Assignment 2 Mandelbrot Set Problem
CSC4005 Distributed and Parallel Computing

Chen Yuan 117010038
School of Science and Engineering
The Chinese University of Hong Kong, Shenzhen
117010038@link.cuhk.edu.cn

Abstract

This assignment implements a mandelbrot set problem, using sequential and parallel method respectively. There are three parallel versions, namely the MPI static method, MPI dynamic load balancing method, Pthread static method, and Pthread dynamic load balancing method. The report would present the principle, codes, as well as the comparasion and analysis of performance of different implementations.

1. Introduction

The Mandelbrot Set is the set of points in a complex plane that are quasi-stable (will increase and decrease, but not exceed some limt) . In this report, the coding aims at visualizing the Mandelbrot Set in parallel version. Both MPI and Pthread, static and dynamic versions are inplemented in C++. The visualization is implementated by X11, as shown in Figure 1.

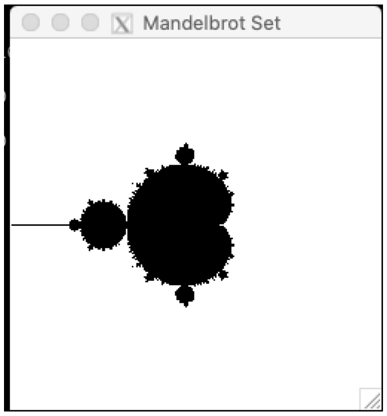


Figure 1. Mandelbrot set by X11

By default, the display is a square. In the experiment, five sizes of the display are examined – $200 \times 200, 800 \times 800, 2000 \times 2000, 6400 \times 6400, 10000 \times 10000$. Also, the parallel algorithm is evaluated under different number of processes and threads – ranging from 1 to 16.

The implentation details and performance analysis are shown below.

2. Method

For a display size with width 200, height 200, we need to calcualte each of 200×200 points to check whether it is in Mandelbrot set. Hence, the core idea for parallel distribution of the data is to maintain the structure of each column of the 200×200 matrix, and distribute the columns to different processes/threads accordingly.

In order to compute running time without drawing by X11, a glpbal variable "global_result" of size 200×200 is set up to sotre all information of the points. Then after calculation, we can use the "global_result" to identify Mandelbrot set and draw them.

2.1. MPI

Since in MPI, we have one master node and muptiple slave nodes, so in the implementation, only slave nodes do the computation – the master node is used to distribute and receive messages, as well as draw graphs, indicating that there si no meaning of single process program. So, in the experiment below, we consider the processes ranging from 2 to 17,

2.1.1 MPI static

The MPI static method identicallly distributes the columns to slaves, and do the calculation in each slave. The Master node is responsible for receiving the partial Mandelbrot set from each slave. Since the width value may not be divisable by number of slaves, the last slave is assigned with extra remainder columns. A flow chart is shown in Figure 2

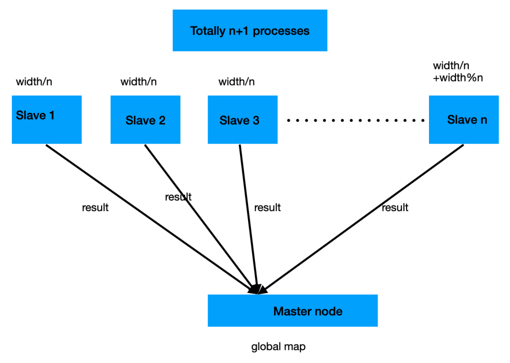
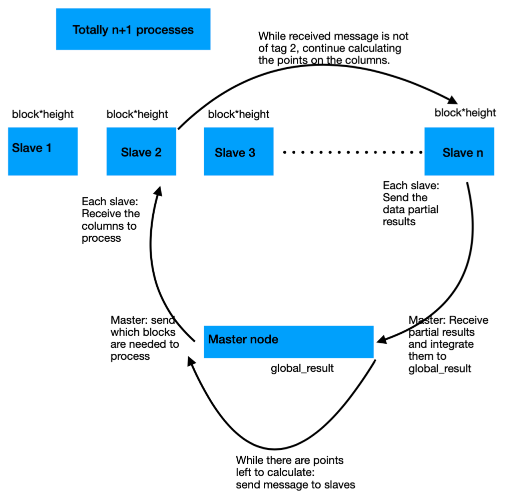


