Qiyuan Qiu
02/12/14
CSC458 Spring 2014
Kai Shen

Parallel implementation of Gaussian Elimination:

In the sequential code, the function that performs all the computation of the Gaussian Elimination is the following computeGauss(int nsize) function.

```
void computeGauss(int nsize)
{
    int i, j, k;
    double pivotval;

    for (i = 0; i < nsize; i++) {
        getPivot(nsize,i);

        /* Scale the main row. */
        pivotval = matrix[i][i];
        if (pivotval != 1.0) {
            matrix[i][i] = 1.0;
            for (j = i + 1; j < nsize; j++) {
                matrix[i][j] /= pivotval;
            }
            R[i] /= pivotval;
        }

        /* Factorize the rest of the matrix. */
        for (j = i + 1; j < nsize; j++) {
            pivotval = matrix[j][i];
            matrix[j][i] = 0.0;
            for (k = i + 1; k < nsize; k++) {
                matrix[j][k] -= pivotval * matrix[i][k];
            }
            R[j] -= pivotval * R[i];
        }
    }
}
```

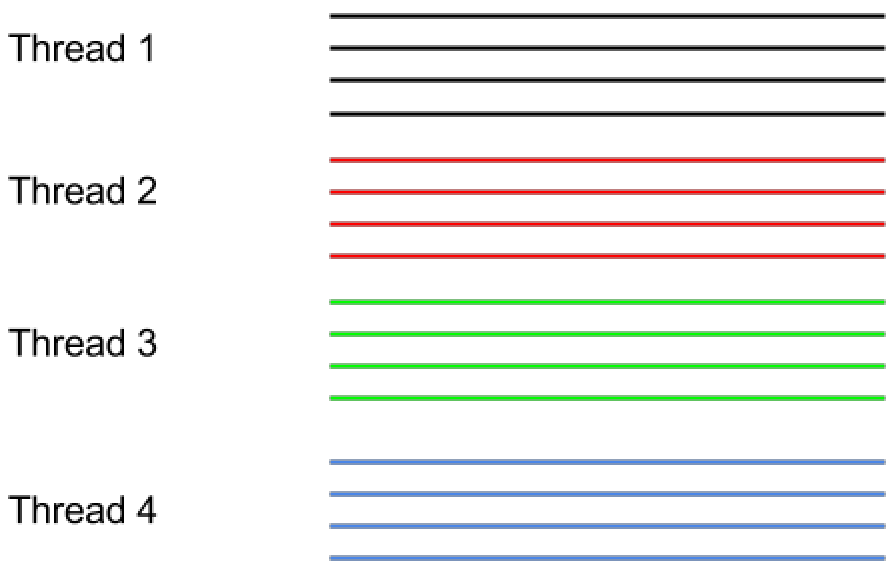
Two methods of parallelizing the red parts were implemented.

Row partition:

During each iteration i. Total number of rows(n_row) that need to be factored are assigned to sections. The number of sections are denoted as n_sections. Each section has the same amount of rows. The total number of rows in all sections is greater or equal to n_row(equal when n_section is a factor of n_row).

Just like the following figure. The overall matrix is divided into four sections. Each thread is in charge of updating the rows in their own section.

Qiyuan Qiu
02/12/14
CSC458 Spring 2014
Kai Shen



Parallel code is the following. The red part is the key part that does what is described in the above figure. The blue part is the part that synchronize all the threads.

