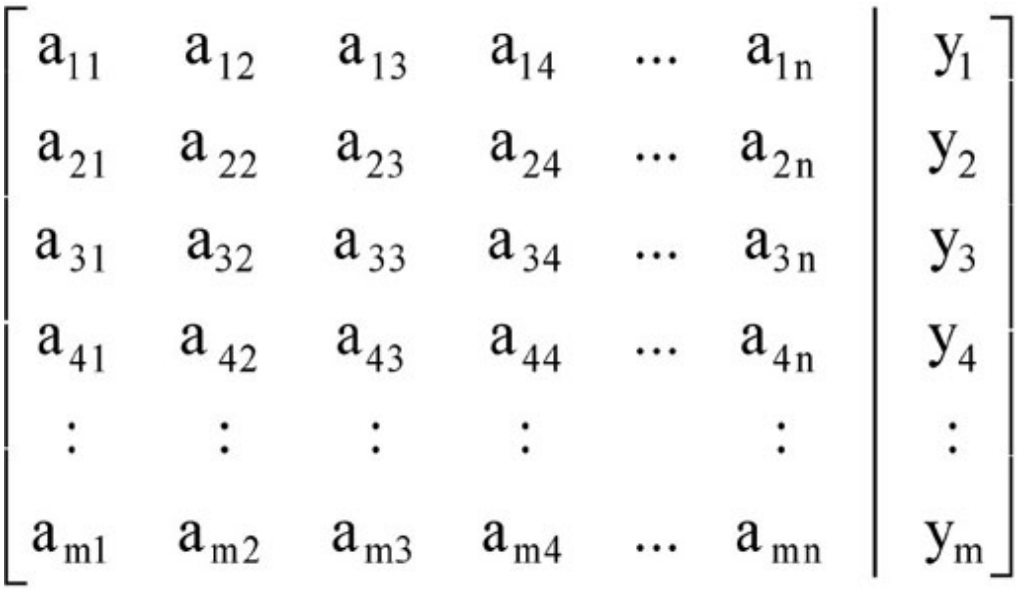
# GAUSSIAN ELIMINATION

*With Partial Pivoting and OpenMP*




Tim Whitaker
CSCI 551 - Spring 2015

# GAUSSIAN ELIMINATION

*With Partial Pivoting and OpenMP*

Tim Whitaker
CSCI 551 - Spring 2015

## Introduction

In this project, I am implementing a parallel version of the Gaussian Elimination Algorithm in C++ with Partial Pivoting and OpenMP. Gaussian Elimination is a method used to solve systems of linear equations and partial pivoting is a little change to the algorithm which helps a couple problems with standard Gaussian Elimination, namely the divide by zero and round off errors. Our algorithm consists of a few steps. The first is a process called forward elimination, where we transform a matrix into a form called upper triangle. We go from something like this:

$$\left[\begin{array}{ccc|c} 2 & -2 & -1 & 3 \\ 3 & -2 & 2 & 7 \\ 1 & -1 & 5 & 7 \end{array}\right]$$

To this:

$$\left[\begin{array}{ccc|c} 2 & -2 & -1 & 3 \\ 0 & 1 & 7/2 & 5/2 \\ 0 & 0 & 11/2 & 11/2 \end{array}\right]$$

After this transformation, we perform back substitution to solve for our variables. For this project, we are dealing with 8000 x 8000 size matrices and we need a way to check our solution. In order to do this we will calculate a residual vector and use that to get an L^2 Norm. If all goes according to plan, this L^2 Norm should be as close to 0 as possible as this means our error is very small.

## How I am Storing The Data

In this implementation, I decided to make my matrix and vectors contiguous in memory. By mallocing enough space for both the a_matrix, b_vector, and solution_vector, I can be confident that the indexing is correct and efficient when addressing them. I've hardcoded the size of 8000 x 8000 into my program to make it easier on the benchmarking tests. The A matrix and b vector are both randomized after they are allocated to numbers between [-1.0e6, 1.0e6).

