Parallel Programming Homework 3

Mandelbrot Set

CS 101062337 Hung-Jin Lin

1. Design

(a) What are the differences between the implementation of six versions?

There are three different multitasking API strategies, OpenMP, MPI, and hybrid of both; and also two schedule strategies, static and dynamic.

First, take a look at OpenMP implementation uses shared memory multithreading model, so it's easy to share data between threads and it does only need some pragma information for compiler and the program then comes parallel. And thus, in this project, I use OpenMP when it comes to calculating whether a single complex number in Mandelbrot Set; in the original sequential implementation, it's a double nested for-loop structure, so just use pragma *collapse(2)* and some hints then can expand the loops into parallel works. When it comes to two scheduling policies, the difference between static and dynamic versions is using two built-in scheduling pragma parameter, static and dynamic, to make it reach the goal.

MPI is another strategy to parallel program between distributed machines, comparing to OpenMP version, this implementation needs to communicate with each processing nodes. In static version, I distribute the total jobs into every process and then make them do their own computation job, finally collect all results from each process at the last step. While in dynamic version, I choose centralized working pool model, so there must be master and slaves for this implementation. In this model, master process gives out the jobs and the slaves request for next job as its previous job done. However there's a problem in dynamic version, when the program only given one process to execute, the program cannot create roles of master and working slaves, so it will degenerate into sequential implementation, the master process itself has to take all the jobs once distributed to its slaves.

The third multiprogramming strategy, hybrid version, use both OpenMP and MPI. Both the static and dynamic implementations are overall very close to MPI implementations, the only difference is to add OpenMP compiler pragma onto computation part which have them speedup and finish each single work as fast as possible.

(b) How to partition the task?

The partition in OpenMP and MPI versions are a little differences. Partition for static version in OpenMP is to divide the jobs into small size, for example, make ten jobs into a job bundle, then use round-robin pattern to distribute the job bundle into workers. Whereas inside MPI static version, I choose a simple way, every process need to calculate the job number that are `distributed` for them, that means, P_1 may respond to compute the job number from 0~10, and P_2 for jobs numbered 10~20, though it isn't a best strategy for load balancing, it can reduce the complexity in collecting results.

The partition for dynamic version are similar in OpenMP, MPI, and hybrid implements. Pack the jobs into job `bundle`, and the pack size cannot be too small or there will be many communication costs during jobs distributions and result receiving. And the bundle size may reduce as the program executes, since that as the workers finish more jobs, the job queue will be much smaller which means some worker may be idle. The solution is to dynamic change the bundle size and make it split the jobs more balance and hence reach load balancing between workers.

(c) What technique used to reduce execution time and increase scalability?

In OpenMP, the only different between multithreading and original sequential version is two additional lines of OpenMP pragmas. And in the part of computing Mandelbrot Set are double nested loop structure, so with clause of *collapse(2)* can make OpenMP parallel the jobs in `points` level. With this feature, it may runs more balance and smooth the time difference. However, there's still problem while using OpenMP scheduler clause, the chunk size for scheduler does very important and will make huge variance, so it need experiments and make some decision to optimize the best size for load balancing.

In MPI program, communication cost is a difficult subject comparing to shared memory model. Let's first look inside the MPI static scheduling version, in this version, I choose to split the jobs into each process as equally as possible and at the final step we need only a MPI gather function then the entire job's done. Though times of communication and the size of communication message need some tradeoff, this implantation has a great scalability on multiprocessing and it can work on arbitrary number of processes. Besides, the MPI dynamic scheduling can also run on *n* processes (n is arbitrary integer that is greater than 1.) Although dynamic version still can works on only one process, it breaks the master-slave relation and we won't take in discussion. As more processes added means it get more resources, slaves or workers, to finish the jobs, and thus this implementation is also easily scalable.