Figure 2. Flow chart for MPI static



Totally n+1 processes
While received message is not of tag 2, continue calculating the points on the columns.
block*height
Slave 1
block*height
Slave 2
block*height
Slave 3
block*height
Slave n
Each slave: Receive the columns to process
Each slave: Send the data partial results
Master: send which blocks are needed to process
Master node
global_result
Master: Receive partial results and integrate them to global_result
While there are points left to calculate: send message to slaves

Figure 3. Flow chart for MPI dynamic

2.1.2 MPI dynamic

This MPI dynamic algorithm is robust for changes in the number of column that one slave process once. First, the master distribute one block for each slave(block denotes the number of columns that each slave node process in one loop). The Master send the specific columns that slave need to process; the slave send back the results of the points on those colunms to Master. Then, while not all points are finished processing, once the master receives a message from any slave, it will repeat the former procedures – send required columns to slave, then slave send back teh result.

A trick here is that when the Master knows all points are down with calculation, it will send a message with tag 2. When the Slave reveives a message with tag 2, it will stop its own while loop on reveiving Master messages. Whenever there is a slave on work, a global count variable is increased by 1. Whenever Master reveives a message by a slave, the global count variable is decreased by 1. At last, when the total count becomes 0, the Master will stop the while loop, end clocking and turn to draw the picture.

Here, the size of block can range from 1 to width/number of processes. A flow chart is illustrated in Figure 3 .

2.2. Pthread

Since all threads can work in parallel, here we adopt the number of threads ranging from 1 to 16 in the experiment. The program is flexible for different inputs of thread numbers, display size. For easier implementation, in both pthread methods, the "struct _thread_data" contains the width, heigth, thread_id, as well as a global varibale "result". "result" is of size width×height, and the point with coordinate (i, j) is in $result[i \times height + j]$ each time a thread adds the information of a point, it adds it to the global variable result. Then there is no need to transmit data back.

2.2.1 Pthread static

Same as the method for MPI static, identically distribute the columns to each thread. The extra remainders are added to the last thread.

Height, width, number of threads and thread_id are passed to the threads for calculate the columns for each thread and calculating the properties for points. The flow chart for Pthread static method is shown in Figure 4.

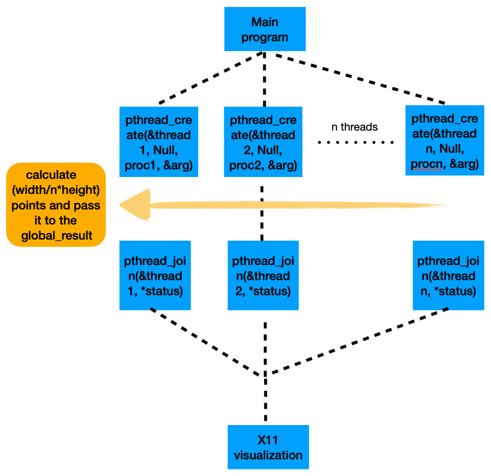

Main program
pthread_create(&thread1, Null, proc1, &arg)
pthread_create(&thread2, Null, proc2, &arg)
n threads
pthread_create(&threadn, Null, procn, &arg)
calculate (width/n*height) points and pass it to the global_result
pthread_join(&thread1, *status)
pthread_join(&thread2, *status)
pthread_join(&threadn, *status)
X11 visualization

Figure 4. Flow chart for Pthread static

2.2.2 Pthread dynamic

In the Pthread dynamic implementation, we use pthread mutex to achieve dynamic scheduling. The assignment, distribution and join operations are the same. The different part is that, there is a global_count for number of columns processed. Each time one thread will process a column, and use pthread mutex to change the global_count and fetch the global_count value as the ith column to be processed. There is a while loop in cal_func, ensuring any thread that finished one column processing could use mutex lock and process the next column. When the global_count equals the column number, there will be no more calculating in the threads and the function will return.

