YANAN ZHANG

ASSIGNMENT REPORT

PART I. PRE-ASSIGNMENT

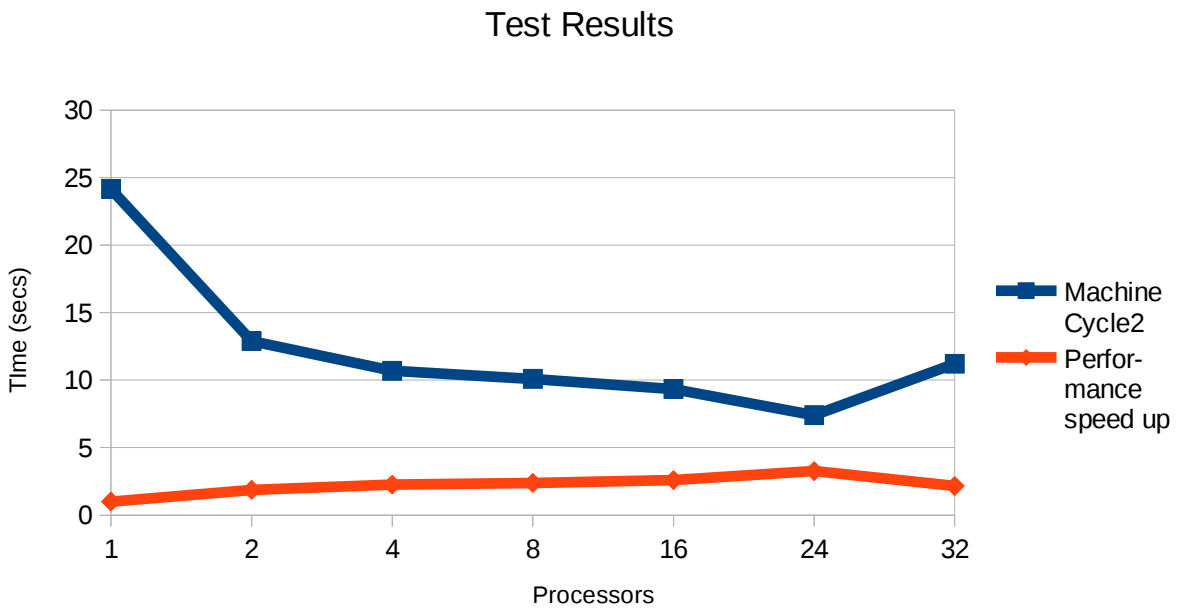
1. Parameters:

iterations = 1000, M = 4096, N = 2048, task_num (number of threads) = 1, 2, 4, 8, 16, 24.

Experiment Machine: cycle2

2. Test Results in cycle2:

Task_num	1	2	4	8	16	24	32
Time (sec)	24.16	12.89	10.69	10.09	9.34	7.40	11.21



3. Analysis:

In Parallel version of SOR, what is different from SOR in serial version is that we should divide the whole task into separate tasks for thread individuals, then assign these tasks to corresponding threads. Threads should be created before assigning tasks to individual threads. After that, threads should do their own works, and data is partitioned by “begin” and “ends” which are calculated by thread IDs in following formula: begin=(M*task_id)/task_num+1; end=(M*(task_id+1))/task_num; And also we can see that in function of threads' working, there is a common lock called barrier() orchestrating among individual threads. Each thread do their individual works and do not influence each other.

YANAN ZHANG

PART II. ASSIGNMENT

1. Introduction

In this assignment, the programs parallelize factorization step, and other steps are done by thread 0 in sequential.

The programs use two different methods to assign the tasks to threads, one method is block assignment, the other is stripe assignment, executable files are denoted by gauss1(block assignment), gauss2(stripe assignment). And the programs run in cycle2 and node2*12*1a.

2. Details of the programs

The main structure of both programs follows the sequential version of gauss elimination. The programs run on the following steps: initialization, compute gauss, including getting pivot, scaling the main row, factorize the rest of matrix, and solve gauss equations. Due to the dependencies of each steps, I found that only factorization step could be paralleled.

