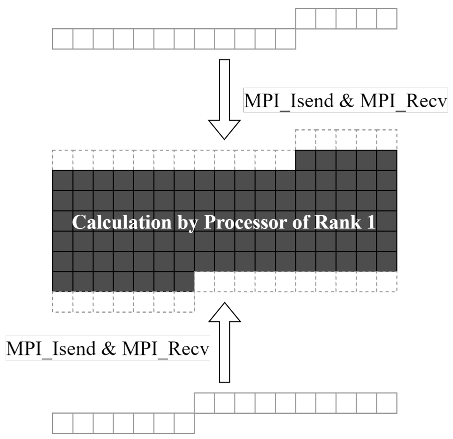


Figure 2: This diagram mainly illustrates the design of MPI program in this project. It can be seen that each processor is viewed as an independent part and do the calculation statically allocated, and communication is minimized by only allowing the boundary state to transfer between processors.

My solution is to partition the grid system into N_p horizontal blocks, where N_p is the number of processors used. The processor rank refers to the block index in the grid system (In this design, I assume the number of rows is larger than the number of processors, otherwise one processor takes the work of grid vertices in one row). Before calculation, the vertex temperatures on the up and down boarder should be synchronized by applying inter-processor communication with `MPI_Isend` and `MPI_Recv`. This design reduce the size of communication from $O(N)$ to $O(\sqrt{N})$ under the situation of square grid system. The latency caused by such communication size is acceptable and being closer to the threotical lowerbound of $(\sqrt{\frac{N}{N_p}})$.


Calculation by Processor of Rank 0
Calculation by Processor of Rank 1
Calculation by Processor of Rank 2

(a) An example partition of 17×17 grid vertices with 3 processors, namely P_0, P_1 and P_2. In order to divide the problem evenly to keep fairness, the boundary is not necessarily a straight row on the mesh.



(b) Message about heat value on top and bottom boundary which is of size $O(\sqrt{N})$ should be sent by adjacant processors (P_0 and P_2) and received by P_1 before the calculation on P_1.

2.4 Parallization With Multithreads (Pthread & OpenMP)

The parallelization on shared memory system is much easier, since the partition method has no effect on communication latency since the states are updating on the same buffer. The only important issue in this situation is to keep the fairness between processors. From the formulation of Jacobi Integration, the workload on calculating each vertex's heat value is the same, applying static job allocation is favorable.

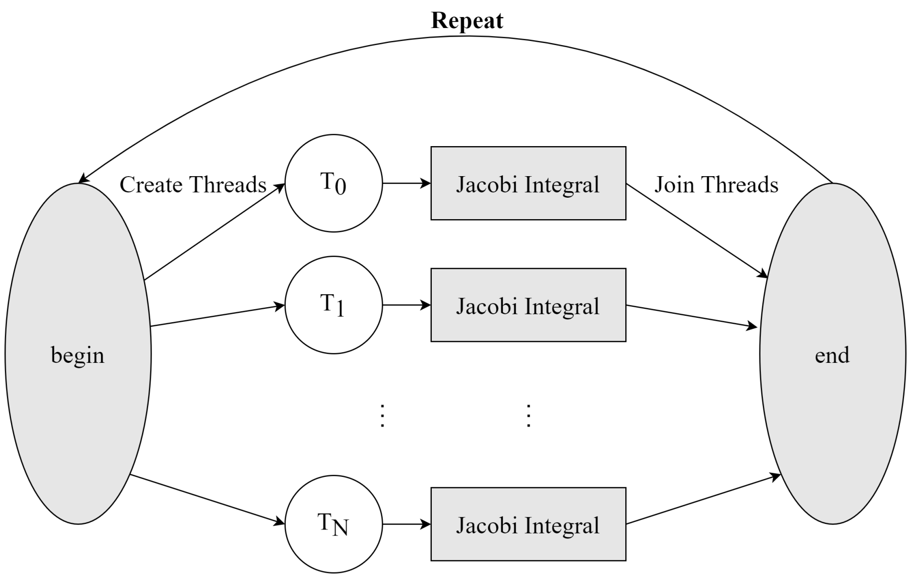


Figure 3: This diagram mainly illustrates the design of multi-thread parallelization of one iteration. Since the threads are working in the same memory space, there is no need to communicate between threads if their works are statically allocated, and thus saves time threotically

2.5 Parallization With GPU (CUDA)

The idea of CUDA parallelization is simple, one worker (thread) only takes the responsibility to update the state of one grid vertex. CUDA supports allocating blocks and thread in various manners, in this project these resources are enumerated in one dimension. This design fits the property of the given simulation task well, where grid value can be updated in one dimensional manner as well: `buff[i] = (s[i + 1] + s[i - 1] + s[i + M] + s[i - M]) / 4`, where `s` is the 1-dimensional access to the previous heat state, `M` is the number of column size of the grid system. This one-dimensional updation is portable to CUDA calculation, first copy the initial heat state to cuda device via `cudaMemcpy` and lauch kernel function `solve_jacobi_step<<<BlockNum, ThreadsPerBlock>>>` to do updation. When there is need to display or export the calculation result, another `cudaMemcpy` is called to collect the data from the GPU device.

The diagram of the calculation process is shown as below:

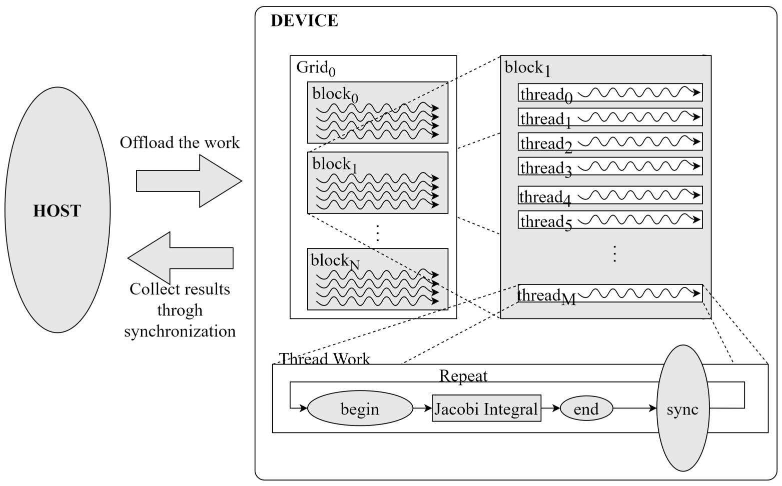


Figure 4: This diagram shows the calculation flow using GPU. The host (CPU) sends the initial heat state to device (GPU) and collect result ater GPU finishes its work. When calculating, GPU creates multiple threads within multiple blocks and the total number of threads equals to the number of vertices in the grid system. Therefore, one threads takes the responsibility to do the updation of one grid vertex.