2.3. Command lines

All the heigh and width in the programs are 200 by default. For the MPI programs, the first two instance of main function are display width and height. For MPI static programs, there is a third instance, meaning the number of columns contained in a block, and it is 1 be default.

For pthread programs, the first instance is represented as number of threads in parallel computing; the second and third instances are display width and height.

In the compile phase, all these programs should include X11 library. The sample code for compiling and running with 8 processes or threads are listed below:(also can be found in ”run.pbs” doc in attached code)

```
#MPI static
mpic++ ./mpi_mandelbrot_static.cpp -lX11 -o
./mpi_s
mpirun -n 8 ./mpi_s 200 200
#MPI dynamic with blocksize 10
mpic++ ./mpi_mandelbrot_dynamic.cpp -lX11 -o
./mpi_d
mpirun -n 8 ./mpi_d 200 200 10
#Pthread static
g++ ./pth_mandelbrot_static.cpp -o ./pth_s -
lpthread -lX11
./pth_s 8 200 200
#Pthread dynamic
g++ ./pth_mandelbrot_dynamic.cpp -o ./pth_d -
lpthread -lX11
./pth_d 8 200 200
```

3. Experiment

The most important metric here is runtime under different circumtances. Since the X11 drawing procedure is slow, here we only consider the computation runtime, including the data transmiting, and without displaying the graph. The total results are attatched in Appendix A. One important figure that consolidates all usful information in performance is shown in Figure 5. The green horizontal line in the figure refers to the sequential implementation time (calculated by the average of 5 times).

Sections below are several analysis based on different evaluation metrics:

3.1. The number of processes or threads used in the program

As shown in Figure 5, generally, as the number of threads or processes grows, the total time for a program decreases. Meaning that as the nodes for parallel process grows, the total excucation time generally shrinks. On the figure shown above we can observe that as the number of the parallel nodes increases, the excucation time first experiences a sudden decreases for the first 1-4 nodes, then may shrink to a rather low area when it comes to 8 to 10 nodes. For parallel nodes exceeding 10, the speed up performance is may not be severe and clear.

However, there is an exception in MPI inplementation on small datasets (namely 200 × 200 figure size). as the number of processes increases, the total time for overall computation increases, and even increases much more than that for sequential implementation. The reason behind this may be that since the MPI needs send and receive implementation, the overhead for data transmitting between nodes are huge. Be-