(d) Other efforts in program

It's about software engineering. I choose to follow the DRY (don't repeat yourself) principle. There are too much duplicate code among six versions, if left them separate in many places, it'll be a

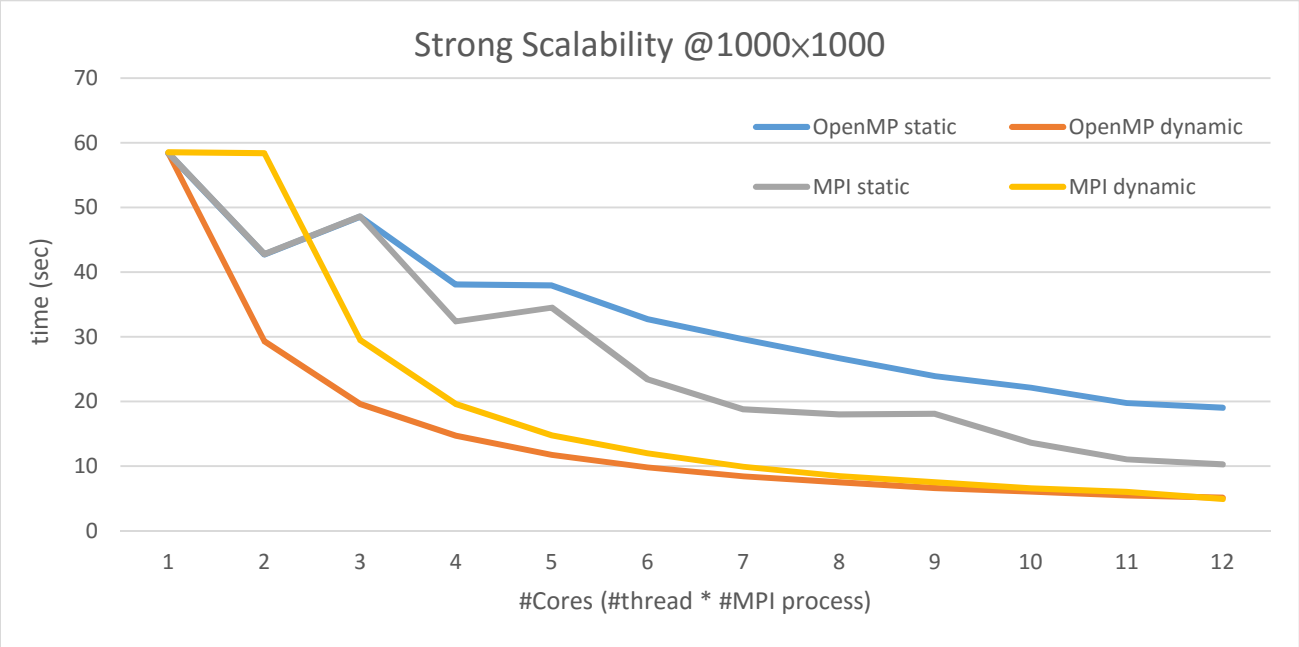
hard time to modify when the similar function calls need some changes. So I made a great effort on refactoring the code into different headers and implementations files, and now with a Makefile to control the compiling flags and dependencies, the effort on any edition in program can be minimize and make it easy to debug and feature development, which means it's scalable in developing.

2. Performance analysis

Fix coordinate's range from (-2,-2) to (2, 2).

(a) Scalability chart

i. Strong scalability – scalability to number of cores
(Problem size is fixed)



