Oriah Ulrich
CSCI 551

Report on Parallelization of Gaussian Elimination with Partial Pivoting

The data is randomly generated and stored into Matrix structs. The data elements for a particular matrix are stored in a 2D array of doubles. The implementation is that of an array of pointers to double pointers that store the doubles themselves. In other words, the Matrix struct encapsulates the actual matrix data stored in a 2D array.

The program generates the column vector b and square matrix A. The goal of the program is to solve the equation A**x** = **b**. One of the ways **x** can be solved is by reducing the coefficients of the system of equations to find values feach element in **x**. This program implements a parallelized gaussian elimination method with partial pivoting to reduce and solve the system of linear equations.

Examining the data dependencies of A**x** = **b**, the best parallelization scheme would be to parallelize only pieces of the gaussian elimination. The first piece of parallelization is on the pivoting step. The step where the maximum value is found could be parallelized by a sort of “greater than reduction”, however openmp does not implement this so I had to implement my own reduction by temporarily storing the partial maximums in a shared array of maximums. However, after experimenting around with the maximum value step i realized the speedup was insignificant, especially compared to the time taken during the elimination step. The next piece that avoids data dependencies is the step that involves reducing the rows (ie. the elimination step). Parallelizing this step proved to be meaningful. The speedup seems to be roughly greater than 1. Measurements are shown in the tables and speedup charts below. The last and final step in the gaussian elimination process is the final upper triangle reduction, where the rows in the upper triangle are reduced to solve for **x**. This is also easily parallelizable by parallelizing the column substitution of the k-th x in **x** to find the exact **b** at some k-th x to reduce the problem size. The timing results here were significantly different. However, the timing results were still magnitudes less than the time it takes to compute the upper triangle.

Synchronization happens whenever a greatest value is found and that thread needs to wait for other threads to finish writing their values memory. See pseudocode for in depth details.

My implementations choices are justified by the timing results and data dependencies in gaussian elimination. During the elimination step where subtraction is done, the row used as the pivot has the greatest valued leading coefficient in that column. The rows to eliminate include all rows below the pivot row, whose values are to the right of the pivot column. Row elimination can therefor be done in parallel. It is not necessary to eliminate the part of the row that has already been eliminated, so the simulation only moves from the pivot column to the end of the equation.

I also parallelize the reduced row operations that reduce the row to a diagonal of elements.
simply by parallelizing the substitution for x_k back into the equations for each column and performing a difference operation on **b** and the substituted term in each equation.

Timing Table

Run #	1	2	5	10	20	30
0	3314.6	1629.3	695.8	461.3	450	354.7
1	3296.1	1699.1	780.5	714.6	498.2	356.1
2	3405.9	1849	686.5	383.7	368.6	362.5
3	3331.8	1617.7	903.6	542.8	430.2	255.8
4	3412.6	1705.8	714.7	441.3	310.1	223.4

Num Threads

[seconds]