2.6 Bonus: Parallization With MPI+OpenMP

Clearly is that MPI only does not meake full use of resources allocated to the process. Hence this project uses multithreaded approach to utilize these calculation resources (mostly CPU time) via OpenMP interface.

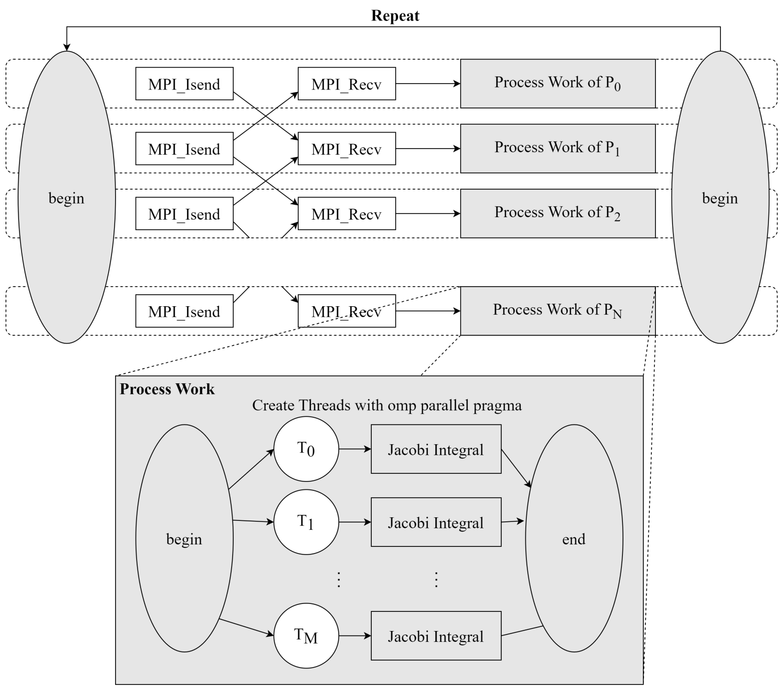


Figure 5: This diagram shows the modified iteration phase of MPI which supports OpenMP multithreading in each process.

3 Results & Analysis

3.1 GUI Results

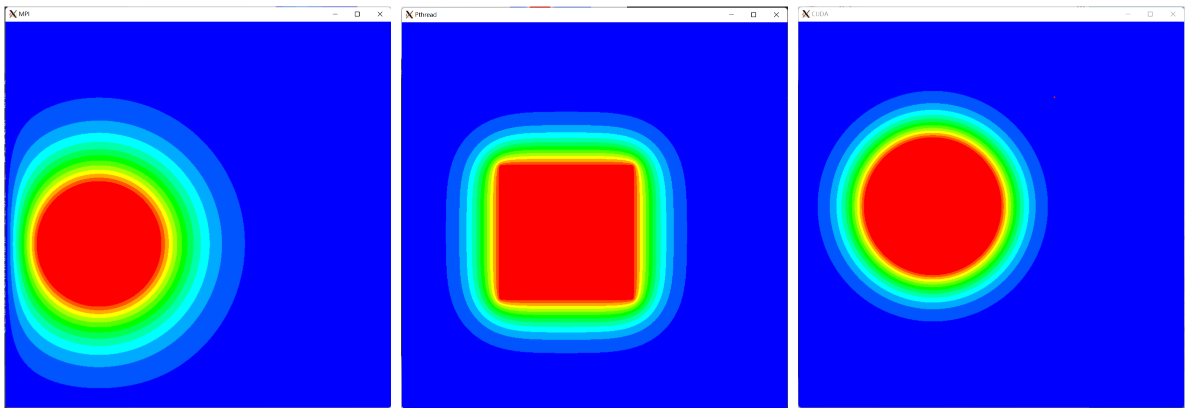


Figure 6: GUI Demo

Above we show three cases of gui and visualization of the heat state. The color is defined in HSL space and ranges from red to blue, which represent the temperature on the grid vertices of the system from hot (90℃) to cold (20℃), the color is set to change until there is a change of 5 ℃.

3.2 Performance Under Different Number of Cores/Threads

In the best case, the speedup should be propotional to the number of processors. However this situation is too ideal in real-life programming, this is caused by the time consumption in communication and the efficiency loss in contention for shared memory. This section is mainly focus on this issue, and thus test the project programs (MPI, Pthread, & OpenMP in this section, I did not include CUDA because the parallism of CUDA is much different and the comparison with it will not give reliable conclusion) under different number of processors, which ranges from 1 to 48. Analysis on the speedup factor is also conducted later in this section.