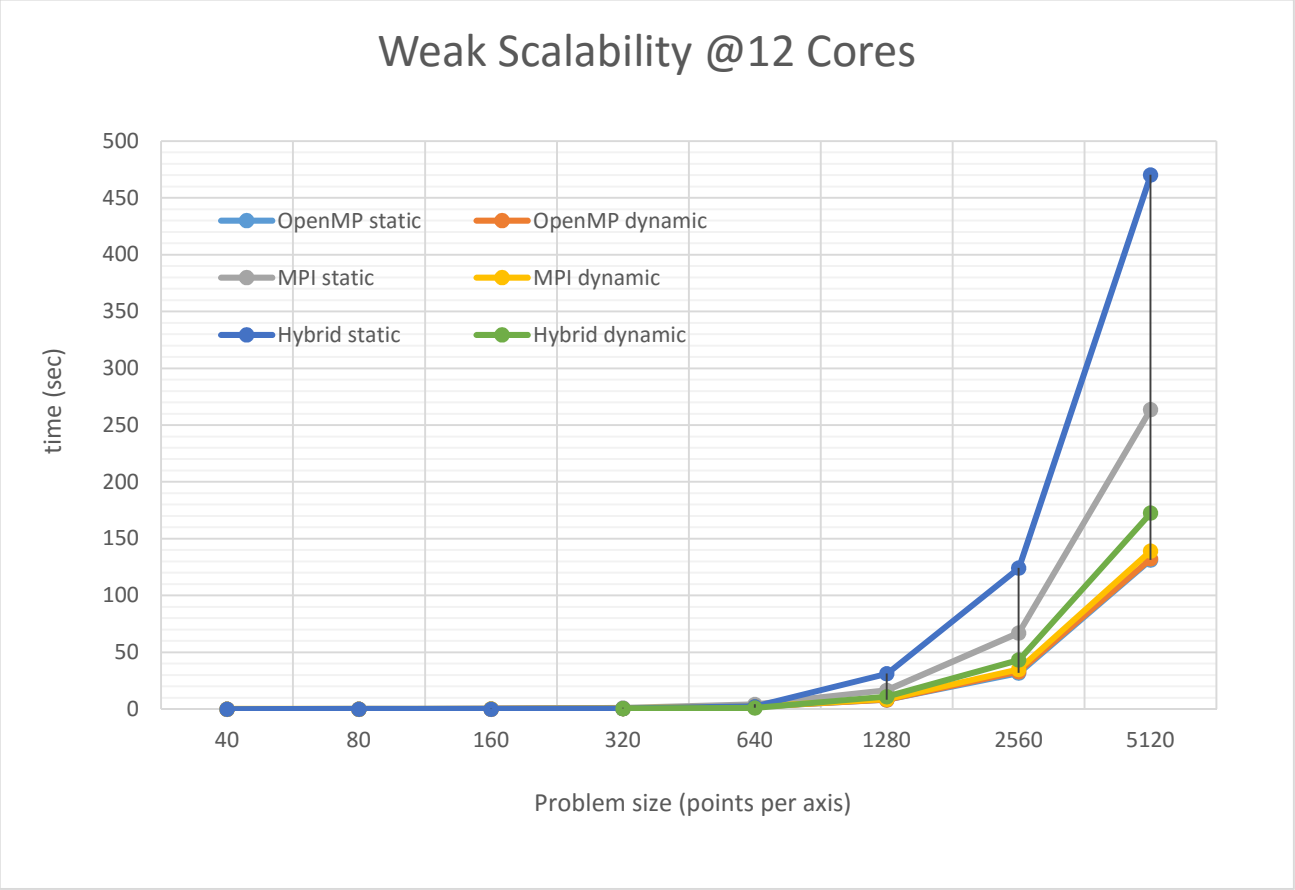
The testing environment is: given each program from one to twelve `cores` to calculate Mandelbrot Set problem in square of one thousand points.

- From the graph we can observe that the curves of dynamic scheduling versions are smoother than static versions, and also much quicker in execution time.
  - Dynamic scheduling is quicker than static version because it can balance computing load when distributing jobs to each computing units. The jobs have different `weight`, dynamic scheduler can make heavy and light jobs distributed instead of gather in certain computing units like static scheduling does.
  - So we know that static version always be bounded by the slowest computations, and hence the statistic is worse than others.

- Another problem is that curves in static scheduling are uneven. We have known that static version may be bounded by slowest unit, so add one more process in may not only have no help to improve the efficiency but make the partition worse, and the final result is bounded by another worse case.

- Dynamic versions' curves of OpenMP and MPI are almost overlap
  - The MPI version is master-slaves centralized working pool model, so the first statistic can be ignored which means we can move forward the curve for one column.
  - Surprisingly found that two curves are extremely similar, they are almost overlapping each other. Though MPI has to conquer the cost in inter-process communications, in my opinion, maybe one of the reasons is I let them run in same node so somehow it can reduce original high cost in inter-node-process; another reason is that OpenMP dynamic scheduler need also some effort to arrange the jobs. They both have some cost besides the computation part and finally they use similar time to fulfill the work.