In factorization step, due to the loop is in this formula: j = i+1; j< nsize; j++, the amount of tasks which need to assign to each threads is varied. So the programs define an global variable “num_iter”, to record the amount of total tasks to be assigned, and define local variables “begin”, “end”, to delimit the tasks to do for each threads. All the three variables need to be calculated at the beginning of the loop.

The difference between block assignment and stripe assignment is the way of loop formula. In block assignment, the loop ranges (j = begin + I; j<=end + I; j++) , whereas in stripe method, the loop ranges I ~ nize, but the step unit is task_num, which are 2,4 8, 16, etc...

There are some barriers to synchronize between processes, typically, they are located in the middle of functions such like initialization, compute gauss, solve gauss equations and so on. More than that, there are barriers inside loops in order to keep the integrity of variables within the loop.

3. Test Results

Table1: Test Results in cycle2, test matrix: matrix_2000.dat

Task_num		1	2	4	8	16	24	32
Block Method	Time(sec)	4.02	2.05	1.43	1.50	0.94	1.26	1.47
	Normal	1	1.96	2.81	2.68	4.28	3.19	2.73
Stripe Method	Time(sec)	4.05	2.08	2.09	1.70	1.38	1.59	1.84
	Normal	0.99	1.93	1.92	2.36	2.91	2.53	2.18

Table2: Test Results in node2*12*1a, test matrix: matrix_2000.dat

Task_num		1	2	4	8	16	24	32
Block Method	Time(sec)	7.47	6.11	4.61	4.27	5.41	7.54	9.51
	Normal	0.54	0.66	0.87	0.94	0.74	0.53	0.42
Stripe Method	Time(sec)	7.81	7.65	5.09	5.51	6.38	8.20	9.66
	Normal	0.51	0.53	0.80	0.73	0.63	0.49	0.42

Table3: Test matrix jpwh_991.dat, test machine: cycle2

Task_num		1	2	4	8	16	24	32
Block Method	Time(sec)	0.32	0.21	0.21	0.17	0.40	0.59	0.82
	Normal	1	1.52	1.52	1.88	0.80	0.54	0.39
Stripe Method	Time(sec)	0.31	0.40	0.16	0.16	0.40	0.64	0.82
	Normal	1.03	0.80	2.0	2.0	0.80	0.50	0.39

Table4: Test matrix saylr4.dat, test machine: cycle2

Task_num		1	2	4	8	16	24	32
Block Method	Time(sec)	18.54	12.78	10.05	9.47	10.10	9.85	10.22
	Normal	1	1.45	1.84	1.96	1.84	1.88	1.81

Table5: Test Matrix sherman3.dat, test machine: cycle2

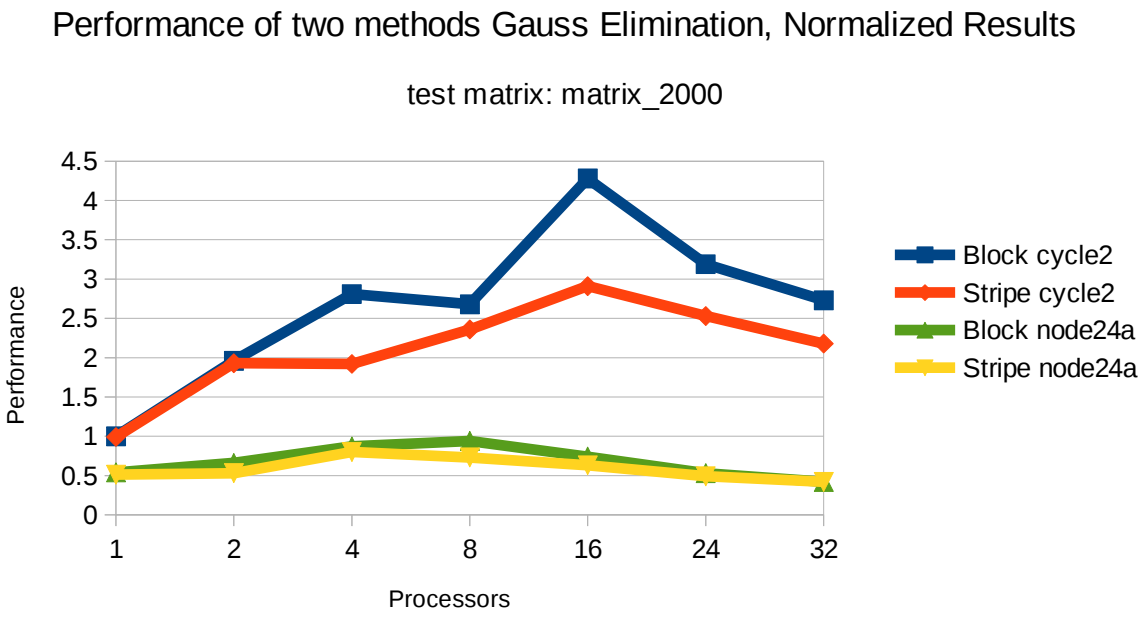
Task_num		1	2	4	8	16	24	32
Block Method	Time(sec)	50.60	29.47	26.58	22.67	26.67	23.55	26.78
	Normal	1	1.72	1.90	2.23	1.90	2.15	1.90