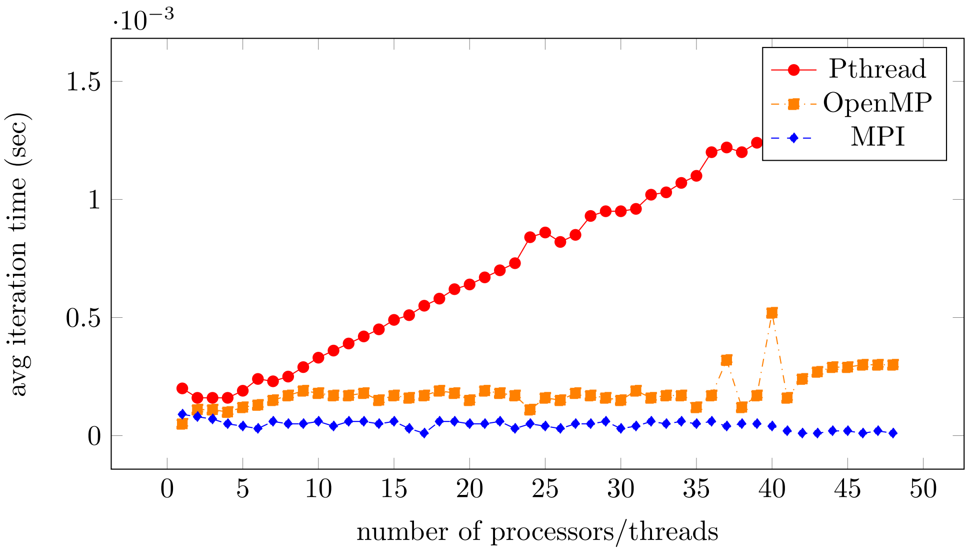


Figure 7: The time vs. number of processors graph with grid size = 100^2

	1	2	3	4	5	6	7	8	9	10	11	12
Pthread, 100^2	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0003	0.0003	0.0003	0.0004	0.0004
MPI, 100^2	0.0001	0.0001	0.0001	0.0000	0.0000	0.0000	0.0001	0.0001	0.0000	0.0001	0.0000	0.0001
OpenMP, 100^2	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002
	13	14	15	16	17	18	19	20	21	22	23	24
Pthread, 100^2	0.0004	0.0005	0.0005	0.0005	0.0006	0.0006	0.0006	0.0006	0.0007	0.0007	0.0007	0.0008
MPI, 100^2	0.0001	0.0001	0.0001	0.0000	0.0000	0.0001	0.0001	0.0001	0.0001	0.0001	0.0000	0.0000
OpenMP, 100^2	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0001	0.0002	0.0002	0.0002	0.0001
	25	26	27	28	29	30	31	32	33	34	35	36
Pthread, 100^2	0.0009	0.0008	0.0009	0.0009	0.0010	0.0010	0.0010	0.0010	0.0010	0.0011	0.0011	0.0012
MPI, 100^2	0.0000	0.0000	0.0001	0.0000	0.0001	0.0000	0.0000	0.0001	0.0000	0.0001	0.0000	0.0001
OpenMP, 100^2	0.0002	0.0001	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0001	0.0002
	37	38	39	40	41	42	43	44	45	46	47	48
Pthread, 100^2	0.0012	0.0012	0.0012	0.0013	0.0013	0.0013	0.0014	0.0014	0.0014	0.0015	0.0015	0.0015
MPI, 100^2	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000	0.0000
OpenMP, 100^2	0.0003	0.0001	0.0002	0.0005	0.0002	0.0002	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003

The data and figure above display the average iteration runtime under various process number in a small problem size. In this set of experiment data, there is no sign on the improvement of performance when additional processors are introduced. In the case of pthread program, the performance goes even worse when number of processors is increasing. From the graph the runtime of pthread program is approximately propotional to the number of processors. A reasonable guess is that the cost of creating a pthread is much larger than that of state updation. The dominant part of program runtime is the time used to create and join threads.

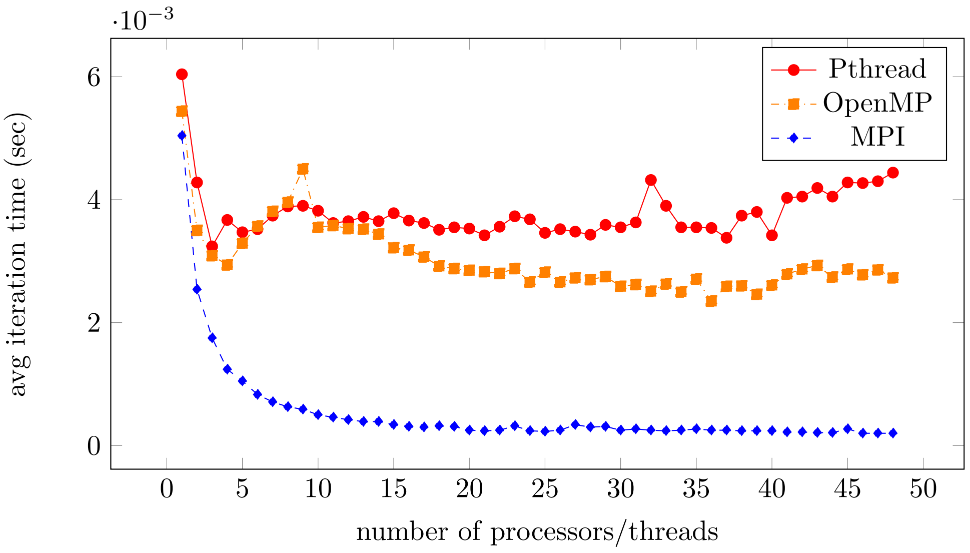


Figure 8: The time vs. number of processors graph with grid size = 1000^2

	1	2	3	4	5	6	7	8	9	10	11	12
Pthread, 1000^2	0.0060	0.0043	0.0032	0.0037	0.0035	0.0035	0.0037	0.0039	0.0039	0.0038	0.0036	0.0037
MPI, 1000^2	0.0050	0.0025	0.0017	0.0012	0.0010	0.0008	0.0007	0.0006	0.0006	0.0005	0.0005	0.0004
OpenMP, 1000^2	0.0054	0.0035	0.0031	0.0029	0.0033	0.0036	0.0038	0.0040	0.0045	0.0035	0.0036	0.0035

	13	14	15	16	17	18	19	20	21	22	23	24
Pthread, 1000^2	0.0037	0.0037	0.0038	0.0037	0.0036	0.0035	0.0036	0.0035	0.0034	0.0036	0.0037	0.0037
MPI, 1000^2	0.0004	0.0004	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0002	0.0002	0.0003	0.0002
OpenMP, 1000^2	0.0035	0.0034	0.0032	0.0032	0.0031	0.0029	0.0029	0.0029	0.0028	0.0028	0.0029	0.0027

	25	26	27	28	29	30	31	32	33	34	35	36
Pthread, 1000^2	0.0035	0.0035	0.0035	0.0034	0.0036	0.0035	0.0036	0.0043	0.0039	0.0035	0.0035	0.0035
MPI, 1000^2	0.0002	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0003	0.0002	0.0003	0.0003	0.0003
OpenMP, 1000^2	0.0028	0.0027	0.0027	0.0027	0.0027	0.0026	0.0026	0.0025	0.0026	0.0025	0.0027	0.0023

	37	38	39	40	41	42	43	44	45	46	47	48
Pthread, 1000^2	0.0034	0.0037	0.0038	0.0034	0.0040	0.0041	0.0042	0.0041	0.0043	0.0043	0.0043	0.0044
MPI, 1000^2	0.0003	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0002	0.0003	0.0002	0.0002	0.0002
OpenMP, 1000^2	0.0026	0.0026	0.0025	0.0026	0.0028	0.0029	0.0029	0.0027	0.0029	0.0028	0.0029	0.0027

From experiment data above, we see the parallelization takes effect in a larger problem scale (compare to last experiment). All of these three programs have improvement when increasing the number of processors. The most improvement is ahieved by the MPI program, where the runtime is curved like a inversion propotional function. The reason might because MPI program reduces its communication to a favorable degree, i.e. $O(\sqrt{N})$ of communication is relatively ignorable comparing to the $O(\frac{N}{N_p})$ computation in state updation.