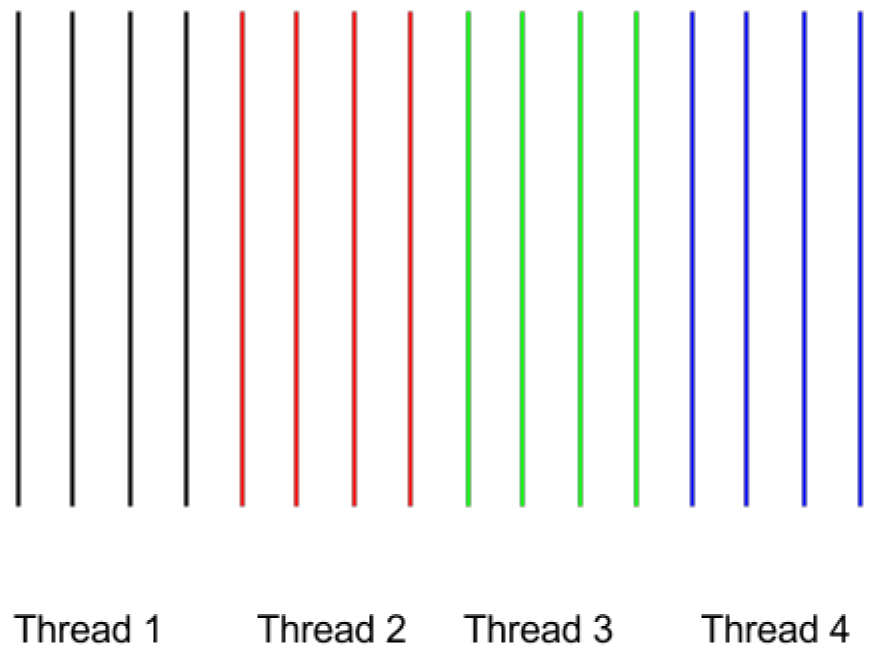
```
void *computeGauss_row_version(void *arg)
{
    thread_arg *message = (thread_arg *) arg;
    int i, j, k, nsize, task_id, begin, end;
    double pivotval;
    pthread_attr_t attr;
    pthread_t *tid;

    task_id = message->task_id;
    nsize = message->nsize;
    for (i = 0; i < nsize; i++) {
        if(task_id == 1){
            getPivot(nsize,i);

            /* Scale the main row. */
            pivotval = matrix[i][i];
            if (pivotval != 1.0) {
                matrix[i][i] = 1.0;
                for (j = i + 1; j < nsize; j++) {
                    matrix[i][j] /= pivotval;
                }
                R[i] /= pivotval;
            }
            barrier(task_num);
        }
        else
            barrier(task_num);

        /* Row partition */
       begin = set_begin(nsize, i, task_id);
       end = set_end(nsize, i, task_id);
       for (j = begin; j <= end; j++) {
            pivotval = matrix[j][i];
            matrix[j][i] = 0.0;
            for (k = i + 1; k < nsize; k++) {
                matrix[j][k] -= pivotval * matrix[i][k];
            }
            R[j] -= pivotval * R[i];
        }
        barrier(task_num);
    }
}
```

Column partition:



```
void *computeGauss_col_version(void *arg)
{
    thread_arg *message = (thread_arg *) arg;
    int i, j, k, nsize, task_id, begin, end;
    double pivotval;
    pthread_attr_t attr;
    pthread_t *tid;

    task_id = message->task_id;
    nsize = message->nsize;
    for (i = 0; i < nsize; i++) {
        begin = set_begin(nsize, i, task_id);
        end = set_end(nsize, i, task_id);

        if(task_id == 1){
            getPivot(nsize,i);

            /* Scale the main row. */
            pivotval = matrix[i][i];
            if (pivotval != 1.0) {
                matrix[i][i] = 1.0;
                for (j = i + 1; j < nsize; j++) {
                    matrix[i][j] /= pivotval;
                }
                R[i] /= pivotval;
            }
        }
        barrier(task_num);
        for (j = i+1; j < nsize; j++) {
            pivotval = matrix[j][i];
            for (k = begin; k <= end; k++) {
                matrix[j][k] -= pivotval * matrix[i][k];
            }
            if(task_id == 1)
                R[j] -= pivotval * R[i];
        }
        barrier(task_num);
        for(k = begin; k <= end; k++)
            matrix[k][i] = 0.0;
    }
}
```

The key here for col version is that because all thread are using the i_th column in i_th iteration. Therefore setting the i_th col to 0.0 need to happen after all threads have finished updating their own section.