Graph of Run Timings

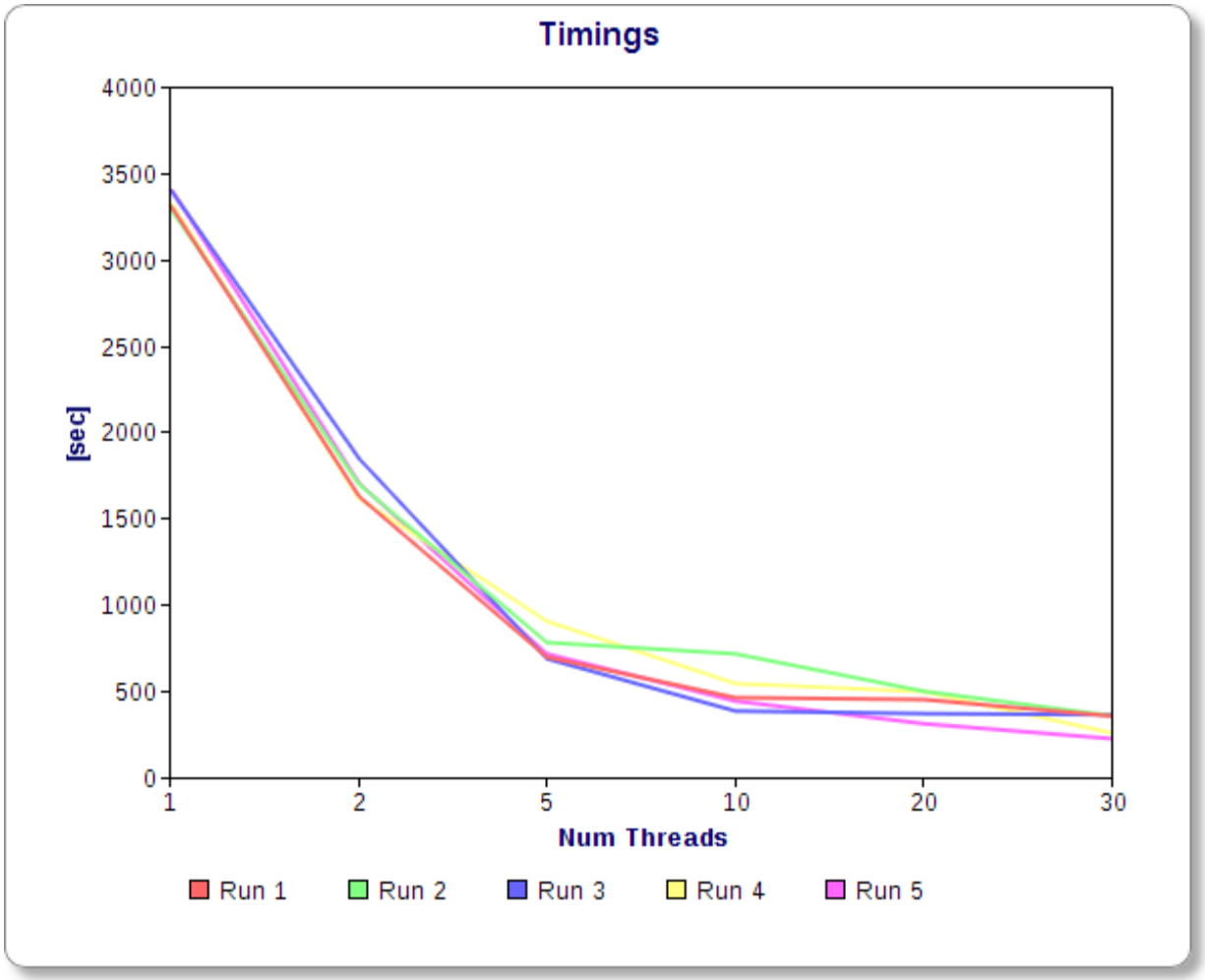


Table of I2-Norm

Run #	1	2	5	10	20	30
0	1.76E+00	4.19E+00	9.30E+00	3.19E+00	4.61E+00	9.34E+00
1	7.72E+01	3.48E+00	5.25E+00	4.03E+01	6.24E+00	7.41E+00
2	5.36E+01	5.70E+00	4.01E+01	5.60E+00	5.46E+01	9.96E+01
3	2.76E+00	8.19E+00	5.11E+00	7.27E+00	4.61E+00	8.26E+00
4	3.76E+00	3.85E+00	9.41E+00	1.30E+00	7.31E+00	2.64E+00

Num Threads

[i2-norm]

Table of Speedup and Efficiency

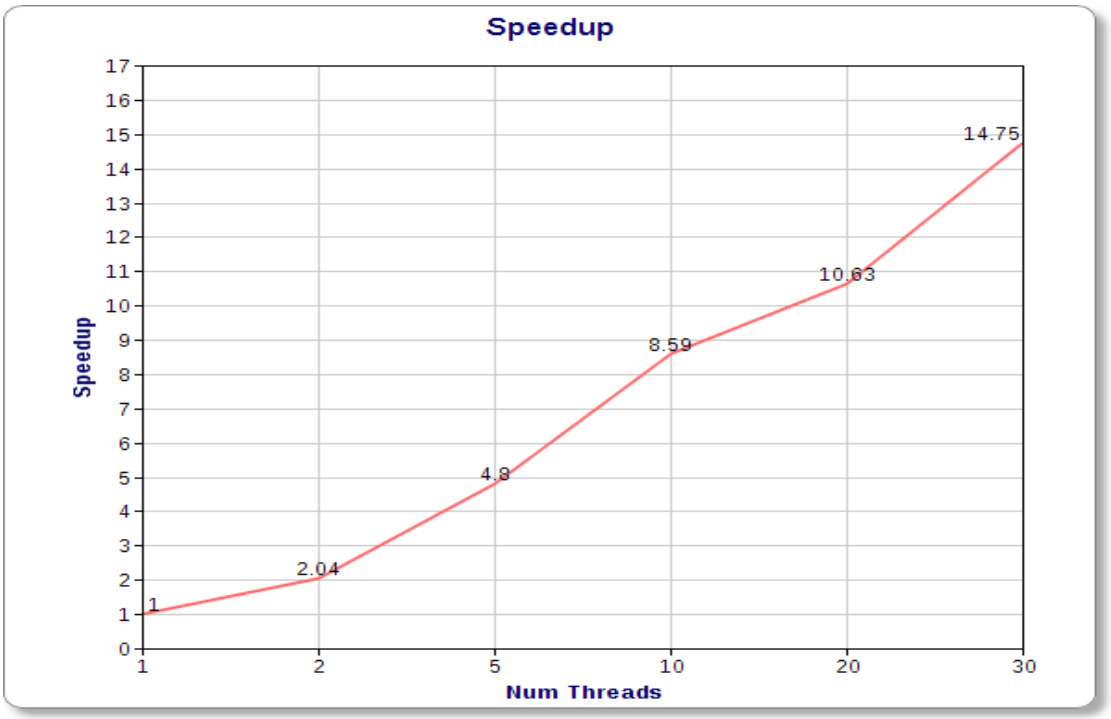
	1	2	5	10	20	30
[Speedup]	1	2.04	4.8	8.59	10.63	14.75