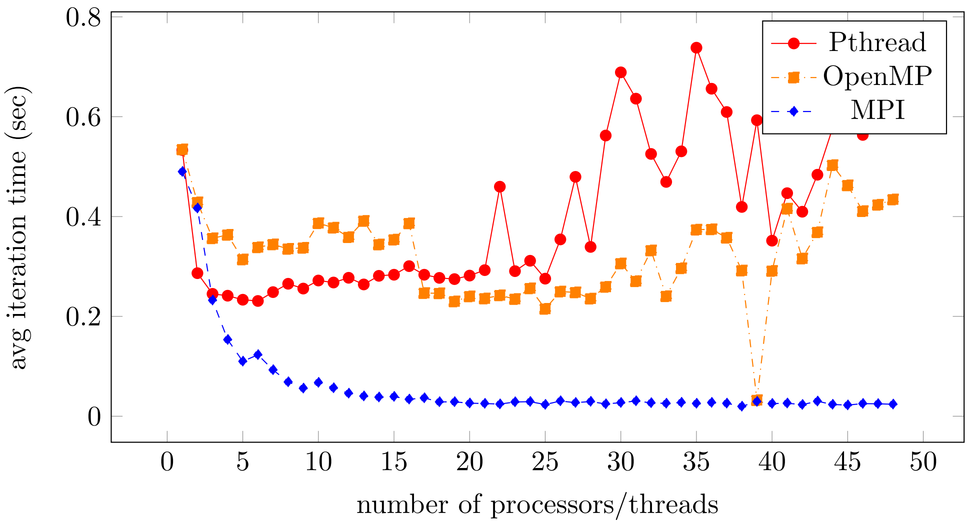


Figure 9: The time vs. number of processors graph with grid size = 10000^2

	1	2	3	4	5	6	7	8	9	10	11	12
Pthread, 10000^2	0.5332	0.2865	0.2451	0.2414	0.2334	0.2309	0.2486	0.2653	0.2557	0.2718	0.2678	0.2773
MPI, 10000^2	0.4900	0.4168	0.2324	0.1534	0.1099	0.1231	0.0928	0.0687	0.0560	0.0675	0.0567	0.0458
OpenMP, 10000^2	0.5345	0.4284	0.3565	0.3634	0.3140	0.3384	0.3442	0.3353	0.3373	0.3867	0.3777	0.3587

	13	14	15	16	17	18	19	20	21	22	23	24
Pthread, 10000^2	0.2640	0.2812	0.2834	0.3006	0.2834	0.2771	0.2747	0.2817	0.2927	0.4598	0.2906	0.3114
MPI, 10000^2	0.0403	0.0382	0.0393	0.0340	0.0366	0.0287	0.0288	0.0261	0.0254	0.0243	0.0285	0.0291
OpenMP, 10000^2	0.3912	0.3440	0.3537	0.3864	0.2467	0.2463	0.2300	0.2400	0.2358	0.2422	0.2347	0.2561

	25	26	27	28	29	30	31	32	33	34	35	36
Pthread, 10000^2	0.2756	0.3542	0.4795	0.3392	0.5623	0.6889	0.6361	0.5254	0.4694	0.5306	0.7382	0.6561
MPI, 10000^2	0.0234	0.0305	0.0272	0.0295	0.0245	0.0272	0.0305	0.0267	0.0257	0.0275	0.0256	0.0274
OpenMP, 10000^2	0.2151	0.2497	0.2481	0.2356	0.2591	0.3060	0.2705	0.3320	0.2404	0.2964	0.3737	0.3744

	37	38	39	40	41	42	43	44	45	46	47	48
Pthread, 1000^2	0.6094	0.4192	0.5930	0.3517	0.4468	0.4097	0.4837	0.5766	0.5893	0.5635	0.5857	0.6088
MPI, 10000^2	0.0258	0.0197	0.0293	0.0252	0.0261	0.0232	0.0300	0.0235	0.0223	0.0252	0.0249	0.0240
OpenMP, 10000^2	0.3575	0.2918	0.0321	0.2910	0.4158	0.3159	0.3685	0.5029	0.4625	0.4108	0.4236	0.4342

From the figure above, it is clearly shown that, MPI program reduces the runtime shapely when additional processors are introduced in a large scale problem. However the pthread and OpenMP versions improve the efficiency at first and 0 are vibrating frequently and does not have the sign to converge in runtime. The reason behind is not because the bugs in pthread program but memory caching mechanism, which will be discussed later.

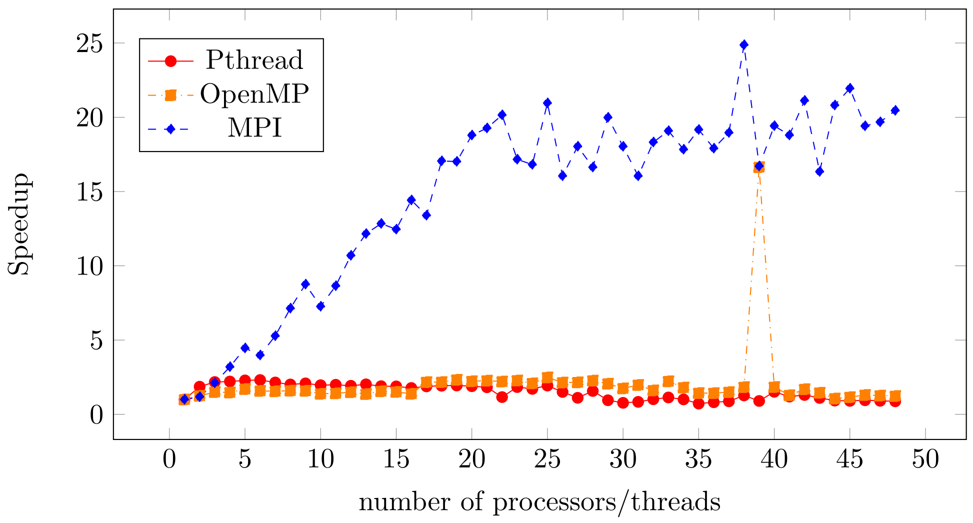
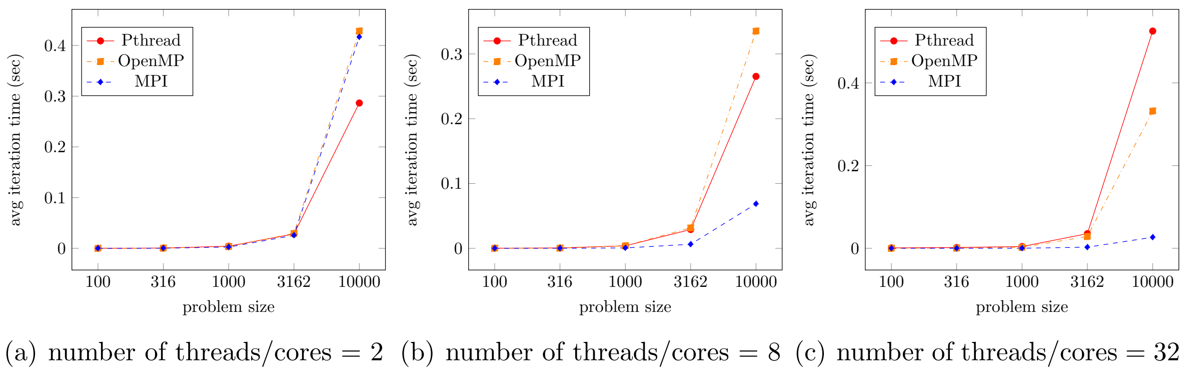