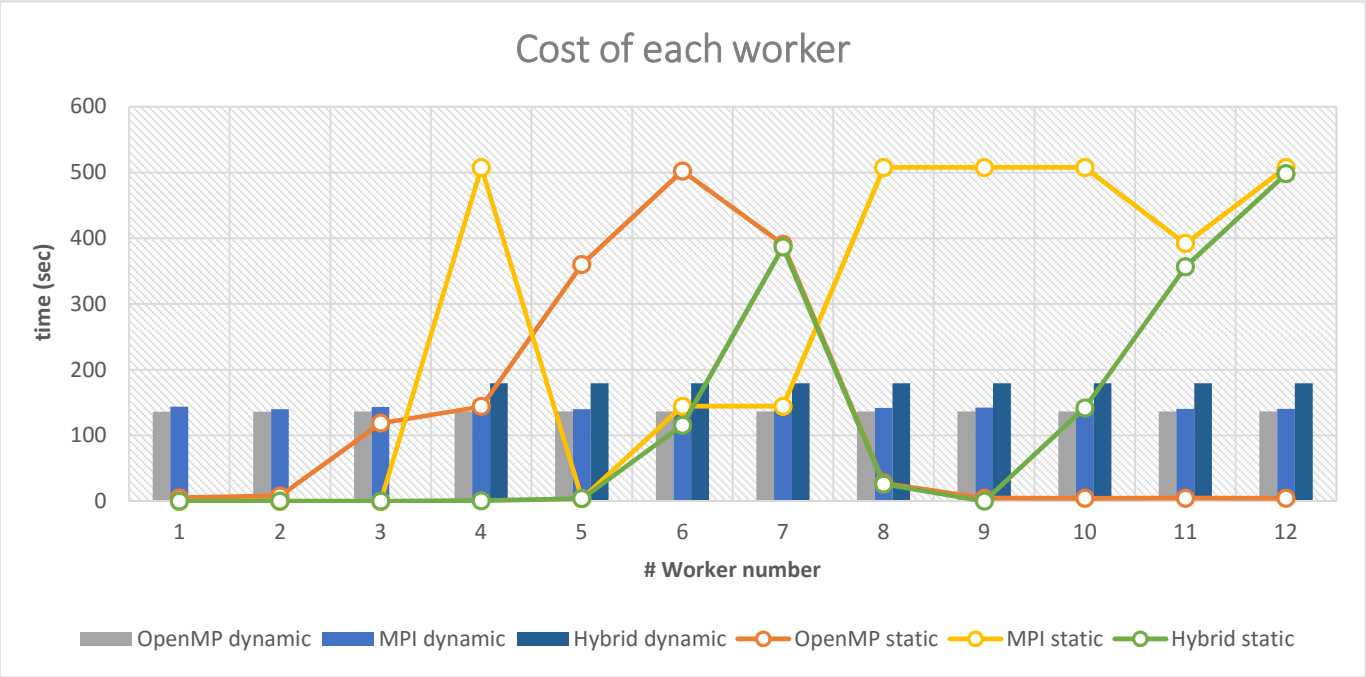
ii. Weak scalability – scalability to problem size (# cores is fixed)



In this experiment, six implementations have the same cores, 12 computing units, to compute the Mandelbrot Set in different problem size. In hybrid version, I give them *4 processes and each process has 3 threads*.

- We can observe that static version is slower than its dynamic version once again and when the problem size grows, the larger the difference between them.
- OpenMP static is faster than MPI static version, MPI static is faster than hybrid static version.
  - OpenMP is shared memory model, so when it comes to comparing with the MPI version or MPI and OpenMP hybrid version, pure OpenMP implementation can outplay them as the problem scale up, it's more obvious.
  - Comparing to OpenMP version, communication costs are more evident when problem size get larger in MPI version.
  - Hybrid static version get a tragedy performance in this experiment, because it not only has communication cost but also has to split the jobs in one process into small packages for inter-threads. What's worse, the inter-threads also use static, which means the time is bounded by single inter-thread!
- Three dynamic versions are similar, hybrid version use a little more time.
  - OpenMP dynamic performs the best due to its shared memory strategy and load balancing scheduling.
  - MPI dynamic also has a nice performance, it's very close to the pure OpenMP dynamic version. The different is the memory model that make MPI version has to compromise, spending bonus effort on message passing comparing to direct memory access in shared memory model.
  - Another factor between MPI and OpenMP is that though both MPI dynamic and OpenMP dynamic both use centralized working pool model, in MPI implementation, its structure is more complicated then OpenMP and has to handle the message blocking cost and bottlenecking on master process in communications.
  - Hybrid dynamic version get a great relief from hybrid static version. We can take it as one that use dynamic scheduling to parallel the inter jobs of MPI dynamic version so it can performs even much better then MPI dynamic does.