## Partitioning the Data, Exploiting Parallelism and Synchronization

Since the matrix and vector we're dealing with is so large, we need to parallelize it in order to speed it up. There are certain parts of this algorithm that need to be done serially in order to keep our data from getting messed up. The pivoting step in particular is important to keep serial. The pseudo code for my parallel section is as follows:

```
for each row k:
        partial pivot();
        parallel section:
                for each row i:
                        figure out multiplier for row;
                        for each column j:
                                matrix[i][j] -= (multiplier * matrix [k][j];
                matrix[i][k] = 0;
                vector[i] -= (multiplier * vector[k]);
```

I decided to do my back substitution and L^2 norm serially, as I couldn't come up with a good way to parallelize those sections of code. The way openMP parallelizes the section of code is that it automatically splits up the for loop to all of the threads available. Since the pivots are done serially, all of the threads are able to perform calculations on the rows in parallel without affecting other rows of the matrix. I think my parallel choices are very clear and readable to other programmers, and the solution is pretty fast even for an 8000x8000 matrix. The synchronization of this program is tied with the shared memory architecture. The matrix and vector are both shared with all of the threads so they can all work on the

same piece of memory as opposed to a distributed memory paradigm where each thread would have to pass messages to other threads in order to share their respective data.

## Running the Benchmarks

I did all of my timing runs on the Manycore Testing Lab. I'm not a big fan of this system. I do like how you can submit all of your jobs at once and it will run them without you needing to monitor it. However it has some serious problems. It seems to run jobs in a LIFO (Last In First Out) order? I don't understand why that decision was made. Also there are only 3 CPUs available to be used at a time. There have been a couple times that these CPUs are being used by other students and qstat shows that they've been going for 7 and 5 hours respectively. Clearly their implementation is awful if it takes 7 hours even for a serial run. This is a lot of wasted time that other students could be using. I'm not sure if everybody knows how to delete jobs as well. I figure others have seen this LIFO order and resubmit jobs without deleting their last ones. There's currently someone with 10 one processor jobs in the queue waiting to be run.

Anyways, here are the results of my timing runs:

Time (seconds) by # of Cores

| 1 | 2 | 5 | 10 | 20 | 30 |
|---|---|---|---|---|---|
| 1894.3 | 978.869 | 398.269 | 239.885 | 321.798 | 284.039 |
| 1894.08 | 979.293 | 397.558 | 244.284 | 223.267 | 218.984 |
| 1897.79 | 976.412 | 467.781 | 431.103 | 224.414 | 284.058 |
| 1899.44 | 978.92 | 396.104 | 438.156 | 223.351 | 417.39 |
| 1898.21 | 976.224 | 465.805 | 237.756 | 222.338 | 217.919 |

L^2 Norm

| 1 | 2 | 5 | 10 | 20 | 30 |
|---|---|---|---|---|---|
| 1.34E-05 | 1.34E-05 | 1.34E-05 | 1.34E-05 | 1.34E-05 | 1.34E-05 |
| 1.34E-05 | 1.34E-05 | 1.34E-05 | 1.34E-05 | 1.34E-05 | 1.34E-05 |
| 1.34E-05 | 1.34E-05 | 1.34E-05 | 1.34E-05 | 1.34E-05 | 1.34E-05 |
| 1.34E-05 | 1.34E-05 | 1.34E-05 | 1.34E-05 | 1.34E-05 | 1.34E-05 |
| 1.34E-05 | 1.34E-05 | 1.34E-05 | 1.34E-05 | 1.34E-05 | 1.34E-05 |

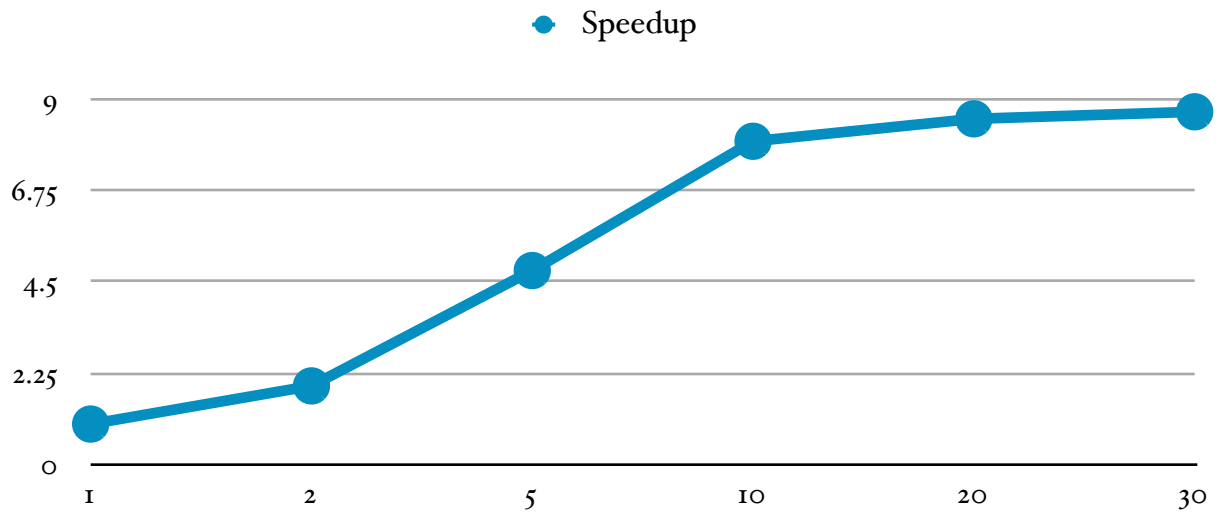
It's a little funny looking that all the L^2 norm values are the same for all my runs. It's okay though because of how the randomized matrices are seeded. This means that each run will get the same "random" numbers so consistency is ensured. This means my solution vectors are the same across runs and thus my L^2 norm is the same. The value is very good and we can say with a high confidence level that my solutions are correct since 1.34E-05 is so close to 0.