Figure 10: Speedup of three implementaions under large scale problem

From the figure above, the speedup factors of OpenMP and pthread lifted when few processors are introduced and are extremely low when number of processors is large. While the curve of MPI is uplifting for a period and then keeps the index almost the same when the number increases continuously.

3.3 Performance Under Different Problem Sizes



(a) number of threads/cores = 2 (b) number of threads/cores = 8 (c) number of threads/cores = 32

Problem size	100	316	1000	3162	10000
Pthread, 2 threads	0.0002	0.0006	0.0043	0.0283	0.2865
MPI, 2 cores	0.0001	0.0003	0.0025	0.0257	0.4168
OpenMP, 2 threads	0.0001	0.0005	0.0035	0.0295	0.4284
Pthread, 8 threads	0.0002	0.0006	0.0039	0.0290	0.2653
MPI, 8 cores	0.0001	0.0001	0.0006	0.0063	0.0687
OpenMP, 8 threads	0.0002	0.0004	0.0040	0.0317	0.3353
Pthread, 32 threads	0.0010	0.0019	0.0043	0.0354	0.5254
MPI, 32 cores	0.0001	0.0000	0.0003	0.0027	0.0267
OpenMP, 32 threads	0.0002	0.0005	0.0025	0.0286	0.3320

From the figures and data above, the programs are running more slowly when increasing the problem size, this is without any suprise. And the result is similar to that in the last section, these programs are similar in reaction to increment in problem size, and when the a lot of processors are introduced MPI make the slowest growth in runtime when problem size grows.

3.4 Performance of GPU Offloading (CUDA)

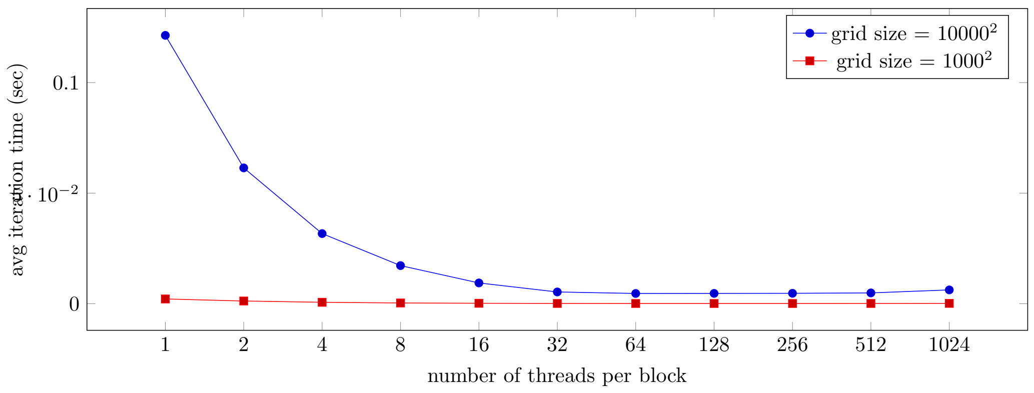
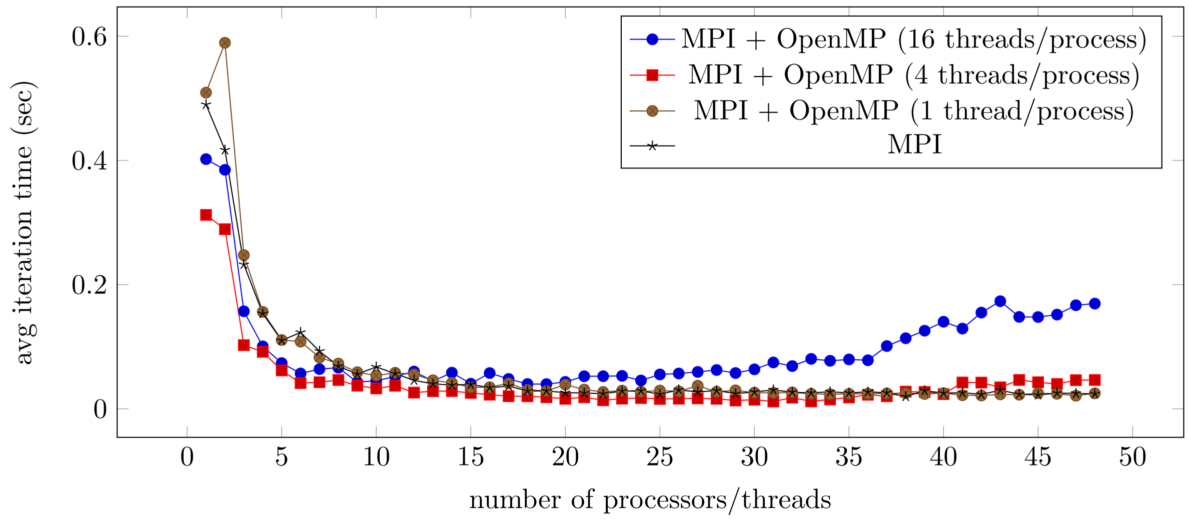