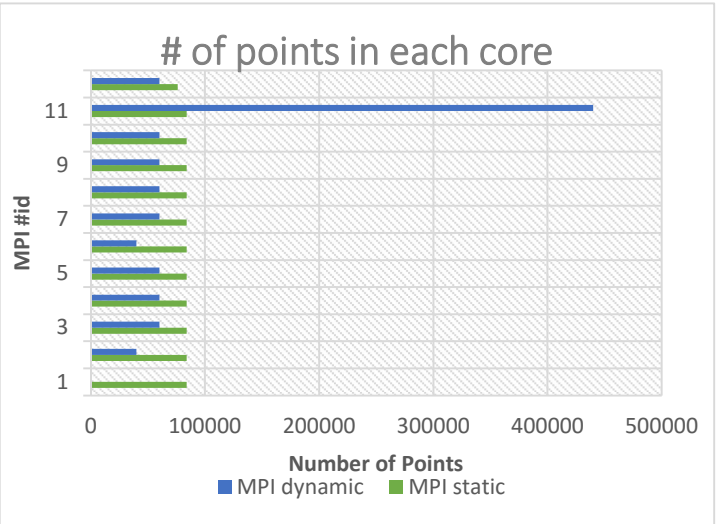
(b) Load balance chart



In this experiment, each version have twelve cores and compute with the problem size = 5120. In the graph, the bar charts are for three dynamic versions; the line charts are for another three static versions.

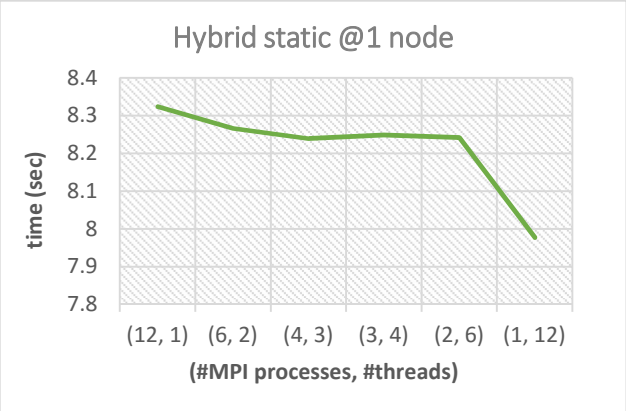
From the graph, we can conclude that work loading in the dynamic versions are almost equally distributed to each nodes, and thus they have similar finishing time. On the contrary, static version are greatly uneven, the jobs size distributed to each node are fixed, however, the finishing time of each node are not similar.

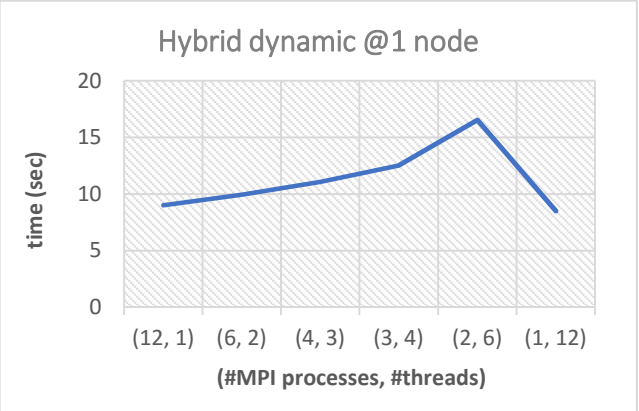
The left graph is the jobs distribution during the scheduling works. It makes static and dynamic version runs the problem size of one thousand with twelve cores.



This use the implementations of MPI as example, because it's much easier to collect the running data. And one thing should be noticed, *core#1 in dynamic is master process who does nothing about computations but as a role of commander*. Jobs are evenly distributed in static version; however uneven in dynamic version as we discussed above.

(c) Other experiments





In this experiment, make both hybrid version run with problem size = 1280, and use up to 12 cores on single node. In static version, it's obviously that it has better to use multithreading in single node rather than multiprocessing due to the difference in message passing.

While in dynamic version, it seems like a non-sense curve, however, we have to remind that dynamic is a master-slaves processing model, so the number of process used in computing is *process per node -1*. That means (12 processes * 1 threads)→11x1 cores, (6 processes * 2 threads)→5x2 cores, (4, 3)→3x3 cores, (3, 4)→2x4 cores, (2, 6)→1x6 cores, and special case (1, 12)→1x12 cores. So in each case are [11, 10, 9, 8, 6, 12] cores, and the last combination has 12 cores and thus run fastest.

3. Experience

(a) What have you learned from this assignment?