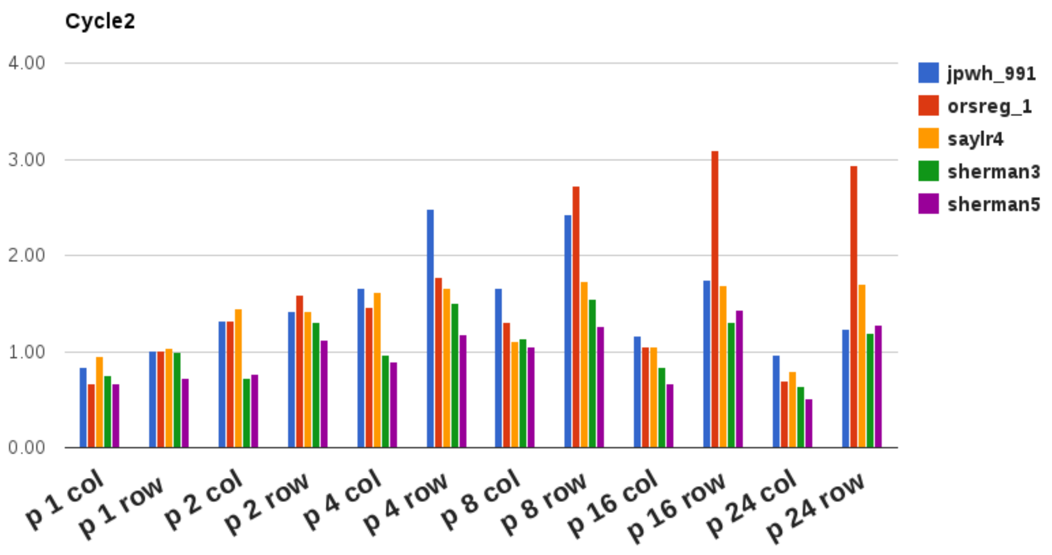
Qiyuan Qiu
02/12/14
CSC458 Spring 2014
Kai Shen

The method I used to verify that my implementation is correct is that the error I got from my implementation is the same as sequential version.

Parallel performance:

For parallel implementation, the one thread version should be as least comparable to the sequential version. Later with more threads helping out with Gaussian Elimination, there should be speedup comes with this. My implementation has achieved both goals.

The following graph can prove this.



The very left two chunks of bar in this graph are the speedup for the parallel Gaussian Elimination running on cycle2 normalized to the speed of sequential Gaussian Elimination. Each bar is close to the vertical position one. This implies that my implementation with one thread has a comparable performance on all matrixes to sequential version. Which achieves the first goal. The second goal is that there should be sped up from multi-thread implementation. This is also shown in the graph. The bars that are higher than one in the above graph are experiments that enjoy speedup from parallelization.

Experiments and Analysis:

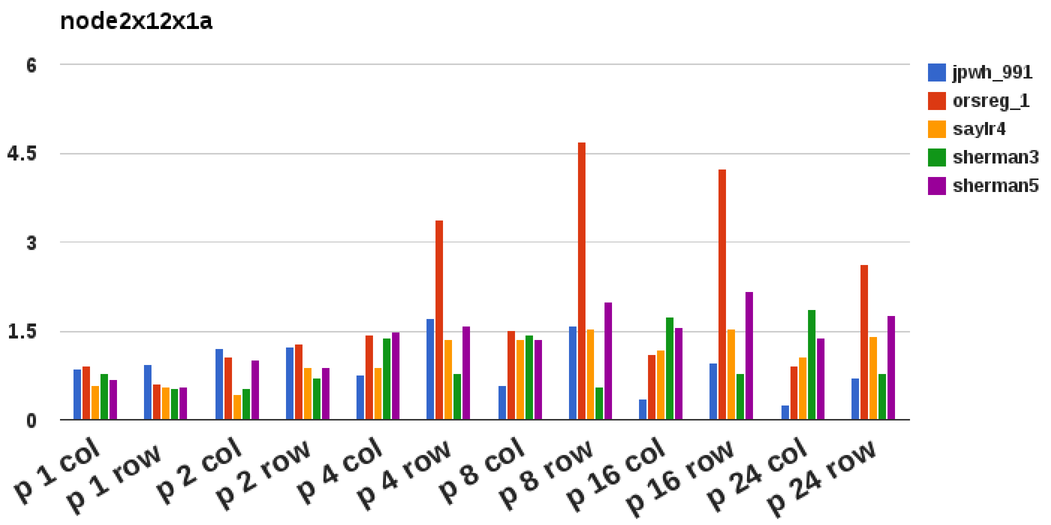
Experiments were run on two machines, node2x12x1a and cycle2. Both of them has 24 processors. For parallel programs, 1, 2, 4, 8, 16, 24 threads version were experimented in both row partition and col partition. For each case of thread number and each case of machine, I ran the code for ten iterations and then take the average as the final runtime. For instance, jpwh_911 as input to column partition program on node2x12x1a machine with 4 threads, the column partition program will be run ten iterations. Each iteration is timed, the average for these ten iterations is taken down in the following chart in position (jpwh_991, pthread 4 col).