Figure 11: The runtime of CUDA program with respect to the numebr of threads in one block

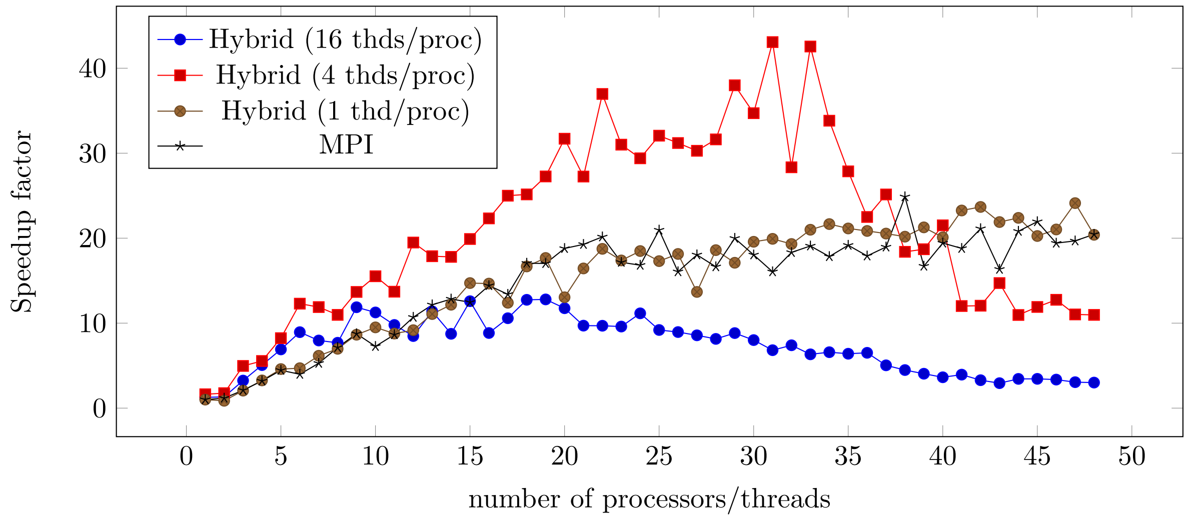
Threads per block	1	2	4	8	16	32	64	128	256	512	1024
grid size = 1000^2	0.0021	0.0012	0.0006	0.0003	0.0002	0.0001	0.0001	0.0001	0.0001	0.0001	0.0001
grid size = 10000^2	0.1214	0.0614	0.0317	0.0172	0.0094	0.0053	0.0046	0.0047	0.0047	0.0049	0.0062

CUDA program raises the highest level of parallelization, where every threads are basically parallelized to each other. For each CUDA block, the calculation is independent but not guaranteed to be parallelized, and for each CUDA thread in the same block, the calculation is parallelized to each other. This design tells us that when the numebr of threads in one block is small, the efficiency will not performed out entirely, and when increasing the number of threads in one block, the performance will increase, but with limit. When the number of therads increases to a certain level, their contension in resources will drag the performance down in certain level. This can be clearly seen in the figure.

3.5 Comparison between MPI w&wo OpenMP(Bonus)



(a)

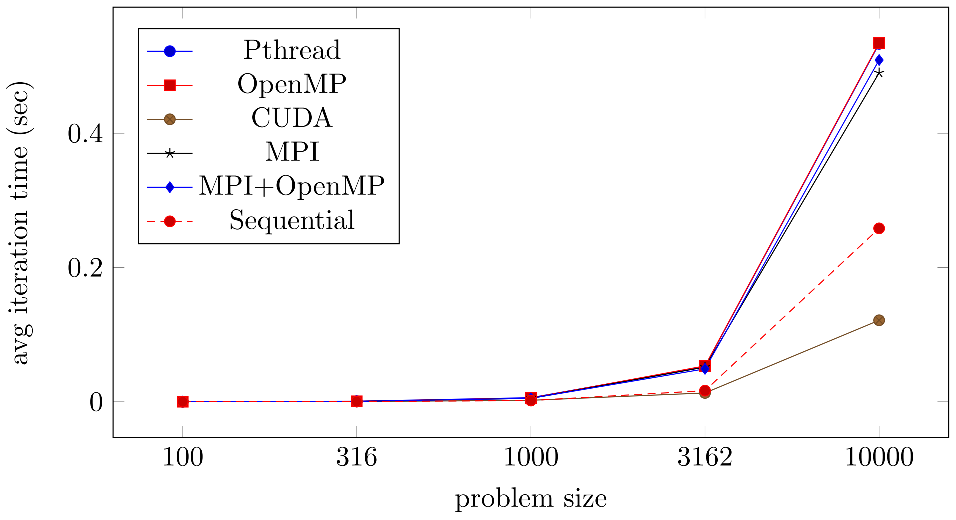


(b)

	1	2	3	4	5	6	7	8	9	10	11	12
1 thread/proc	0.5092	0.5894	0.2475	0.1561	0.1108	0.1083	0.0827	0.0729	0.0589	0.0536	0.0580	0.0555
4 threads/proc	0.3122	0.2893	0.1026	0.0918	0.0618	0.0414	0.0428	0.0464	0.0372	0.0328	0.0372	0.0261
16 threads/proc	0.4021	0.3849	0.1571	0.1006	0.0737	0.0570	0.0640	0.0662	0.0429	0.0452	0.0520	0.0600
	13	14	15	16	17	18	19	20	21	22	23	24
1 thread/proc	0.0460	0.0418	0.0346	0.0347	0.0411	0.0305	0.0288	0.0390	0.0310	0.0272	0.0293	0.0275
4 threads/proc	0.0285	0.0286	0.0256	0.0228	0.0204	0.0202	0.0187	0.0161	0.0187	0.0138	0.0164	0.0173
16 threads/proc	0.0447	0.0582	0.0405	0.0576	0.0481	0.0400	0.0398	0.0433	0.0525	0.0525	0.0530	0.0456
	25	26	27	28	29	30	31	32	33	34	35	36
1 thread/proc	0.0294	0.0281	0.0372	0.0274	0.0298	0.0260	0.0256	0.0264	0.0243	0.0235	0.0241	0.0244
4 threads/proc	0.0159	0.0163	0.0168	0.0161	0.0134	0.0147	0.0118	0.0180	0.0120	0.0151	0.0183	0.0226
16 threads/proc	0.0554	0.0569	0.0594	0.0625	0.0577	0.0636	0.0747	0.0688	0.0804	0.0774	0.0796	0.0782
	37	38	39	40	41	42	43	44	45	46	47	48
1 thread/proc	0.0248	0.0252	0.0239	0.0253	0.0219	0.0215	0.0233	0.0227	0.0251	0.0242	0.0211	0.0250
4 threads/proc	0.0203	0.0277	0.0272	0.0237	0.0424	0.0422	0.0346	0.0465	0.0428	0.0399	0.0462	0.0464
16 threads/proc	0.1011	0.1137	0.1258	0.1402	0.1292	0.1550	0.1733	0.1480	0.1478	0.1517	0.1668	0.1694