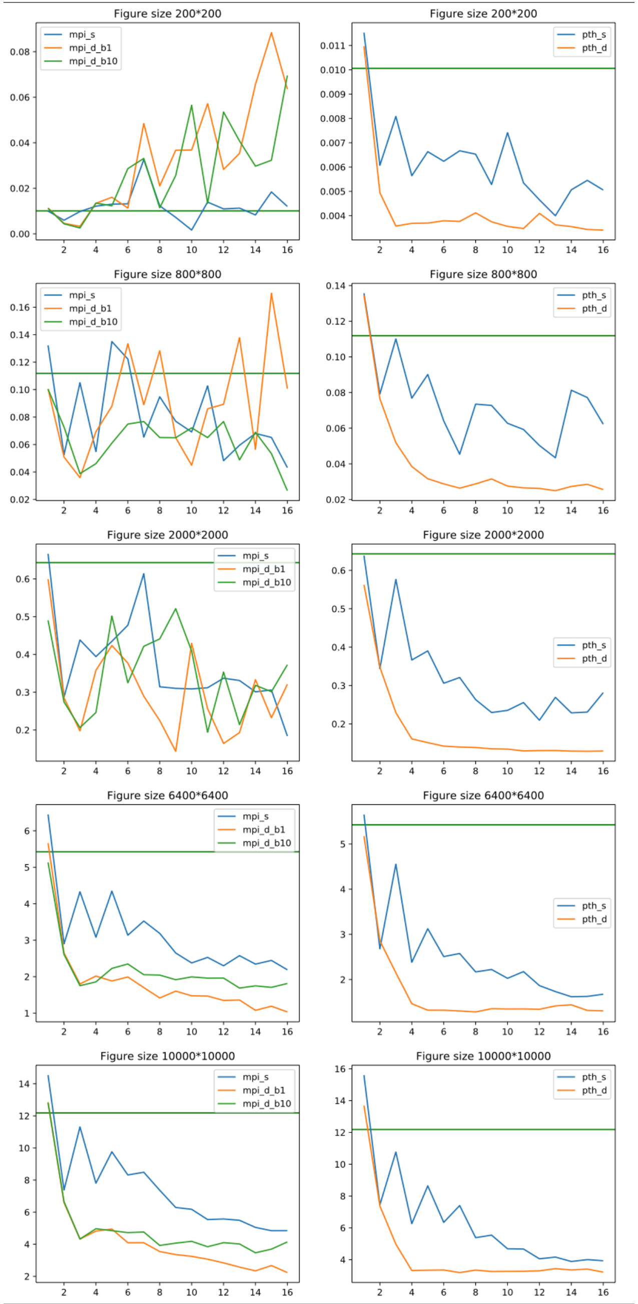


Figure 5. Performance conparasion figure.

cause in pthread implemnetation we use a global map to store all data, and it is called by reference. There is no need for huge data returning and transmission in pthread implementation. But in MPI we need the data transmission. Moreover, the mpi dynamic version with block size 10 performs better than that of block size 1, and there are fewer data sending and receiving in mpi_d_b10, so it fufills the assumption we made before.

3.2. Size of the output images

The size of output greatly influences the time for data processing. As shown in Figure 6, as the display size increases, the running time also increases. It shows a convex function trend and the increase rate in time versus display size is the biggest for sequential work, then for static MPI, then static pthread, then dynamic MPI, at last static pthread. Although the pthread method increases the least in response time than other mehtods, the curve is growing up and havs the trend of never ending.

Also with the study with different metrics, the pthread dynamic implementation performs really well.

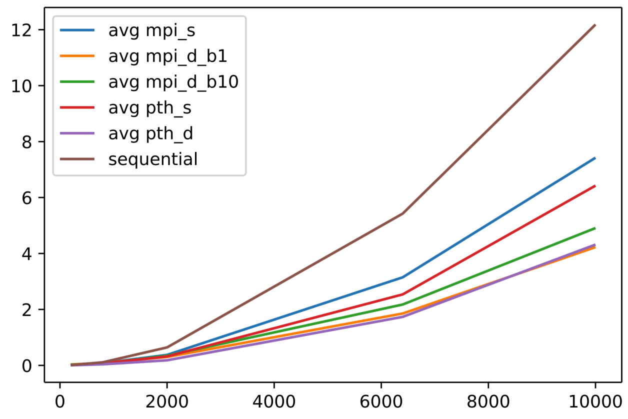


Figure 6. Average runtime for different methods with figuresize increases.

3.3. MPI static vs dynamic

As shown in the first column figures in Figure 5, for the static method, it performs well in small-size drawing data than other methods, but is not so good in other drawing methods. For larger dataset, the mpi dynamic with one column distributed at one time is always the best in performance, and the mpi static method is the worst in performance. The MPI dynamic method with more than 1 column assigned one time is somewhere in between.

Intuitively, as the MPI version changes from static to dynamic, the data transmitting overhead increases, parallel distributions increases, and the latency (not busy nodes in CPU) becomes less. It follows an FIFO spirit. Hence, as the data size(display size) increases, the data transmission overhead seems not important, but the factor of how many nodes partitipate in parallel computiong matters. So the dynamic method tend to be the fastest.