Table6: Test Matrix sherman5.dat, test machine: cycle2

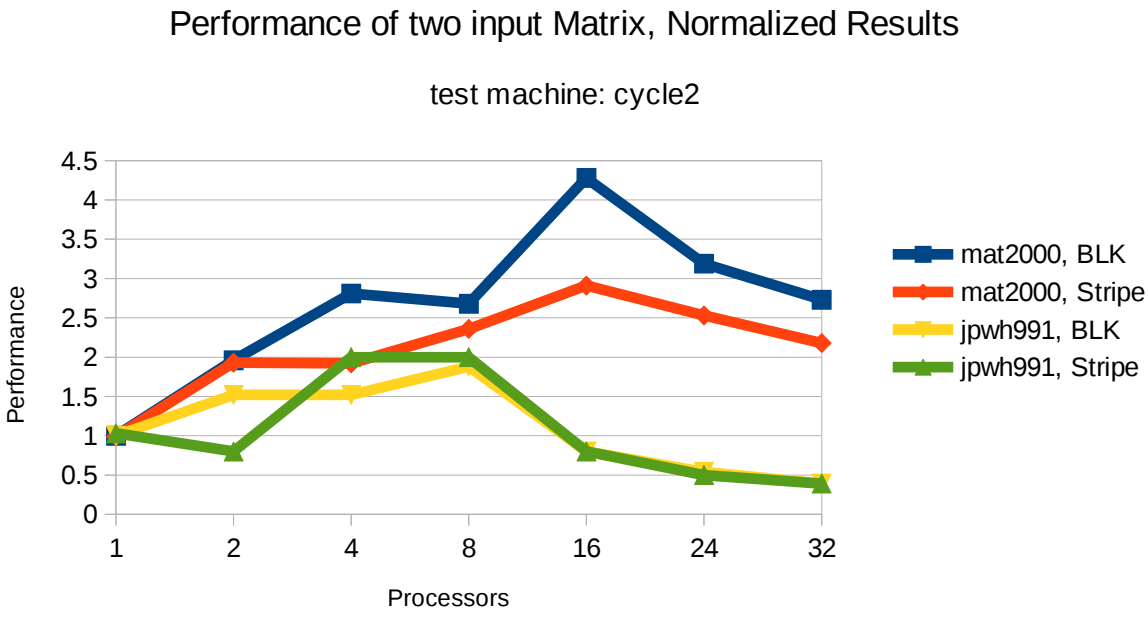
Task_num		1	2	4	8	16	24	32
Block Method	Time(sec)	14.88	9.73	8.60	6.64	7.61	7.29	8.44
	Normal	1	1.53	1.73	2.24	1.96	2.04	1.76