From the comparison between MPI program and different configuration of MPI+OpenMP hybrid program, it is seen that the resources are indeed used when more threads are introduced. In essence, the non-used CPU resources and other resource fragment in calculation are utilized by introducing light weighted threads. However, threads are not increasing the performance all the time, when there are 16 threads in on MPI process, the performance will go even worse and take more time to complete all the calculation.

3.6 Comparison on Sequential & Parallel Implementaions



Problem size	100	316	1000	3162	10000
Pthread	0.0002	0.0007	0.0060	0.0524	0.5332
OpenMP	0.0001	0.0006	0.0054	0.0534	0.5345
CUDA	0.0000	0.0002	0.0021	0.0131	0.1214
MPI	0.0001	0.0005	0.0050	0.0515	0.4900
MPI + OpenMP	0.0001	0.0005	0.0050	0.0490	0.5092
Sequential	0.0000	0.0002	0.0018	0.0165	0.2583

Theoretically, the parallelized program will perform just in the way of a sequential program, and which is the basis of efficiency parallelism. In the figure above, it is shown that most program are running just a little bit slower than the sequential one, in my personal perspective, these differences come from the communication overhead of parallelization. However, it is seen that cuda program runs faster than the sequential version, this is because cuda threads are not totally sequential ones, which comes from the design of this program.

3.7 DicussionAn Idea of Improving The Pthread Program

In the comparison above, it is seen that pthread program is running slowly most of the times. One reason is that each vertex are not computed with much effort however the space range quite widely in memory (10000×10000 vertices takes amost 800MB of the main memory). To keep the fairness of computation, the locality of memory is ignored in this process, so one idea to improve its performance is assigning continuous jobs to each thread. I come up with this idea and retest the program on the largest data, and the result greatly satisfied my guess: the test data is too large and there indeed is some implicit memory caching happening!

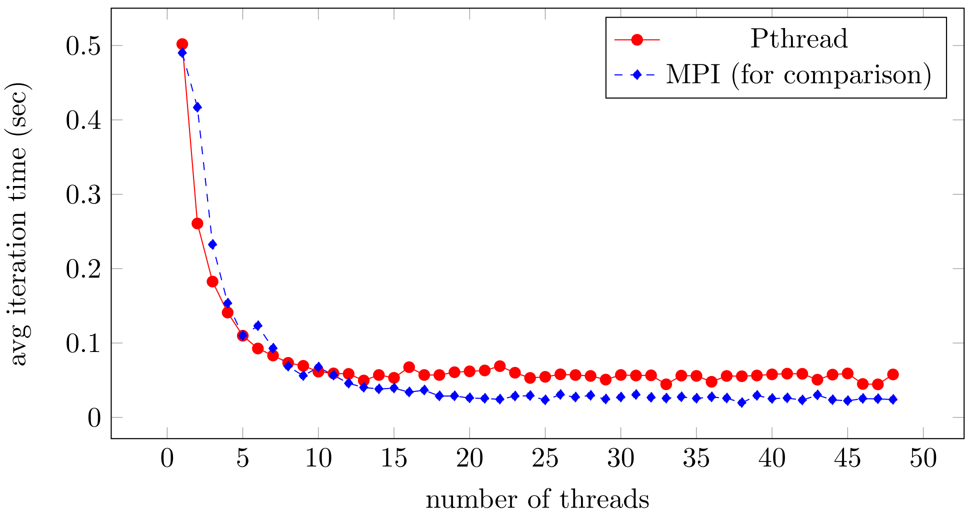


Figure 12: time v.s. number of threads in modified pthread program, grid size = 10000^2

Unfortunately, due to time, when I discovered this mechanism, I finished most of the contents and did not have enough time to re-conduct all the experiments. However, even the previous experiment would still come up with important conclusion: that is, even the implement idea is the same, different API requires different careful examination in threotical performance.

4 Conclusion

In this project, I see the parallelizing improvement in $O(N)$ algorithm for the first time. In such problem, partitioning of problem should be done carefully, especially in message pass models.

And when handling large memory, it is another great importance to keep the spcae locality. In most situations, the cuda runs the fastest because of its high degree of parallelizing (using the most number of computation cores). And for the first time in the four projects, sequential program runs faster than many of the parallel programs when set other parallel program to run on one processor only, I believe this is caused by the program's nature of having low density of computation per memory unit. At last the hybrid method of using MPI and OpenMP together indeed speed up the calculation in some degree but overusing this technic might still drag the overall performance down.

Appendix A. How to Compile & Run the Program

Compile:

The compilation can be done by calling `make`: If you want to compile all implementaions with gui:

```
$make gui
```

If you want to compile all implementaions without gui:

```
$make no_gui
```

If compiling a specific implementation:

1. CUDA (with GUI): `$make cudag`
2. CUDA (without GUI): `$make cuda`
3. MPI (with GUI): `$make mpig`
4. MPI (without GUI): `$make mpi`
5. Pthread (with GUI): `$make pthreadg`
6. Pthread (without GUI): `$make pthread`
7. OpenMP (with GUI): `$make openmpg`
8. OpenMP (without GUI): `$make openmp`
9. Sequential (with GUI): `$make seqg`
10. Sequential (without GUI): `$make seq`
11. MPI + OpenMP (with GUI): `$make bonusg`
12. MPI + OpenMP (without GUI): `$make bonus`

Run:

For all the compiled executables above, running directly will start simulation with $N_{row} = N_{col} = 800$, $iter = 100$.