Morover, then look closely to the figure, after nodes increase to 4 or more, the dynamic with block 1 method reaches a peak and this point tend to be the lowest time costed point. After that, adding more nodes brings generally no speed up in the project. This may because that the dynamic attributes each nodes with a resonable workload, so increasing the number of nodes may not cause increase in efficiency.

3.4. MPI vs Sequential vs Pthread

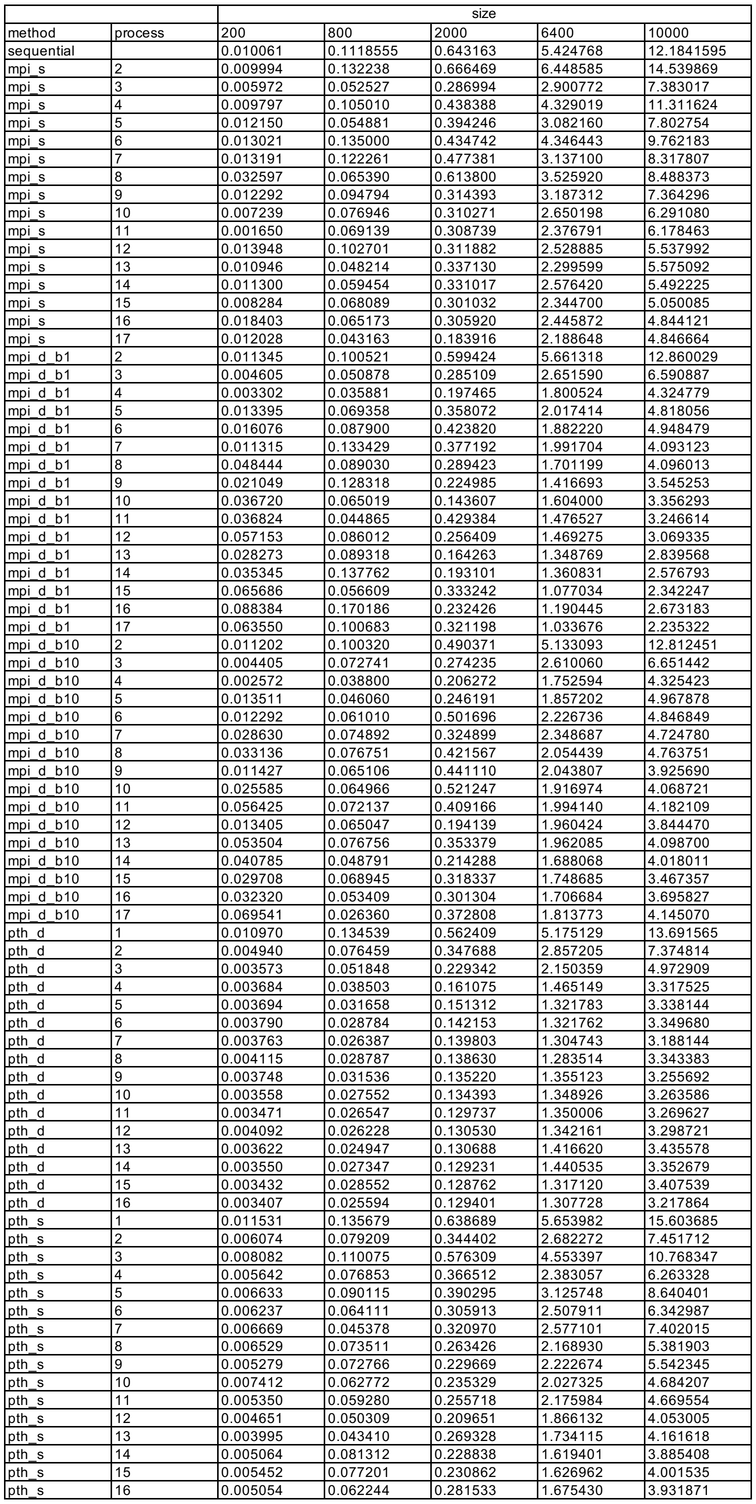
Actually Figure 5 shows clearly the 3 methods differences. The sequential method consumes the most of the time. The MPI version may be unstable (have flunctations) between some number of processors, but could speed up the problem, especially for large size display. Then the pthread method has the best speed up, and the performance is stable. It shrinks the data time to a much lower level than that of sequential version.