Qiyuan Qiu
02/12/14
CSC458 Spring 2014
Kai Shen

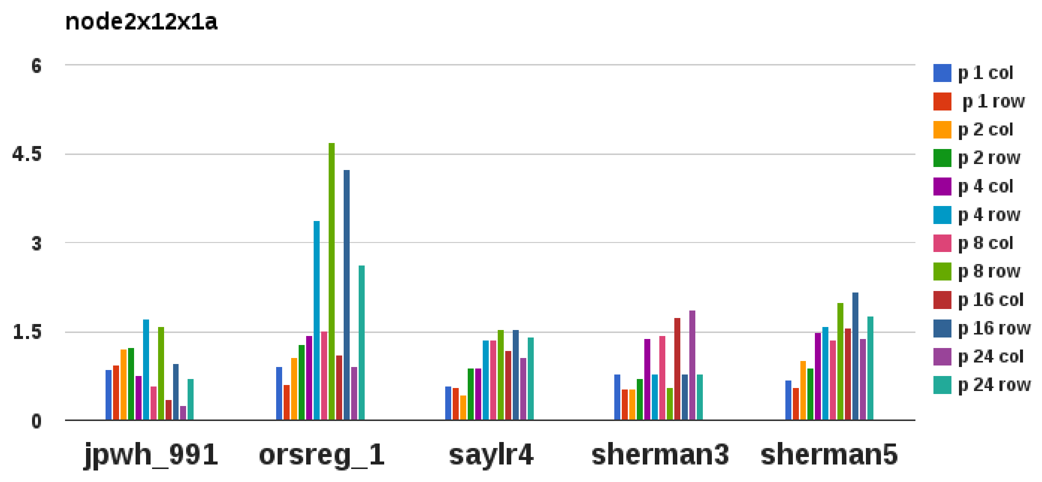
Results from node2x12x1a.

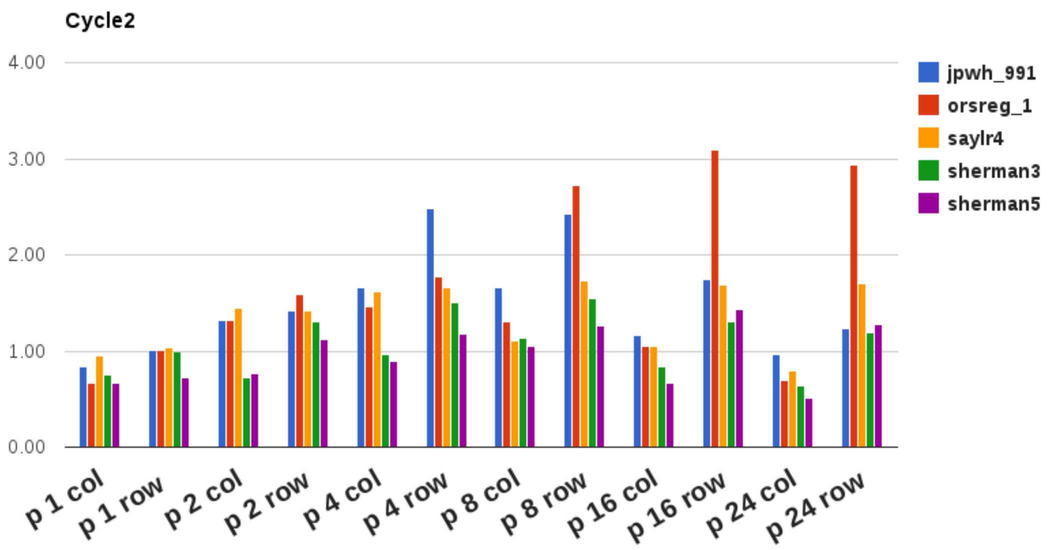
node2x12x1a	seq	pthread 1 col	pthread 1 row	pthread 2 col	pthread 2 row	pthrea d 4 col	pthread 4 row	pthrea d 8 col	pthread 8 row	pthread 16 col	pthread 16 row	pthread 24 col	pthread 24 row
jpwh_991	0.58	0.67	0.63	0.48	0.47	0.76	0.34	1.00	0.37	1.67	0.61	2.25	0.82
orsreg_1	8.99	9.98	14.77	8.55	7.02	6.27	2.66	5.93	1.92	8.07	2.12	9.79	3.43
saylr4	38.17	66.66	68.69	87.34	43.85	43.60	28.10	27.79	24.86	32.13	24.78	35.71	27.17
sherman5	30.80	46.04	55.59	30.33	35.19	20.80	19.40	22.62	15.43	19.75	14.22	22.14	17.48
sherman3	106.11	137.54	195.81	204.48	149.97	76.21	134.03	74.25	189.42	60.75	133.96	57.02	136.87

