Ran Pang
Sandhya Dwarkadas
February 1, 2015

Program Assignment #1 Report

Parallel Gaussian Elimination with partial pivoting

The goal of the program is to implement a paralleled version of Gaussian Elimination with partial pivoting using pthread. Considerations of parallelizing include task assignment, inter-threads communication and synchronization. In this project, the evolution of the programs mainly focus on task assignment, while the approach to do inter-threads communication and synchronization are simply inherited through out the programs.

Before we discuss the parallel approaches, let's first take a look at the algorithm of Gaussian Elimination with partial pivoting:

```
for (i=0; i<matrix_row_number; i++) {
      find_pivot_row(matrix);
      swap_row(pivot_row, row_i)
      normalize_row(row_i)
      for (j=i; j<matrix_row_number; j++) {
            eliminate_row(row_j, row_i)
      }
}
```

As to inter-thread communication, all threads operate on the same globally visible matrix, which requires careful task assignment and synchronization to prevent synchronizing issues. As to task assignment, the most intuitive and practical way is to assign each thread a block of continuous rows in the matrix. find_pivot_row() can be completed concurrently by each thread where each thread searches for its local pivot row within the block assigned to that thread, and update the global pivot row. swap_row() can also be paralleled by assigning each thread some columns to swap. Similarly normalize_row() can be done concurrently by all threads where each thread takes charge of some columns. eliminate_row() can be naturally paralleled where each thread eliminate the rows in its block.

At a glance it seems that parallelizing find_pivot_row() and eliminate_row() make the greatest contribution to the performance, while threading on swap_row() and normalize_row() seems trivial. To provide evidence with such intuition, two of the four programs parallelize swap_row() and normalize_row().

A more important consideration on performance should be the way to assign tasks to each thread. If each thread is assigned a block a continuous rows, the advantage of parallelizing is not fully utilized. The reason it that as we moving down the rows of the matrix, the threads that takes charge of the rows near the top of the matrix are becoming idle, since once we move over a row, we no long need to access that row again.

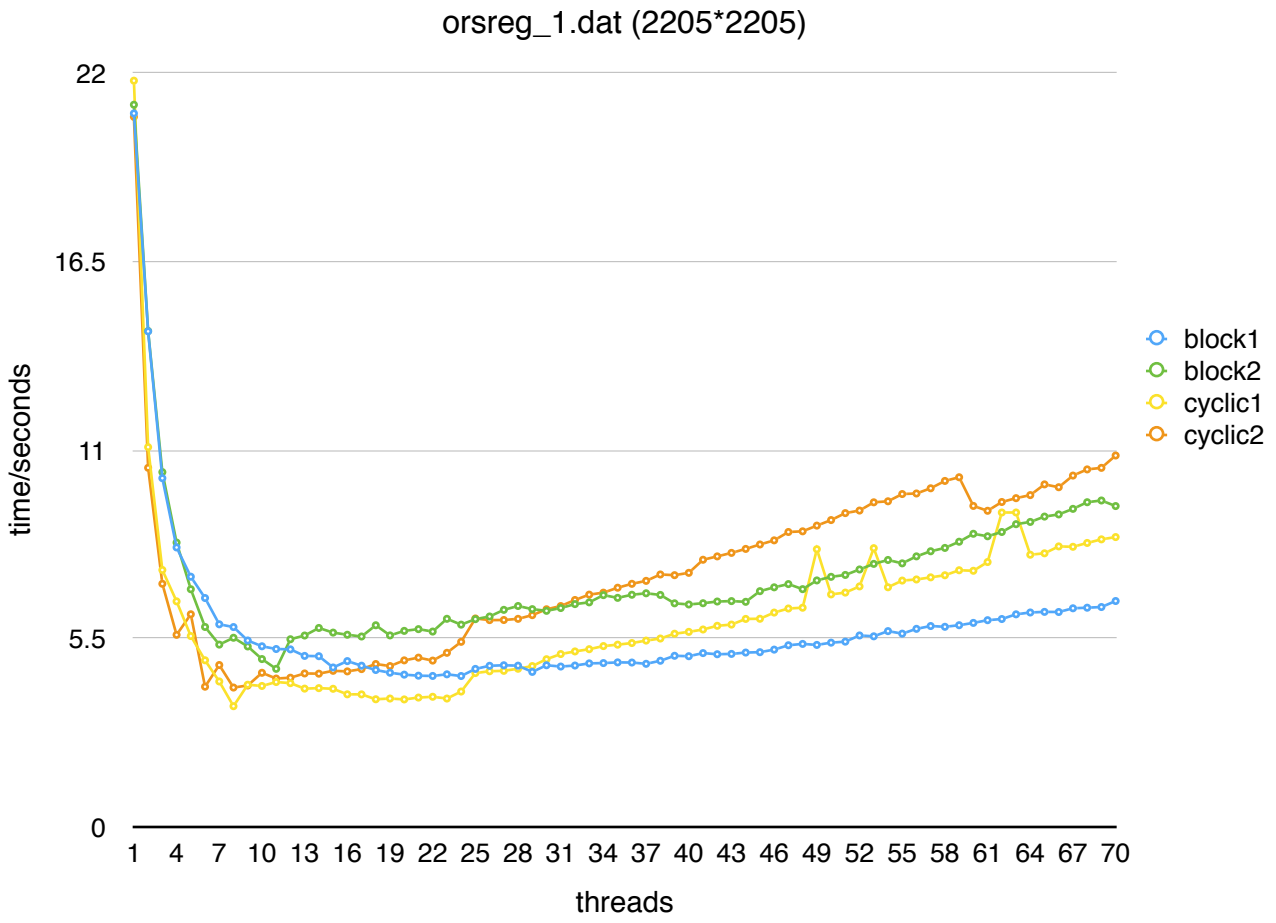
One solution is to assign rows to each thread in a cyclic pattern. For example if there are 3 threads, we assign row 1, 4, 7… to thread one, row 2, 5, 8… to thread two, and row 3, 6, 9… to thread three. In this way thread idling can be avoid to a large extend as the program runs to its end.