During programming MPI program, I'm thinking about how to send structured data between processes, after searching the MPI documents and knowing that MPI support sending structured data by some pre-defined initialization before starting communications. Nevertheless, I don't use those functions instead I find a simple implementations can get the same achievement: use an array to store all needed data! The returned array should contains a list of color results and now add some additional information such as current job's id for master to recognize and decide which position to display. A most easy implementation is reserved first place to store the job id and the later space for real result content.

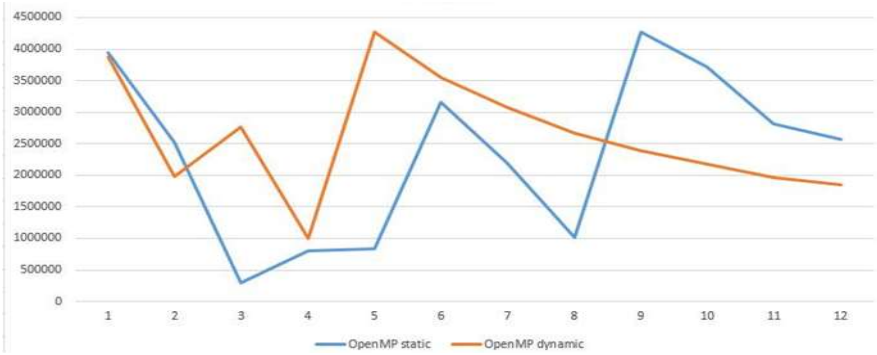
During experiment, it's inconvenient to submit job to PBS for result: first we need a job shell script and then modified the numbers in script, and finally use *qsub* to submit the job. I'd like to say it should be done automatically instead of manually, so I write a script to fulfill the testing task. Additionally, while the job queue is full, the script will automatically handle that situation by waiting timeout and then retry to submit the jobs until successfully get response from PBS. Moreover, with help of Linux command, *nohup*, I can trigger the script in background and log out from server, and it'll

be detached and still run; then I can go to do other tasks and just wait for the experiment results like a boss.

```
(16:55:49) ± [s101062337@pp01] ~/hw3 (master U:10 ?:11 x)
→ ./src/jobs.py
Total jobs: 6
Testcase#0 57812.

Testcase#1 57813.

Testcase#2 57814.

Testcase#3 57815.

Testcase#4 57816.

Testcase#5 57817.
```

```
(17:51:54) ± [s101062337@pp01] ~/hw3 (master U:10 ?:11 x)
→ ll output/
total 28
-rw------- 1 s101062337 s101062337 89 Dec 12 17:43 ms_hybrid-dynamic-12x1
-rw------- 1 s101062337 s101062337 90 Dec 12 16:45 ms_hybrid-dynamic-1x1
-rw------- 1 s101062337 s101062337 92 Dec 12 17:42 ms_hybrid-dynamic-1x12
-rw------- 1 s101062337 s101062337 60 Dec 12 17:42 ms_hybrid-dynamic-2x6
-rw------- 1 s101062337 s101062337 71 Dec 12 17:43 ms_hybrid-dynamic-3x4
-rw------- 1 s101062337 s101062337 78 Dec 12 17:43 ms_hybrid-dynamic-4x3
-rw------- 1 s101062337 s101062337 83 Dec 12 17:43 ms_hybrid-dynamic-6x2
(17:51:58) ± [s101062337@pp01] ~/hw3 (master U:10 ?:11 x)
→
```

(b) What difficulty did you encounter when implementing this assignment?



I face into a bug due to my ignorance about standard C++11 library. When measuring elapsed time in multiprocessing program, it has better to use wall time counter to collect the real consuming time which I've learned from previous project, however, this time I use a CPU clock function to measure the time and finally got an extreme bizarre graph as above attached. I used to apply C++ new feature *chrono* library to measure the program performance in other projects because of its modern template type casting, high level of encapsulation and additional it can measure at nanosecond scale, and it actually support measuring system wall clock. At that time, I took the often used one which is for CPU clock measurement and got this terrible mistake, got bothered and pondered over for a period. Finally, I read over the C++ library documents and found that I'm horribly wrong…, and finnally I made a hotfix to rewrite timing code with wall time measurement function provided by OpenMP and MPI library.

(c) If you have any feedback, please write it here.

The PBS submit procedure is a little inconvenient while making experiment and want to test through all possible case.