YANAN ZHANG

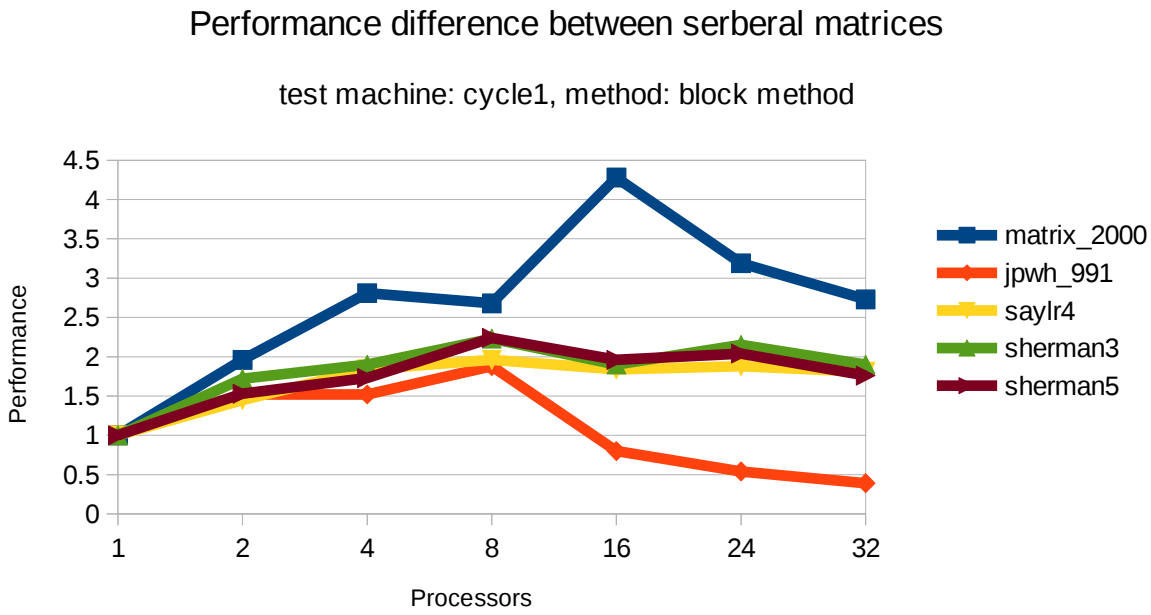
Graph1: Performance difference between two methods and two different machines



Graph2: Performance difference between two input matrices (test in same machine, same method)



Graph3: Performance difference between several matrices



4. Conclusion obtained from the graph

There are four main points that could be learned from the graph. First, no matter what methods are used, peak performances occur at 8/16 processors. Second, the overall performance of block method is better than that of stripe method in both two experimental machines. Third, the overall performance in cycle2 is better than that in node24a. Fourth, the overall performance of jpwh_991 is lower than that of matrix_2000.

5. Analysis and Summary of the experimental results

Theoretically, the peak of the performance would occur at the situation where the number of launched threads equals to the number of processors. But based on the result from the test, it is not always that case. Possible reasons may include: the task amount for each threads are not equal, threads are possessed by other users, size of inputs...and so on. Secondly, in point of space locality of the cache, the block assignment method is dominant to stripe assignment method. That conclusion is supported by the test result where the performance in block method is better than the performance in stripe method. As for the program's performance in node24a is significantly worse than that in cycle2, partly is due to the fact that some user possessed large amount of CPU resources. And I think that is part of the reason why the peak performance would occur in 8 processors instead of 16 processors in cycle2, and let alone the best performance occur in 24 processors when I ran the test program in my local machine.

Moreover, the performance would be varied due to the size of input and the status of input. As we can

see in graph2, the overall performance in jpwh_991 is lower than the performance in matrix_2000. Because the amount of task is very slight, other time cost (overhead) such as thread creation, context switch, cache warm up will be outstanding to the time cost in computations, then they will influence the performance; Whereas in matrix_2000, the amount of work is sufficient, those overheads would hide under the time of computations. Besides, the performance of matrix_2000 is significantly better than others, cause matrix_2000 is a artificial matrix and it does not require any pivot movement, which is a serial step in the program.

In conclusion, parallelizing the program could enhance the performance, but it would not always perform its best that we expect theoretically. It will be influenced by the partial of work which are actually paralleled, the performance and status of experiment machine, a proper method to take care of space/time locality, and the input scale and status of computations.