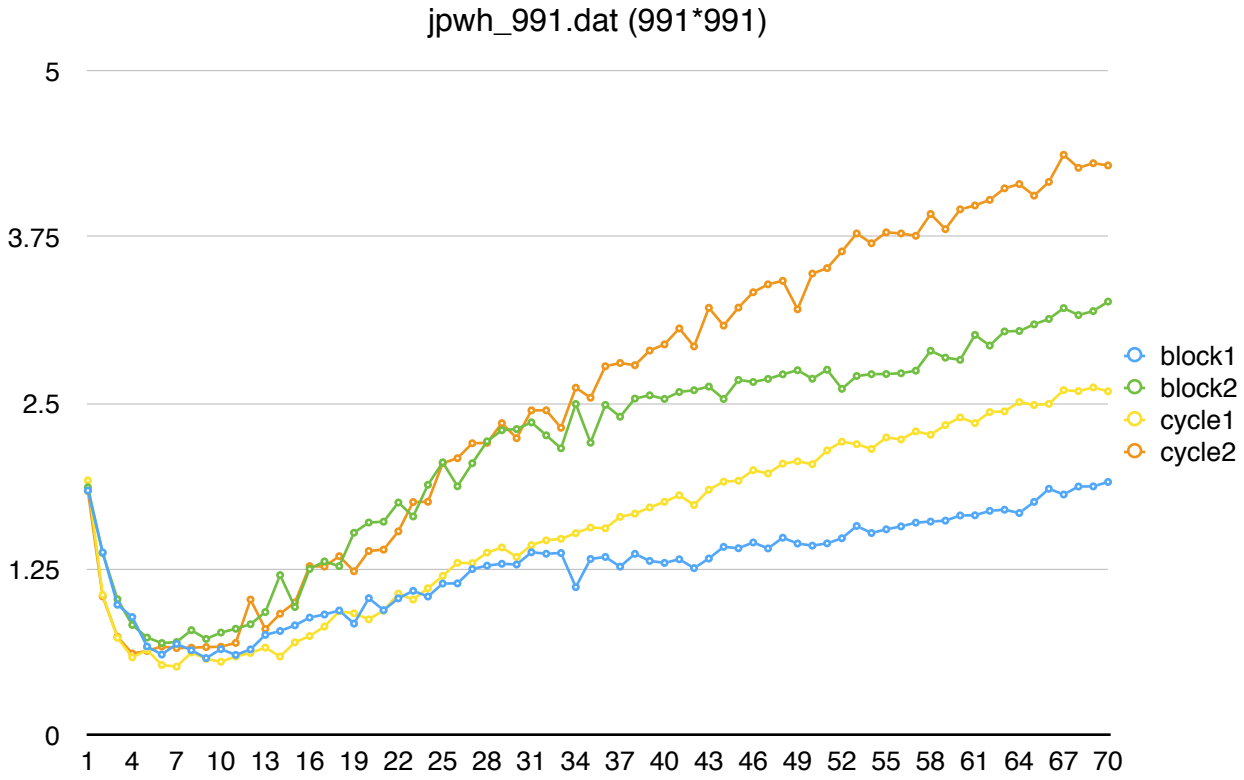
Four c++ programs are produced in the experiment with their code in files named gauss_pthread_block_v1.cpp gauss_pthread_block_v2.cpp gauss_pthread_cyclic_v1.cpp gauss_pthread_cyclic_v2.cpp where gauss_pthread_block_v1.cpp gauss_pthread_block_v2.cpp implement block assignment while gauss_pthread_cyclic_v1.cpp gauss_pthread_cyclic_v2.cpp use cyclic assignment. The code file name ending with _v1 do not parallelize swap_row() nor normalize_row(), while code file name ending with _v2 do.