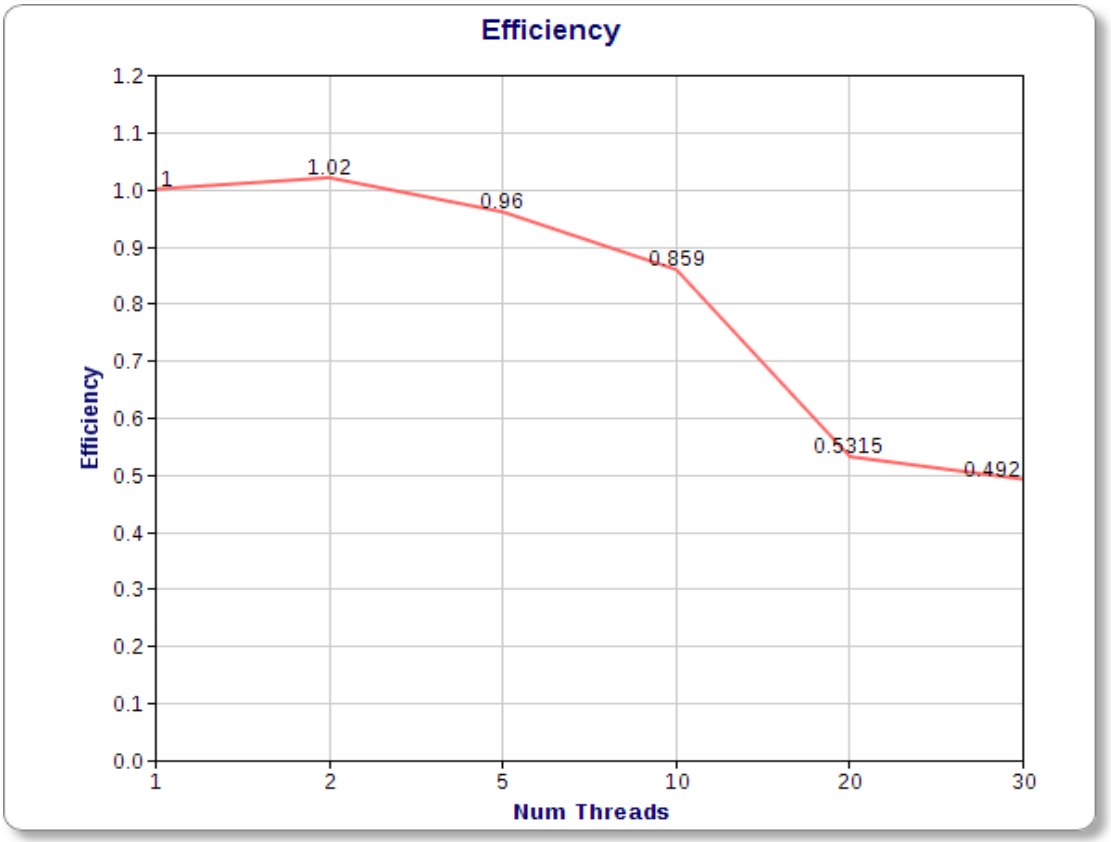
Num Threads

	1	2	5	10	20	30
[Efficiency]	**1**	**1.02**	**0.96**	**0.859**	**0.5315**	**0.492**