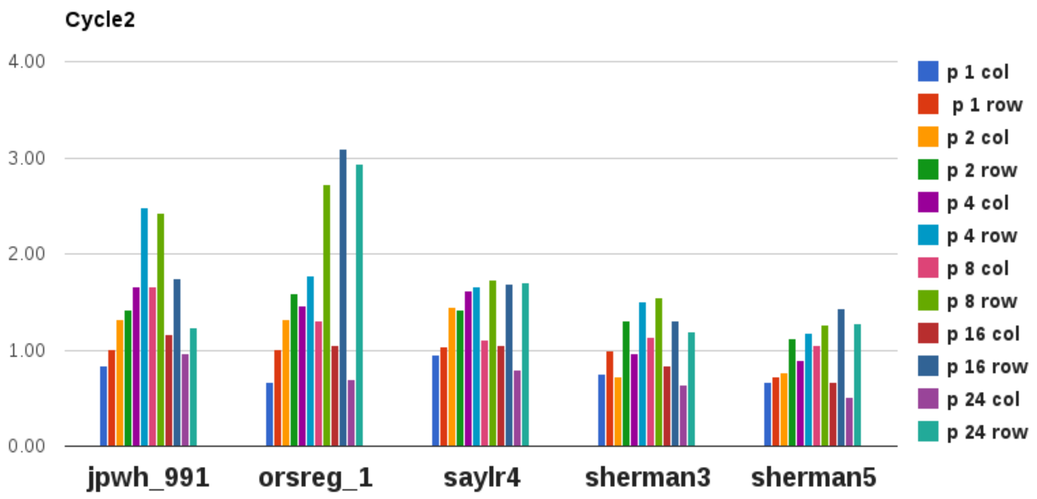
Following graph is the speedup chart for node2x12x1a. This is normalized to the time spent by sequential program. From the result we can see that when the number of threads are not very large, the more threads we use, the more speed up we get until the number of threads are too large. Which means that the speedup we get with the increasing of threads first goes up and then goes down. For cases where the matrix is small, we can even get slow down. For example, when the input matrix is jpwh_991, we have slow down using more than eight threads.



The following graph shows the speedup gained for different sized matrixes on the same node2x12x1a machine. We can see that the greatest speedup happens with row partition and on the “orsreg_1” matrix. The most insignificant speedup(or rather slowdown) happens to the the column partition on matrix “jpwh_991”. The reason for this is that the thread synchronization has overhead, and cache size has great impact on performance. More specifically, when the matrix is too small, the overhead of synchronization outweighs the benefits brought by parallelism. When the cache size matches the size of rows, the speedup is the most significant.


node2x12x1a
6
4.5
3
1.5
0
jpwh_991
orsreg_1
saylr4
sherman3
sherman5
p 1 col
p 1 row
p 2 col
p 2 row
p 4 col
p 4 row
p 8 col
p 8 row
p 16 col
p 16 row
p 24 col
p 24 row


Cycle2
4.00
3.00
2.00
1.00
0.00
p 1 col
p 1 row
p 2 col
p 2 row
p 4 col
p 4 row
p 8 col
p 8 row
p 16 col
p 16 row
p 24 col
p 24 row
jpwh_991
orsreg_1
saylr4
sherman3
sherman5



Comparison and Analysis:

Another thing noticeable is that the column partition parallel version on this sized(991 * 991) matrix does not provide much speed up, except for the 2 thread version. Overall the col partition version on this matrix only slow the program down. However the row partition enjoys more speed up. This I believe results from cache organization. If we partition the input matrix into chunks of columns, because the data layout in memory is row based, therefore we have more cache misses, which significantly slows down the overall speed.