Theses programming were all running on cycle3.cs.rochester.edu. The features of the experimenting environment is show in Table below.

Environment Specifications	
machine	cycle3.cs.rochester.edu
OS	Linux 3.17.7 x86_64
processor model name	Intel Xeon CPU E5-2430 2.20GHz
number of processors	24
number of cores per processor	6
cache size	15360 KB
memory available	63127064 KB

The experiment were performed on all four programs and on matrices of different sizes and a variety of number of threads. In each execution of a program, 5 iterations were performed to average out noises. The results are shown in figures below.


orsreg_1.dat (2205*2205)
time/seconds
22
16.5
11
5.5
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70
threads
block1
block2
cyclic1
cyclic2


jpwh_991.dat (991*991)
5
3.75
2.5
1.25
0
1 4 7 10 13 16 19 22 25 28 31 34 37 40 43 46 49 52 55 58 61 64 67 70
block1
block2
cycle1
cycle2

Some common patterns across different matrix sizes include: running time decrease in descending speed as thread number rises from 1 to around 10 for all four programs; running time increases in ascending speed as thread number rises further; within the speeding up interval, cyclic assignment performs slightly better than block assignment, performance of two cyclic programs are similar and performance of two block programs are similar; as the number of threads further increases, the performance of four programs diverges in a way that cyclic program that parallelize swap_row() and normalize_row() is the slowest, block assignment with paralleled swap_row() and normalize_row() is the second slowest, the cyclic program that does not parallelize these functions runs faster, and the block program that does not parallelize these functions run the fastest.

This result suggests the drawback of over-parallelizing, i.e. if task are split into too small pieces which each thread takes some pieces, it not only does not significantly increase the performance of the program as more threads decrease running time, it also deteriorate the performance at a faster pace as the cost of more threads overcomes its benefit. Cyclic task assignment exhibits similar effects as over-parallelizing does when more threads means worse performance.

The impact of matrix size on relative performance among the four parallel designs is also indicated by the differences between the two figures above. Larger data size suggests better parallel performance and less cost of over-parallelization. The program profiling was performed using standard gprof with the following results which indicates that the running time is mostly consumed by eliminate_row(). swap_row() takes a little more time than normalize_row() and find_local_pivot(). This provide support of the efficiency to parallelize eliminate_row() by assign different rows to different threads

Functions Running Rime by gprof

program/function	eliminate_row()	normalize_row()	swap_row()	find_local_pivot()
block_v1	96.73	0.57	1.43	0.86
block_v2	99.27	0.00	0.30	0.30
cyclic_v1	100.17	0.00	0.00	0.00
cyclic_v2	98.81	0.27	0.82	0.00