4. Conclusion

In this experiment, 5 parallel versions of Mandelbrot Set Problem is conducted with pthread, MPI, static, and dynamic approach. The results shows that generally as the number of threads or processes increases, the total time for the problem will decrease. After 10 nodes, the performance would not have a big speed up. And the size of figure determines generally how much time a sequential version could be used. As the size of figure increases, the time needed increases as well. Also, the best version is the pthread dynamic parallel version. The application time not only converge fast, but also is the lowest in thesr 5 parallel versions.

A. Experiment Results

The total results are listed in Figure 7. pth denotes pthread method; s denotes static, d denotes dynamic, and in "mpi_d_b1", b1 denotes blocksize 1. The data are runtime for each cases.



		size				
method	process	200	800	2000	6400	10000
sequential		0.010061	0.1118555	0.643163	5.424768	12.1841595
mpi_s	2	0.009994	0.132238	0.666469	6.448585	14.539869
mpi_s	3	0.005972	0.052527	0.286994	2.900772	7.383017
mpi_s	4	0.009797	0.105010	0.438388	4.329019	11.311624
mpi_s	5	0.012150	0.054881	0.394246	3.082160	7.802754
mpi_s	6	0.013021	0.135000	0.434742	4.346443	9.762183
mpi_s	7	0.013191	0.122261	0.477381	3.137100	8.317807
mpi_s	8	0.032597	0.065390	0.613800	3.525920	8.488373
mpi_s	9	0.012292	0.094794	0.314393	3.187312	7.364296
mpi_s	10	0.007239	0.076946	0.310271	2.650198	6.291080
mpi_s	11	0.001650	0.069139	0.308739	2.376791	6.178463
mpi_s	12	0.013948	0.102701	0.311882	2.528885	5.537992
mpi_s	13	0.010946	0.048214	0.337130	2.299599	5.575092
mpi_s	14	0.011300	0.059454	0.331017	2.576420	5.492225
mpi_s	15	0.008284	0.068089	0.301032	2.344700	5.050085
mpi_s	16	0.018403	0.065173	0.305920	2.445872	4.844121
mpi_s	17	0.012028	0.043163	0.183916	2.188648	4.846664
mpi_d_b1	2	0.011345	0.100521	0.599424	5.661318	12.860029
mpi_d_b1	3	0.004605	0.050878	0.285109	2.651590	6.590887
mpi_d_b1	4	0.003302	0.035881	0.197465	1.800524	4.324779
mpi_d_b1	5	0.013395	0.069358	0.358072	2.017414	4.818056
mpi_d_b1	6	0.016076	0.087900	0.423820	1.882220	4.948479
mpi_d_b1	7	0.011315	0.133429	0.377192	1.991704	4.093123
mpi_d_b1	8	0.048444	0.089030	0.289423	1.701199	4.096013
mpi_d_b1	9	0.021049	0.128318	0.224985	1.416693	3.545253
mpi_d_b1	10	0.036720	0.065019	0.143607	1.604000	3.356293
mpi_d_b1	11	0.036824	0.044865	0.429384	1.476527	3.246614
mpi_d_b1	12	0.057153	0.086012	0.256409	1.469275	3.069335
mpi_d_b1	13	0.028273	0.089318	0.164263	1.348769	2.839568
mpi_d_b1	14	0.035345	0.137762	0.193101	1.360831	2.576793
mpi_d_b1	15	0.065686	0.056609	0.333242	1.077034	2.342247
mpi_d_b1	16	0.088384	0.170186	0.232426	1.190445	2.673183