Additionally, you can use the following commands to run with custom options:

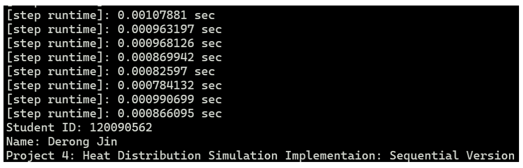
Run with GUI

CUDA	`./cudag <# row> [<# col>] <# iters> <# thds/proc>`
MPI	`mpirun -np <# procs> ./mpig <# row> [<# col>] <# iters>`
Pthread	`./pthreadg <# row> [<# col>] <# iters> <# thds>`
OpenMP	`./openmpg <# row> [<# col>] <# iters> <# thds>`
MPI + OpenMP	`mpirun -np <# procs> ./bonusg <# row> [<# col>] <# iters> <# thds/proc>`
Sequential	`./seqg <# row> [<# col>] <# iters>`

Run without GUI

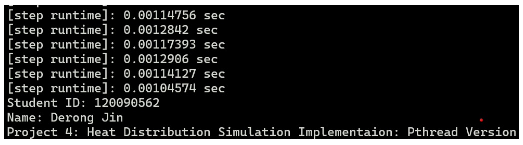
CUDA	`./cuda <# row> [<# col>] <# iters> <# thds/proc>`
MPI	`mpirun -np <# procs> ./mpi <# row> [<# col>] <# iters>`
Pthread	`./pthread <# row> [<# col>] <# iters> <# thds>`
OpenMP	`./openmp <# row> [<# col>] <# iters> <# thds>`
MPI + OpenMP	`mpirun -np <# procs> ./bonus <# row> [<# col>] <# iters> <# thds/proc>`
Sequential	`./seq <# row> [<# col>] <# iters>`

Command Line Outputs:



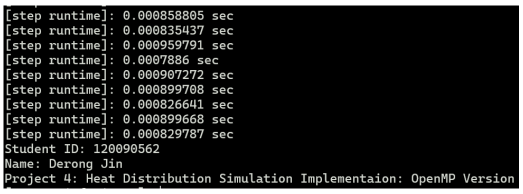
```
[step runtime]: 0.00107881 sec
[step runtime]: 0.000963197 sec
[step runtime]: 0.000968126 sec
[step runtime]: 0.000869942 sec
[step runtime]: 0.00082597 sec
[step runtime]: 0.000784132 sec
[step runtime]: 0.000990699 sec
[step runtime]: 0.000866095 sec
Student ID: 120090562
Name: Derong Jin
Project 4: Heat Distribution Simulation Implementaion: Sequential Version
```

(a) Sequential Version



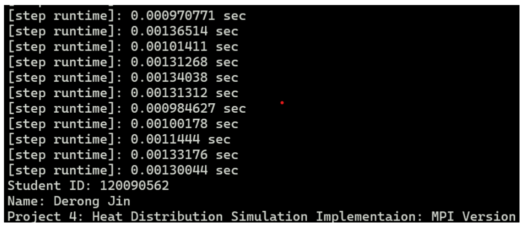
```
[step runtime]: 0.00114756 sec
[step runtime]: 0.0012842 sec
[step runtime]: 0.00117393 sec
[step runtime]: 0.0012906 sec
[step runtime]: 0.00114127 sec
[step runtime]: 0.00104574 sec
Student ID: 120090562
Name: Derong Jin
Project 4: Heat Distribution Simulation Implementaion: Pthread Version
```

(b) Pthread Version



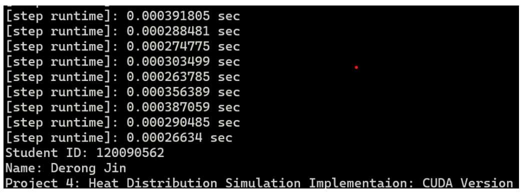
```
[step runtime]: 0.000858805 sec
[step runtime]: 0.000835437 sec
[step runtime]: 0.000959791 sec
[step runtime]: 0.0007886 sec
[step runtime]: 0.000907272 sec
[step runtime]: 0.000899708 sec
[step runtime]: 0.000826641 sec
[step runtime]: 0.000899668 sec
[step runtime]: 0.000829787 sec
Student ID: 120090562
Name: Derong Jin
Project 4: Heat Distribution Simulation Implementaion: OpenMP Version
```

(c) OpenMP Version



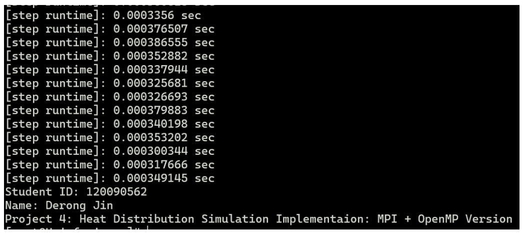
```
[step runtime]: 0.000970771 sec
[step runtime]: 0.00136514 sec
[step runtime]: 0.00101411 sec
[step runtime]: 0.00131268 sec
[step runtime]: 0.00134038 sec
[step runtime]: 0.00131312 sec
[step runtime]: 0.000984627 sec
[step runtime]: 0.00100178 sec
[step runtime]: 0.0011444 sec
[step runtime]: 0.00133176 sec
[step runtime]: 0.00130044 sec
Student ID: 120090562
Name: Derong Jin
Project 4: Heat Distribution Simulation Implementaion: MPI Version
```

(d) MPI Version



```
[step runtime]: 0.000391805 sec
[step runtime]: 0.000288481 sec
[step runtime]: 0.000274775 sec
[step runtime]: 0.000303499 sec
[step runtime]: 0.000263785 sec
[step runtime]: 0.000356389 sec
[step runtime]: 0.000387059 sec
[step runtime]: 0.000290485 sec
[step runtime]: 0.00026634 sec
Student ID: 120090562
Name: Derong Jin
Project 4: Heat Distribution Simulation Implementaion: CUDA Version
```

(e) CUDA Version



```
[step runtime]: 0.0003356 sec
[step runtime]: 0.000376507 sec
[step runtime]: 0.000386555 sec
[step runtime]: 0.000352882 sec
[step runtime]: 0.000337944 sec
[step runtime]: 0.000325681 sec
[step runtime]: 0.000326693 sec
[step runtime]: 0.000379883 sec
[step runtime]: 0.000340198 sec
[step runtime]: 0.000353202 sec
[step runtime]: 0.000300344 sec
[step runtime]: 0.000317666 sec
[step runtime]: 0.000349145 sec
Student ID: 120090562
Name: Derong Jin
Project 4: Heat Distribution Simulation Implementaion: MPI + OpenMP Version
```

(f) MPI + OpenMP Version