Num Threads

Graphs of Speedup and Efficiency





Conclusions

I did not expect the efficiency to get worse with 20 cores. Although the speedup seems to be increasing with the number of processors, the speedup seems to be optimal at 10 cores on this problem size where n = 8000. Clearly this is the point where efficiency starts to drop since the speedup change is significantly smaller at this point and any neighborhood of cores around 20 would also have similar speedup change. So the most efficient number of cores is where the change is maximum somewhere around 5 to 10 cores would be the sweet spot number of cores to maximize the efficiency and minimize timings. However, if efficiency was not a concern and cores are expendable then running at 20 cores or more would be reasonable. The point is, even though the speedup is always increasing, the rate of increase is decreasing. Ideally we could through any number of processors and get good timings with some speedup and still have 'good' efficiency. Clearly the efficiency dramatically decreases because the speedup isn't significantly affected after adding some number of cores to the problem.

Since the biggest portion of work has to do with eliminating the lower triangle to find the value of the n-th x, the row elimination of the next pivot is dependant on previous row eliminations. So i think another way to parallelize the gaussian elimination would be to stagger the operations into parallel stages. So that the multiplier value necessary to compute the next column's elimination is computed **first**. Then that could trickle down into the **rest** of the columns in parallel. Meanwhile the other cores could work on completing the rest of the column with the multiplier value. So the parallelization can get ahead of the row elimination and move onto the next columns by precalculating partial multipliers. However, there is still a data dependency. The next stage will require that the first row be eliminated. Then as each row is eliminated that would reveal the next set of values for the next column. Although, the overhead from this might be greater than the time it takes to simply partition the rows of the matrix during elimination.

I think it would be possible to maximize the efficiency of the parallelization by reducing thread fork and join overhead. Maybe implementing a fork once and join later sort of scheme would reduce the overhead. This scheme would replace the overhead of thread creation with one of waiting for the master thread to finish initializing each iteration step.

Pseudocode

This here parallelizes the part of the code that finds the greatest value in the column during the current iteration. It splits up the work amongst the threads to find the greatest value. I put a critical section around the greatest_value write to ensure only threads that are ready to write and the value is still the greatest so far, otherwise each thread will continue to iterate over it's share of the column.

```
greatest_value = mat->data[pivot][pivot];
greatest_val_row_id = pivot;
#pragma omp parallel num_threads(thread_count) \
   shared(mat, pivot, greatest_value, greatest_val_row_id, aug_col_size) \
   private(row, multiplier, k)
{
   #pragma omp for schedule(static, 1)
   for (row = pivot + 1; row < aug_col_size; ++row)
      if (labs(greatest_value) < labs(mat->data[row][pivot]))
         #pragma omp critical (greatest_value)
            if (labs(greatest_value) < labs(mat->data[row][pivot]))
               greatest_value = mat->data[row][pivot];
               greatest_val_row_id = row;

}
```

The following is the row elimination step. This part is the more significant chunk of the program that needs to be parallelized. The best method of parallelization i found that would work was to simply separate the rows of the matrix and map sets of rows to threads. This just came down to forking threads that each have access to a chunk of the data in A. Specifically this means each thread has access to a subinterval rows in A.

```
#pragma omp parallel num_threads(thread_count) \
   shared(mat, pivot, aug_col_size) \
   private(row, multiplier, k)
{
   #pragma omp for schedule(static, 1)
   for (row = pivot + 1; row < aug_col_size; ++row):
      /* Find constant multiplier used to eliminate the A[row][pivot] element from A[row]*/
       multiplier = mat->data[row][pivot] / mat->data[pivot][pivot];

       /* perform a standard row elimination step */
       for (k = pivot; k < aug_row_size; ++k):
          mat->data[ row ][ k ] -= mat->data[ pivot ][ k ] * multiplier;

}
```

The following code is my implementation for the row reduction into a diagonal matrix used to solve for vector x. The column substitution is parallelized and the b column vector is updated with the result of the operation: b - A[rows in A][col] .

```
for (int row = aug_col_size - 1; row >= 0; --row) {
x = MATDATA(augmat)[row][b_vec_index] / MATDATA(augmat)[row][row];
MATDATA(augmat)[row][b_vec_index] = x;
MATDATA(augmat)[row][row] = 1;
#pragma omp parallel num_threads(thread_count) \
      shared(augmat, x, b_vec_index, aug_col_size)
{
         #pragma omp for schedule(static, 1)
         for (int col_item = row - 1; col_item >= 0; --col_item) {
               MATDATA(augmat)[col_item][b_vec_index]
                        -= ( MATDATA(augmat)[col_item][row] * x );
               MATDATA(augmat)[col_item][row] = 0;
         }
}
```