mpi_d_b1	17	0.063550	0.100683	0.321198	1.033676	2.235322
mpi_d_b10	2	0.011202	0.100320	0.490371	5.133093	12.812451
mpi_d_b10	3	0.004405	0.072741	0.274235	2.610060	6.651442
mpi_d_b10	4	0.002572	0.038800	0.206272	1.752594	4.325423
mpi_d_b10	5	0.013511	0.046060	0.246191	1.857202	4.967878
mpi_d_b10	6	0.012292	0.061010	0.501696	2.226736	4.846849
mpi_d_b10	7	0.028630	0.074892	0.324899	2.348687	4.724780
mpi_d_b10	8	0.033136	0.076751	0.421567	2.054439	4.763751
mpi_d_b10	9	0.011427	0.065106	0.441110	2.043807	3.925690
mpi_d_b10	10	0.025585	0.064966	0.521247	1.916974	4.068721
mpi_d_b10	11	0.056425	0.072137	0.409166	1.994140	4.182109
mpi_d_b10	12	0.013405	0.065047	0.194139	1.960424	3.844470
mpi_d_b10	13	0.053504	0.076756	0.353379	1.962085	4.098700
mpi_d_b10	14	0.040785	0.048791	0.214288	1.688068	4.018011
mpi_d_b10	15	0.029708	0.068945	0.318337	1.748685	3.467357
mpi_d_b10	16	0.032320	0.053409	0.301304	1.706684	3.695827
mpi_d_b10	17	0.069541	0.026360	0.372808	1.813773	4.145070
pth_d	1	0.010970	0.134539	0.562409	5.175129	13.691565
pth_d	2	0.004940	0.076459	0.347688	2.857205	7.374814
pth_d	3	0.003573	0.051848	0.229342	2.150359	4.972909
pth_d	4	0.003684	0.038503	0.161075	1.465149	3.317525
pth_d	5	0.003694	0.031658	0.151312	1.321783	3.338144
pth_d	6	0.003790	0.028784	0.142153	1.321762	3.349680
pth_d	7	0.003763	0.026387	0.139803	1.304743	3.188144
pth_d	8	0.004115	0.028787	0.138630	1.283514	3.343383
pth_d	9	0.003748	0.031536	0.135220	1.355123	3.255692
pth_d	10	0.003558	0.027552	0.134393	1.348926	3.263586
pth_d	11	0.003471	0.026547	0.129737	1.350006	3.269627
pth_d	12	0.004092	0.026228	0.130530	1.342161	3.298721
pth_d	13	0.003622	0.024947	0.130688	1.416620	3.435578
pth_d	14	0.003550	0.027347	0.129231	1.440535	3.352679
pth_d	15	0.003432	0.028552	0.128762	1.317120	3.407539
pth_d	16	0.003407	0.025594	0.129401	1.307728	3.217864
pth_s	1	0.011531	0.135679	0.638689	5.653982	15.603685
pth_s	2	0.006074	0.079209	0.344402	2.682272	7.451712
pth_s	3	0.008082	0.110075	0.576309	4.553397	10.768347
pth_s	4	0.005642	0.076853	0.366512	2.383057	6.263328
pth_s	5	0.006633	0.090115	0.390295	3.125748	8.640401
pth_s	6	0.006237	0.064111	0.305913	2.507911	6.342987
pth_s	7	0.006669	0.045378	0.320970	2.577101	7.402015
pth_s	8	0.006529	0.073511	0.263426	2.168930	5.381903
pth_s	9	0.005279	0.072766	0.229669	2.222674	5.542345
pth_s	10	0.007412	0.062772	0.235329	2.027325	4.684207
pth_s	11	0.005350	0.059280	0.255718	2.175984	4.669554
pth_s	12	0.004651	0.050309	0.209651	1.866132	4.053005
pth_s	13	0.003995	0.043410	0.269328	1.734115	4.161618
pth_s	14	0.005064	0.081312	0.228838	1.619401	3.885408
pth_s	15	0.005452	0.077201	0.230862	1.626962	4.001535
pth_s	16	0.005054	0.062244	0.281533	1.675430	3.931871

Figure 7. Overall result