## Speedup

The speedup for my program was pretty good. Speedup is a good measure of how much faster our program runs on multiple cores. The higher the number, the better. Here's a table of my speedup values based on the minimums for each core from my timing runs:

Speedup = T1/Tp

| 1 | 2 | 5 | 10 | 20 | 30 |
|---|---|---|---|---|---|
| 1 | 1.940 | 4.782 | 7.967 | 8.520 | 8.693 |

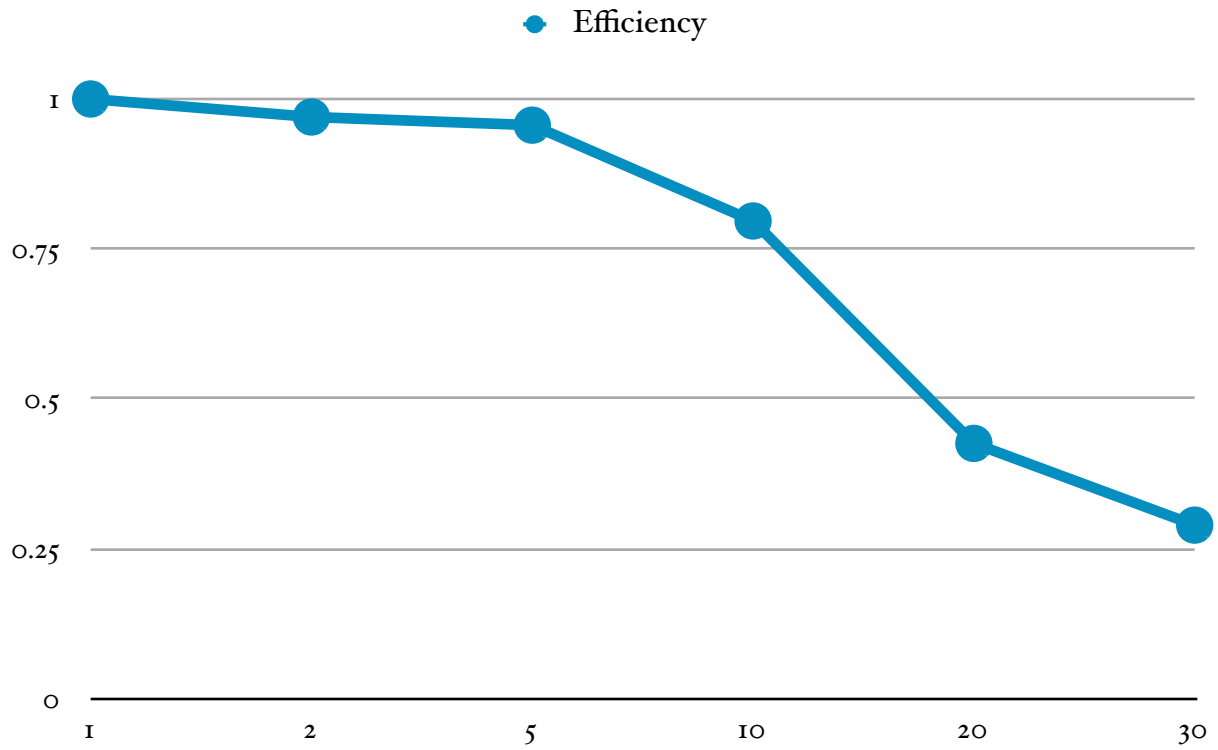


## Efficiency

The efficiency for my project was okay. Efficiency is a measure of how well utilized our program is at making use of parallelism. The closer to 1 the better. Here's a table of my efficiency values based on the minimums for each core from my timing runs:

Efficiency = SU/P

| 1 | 2 | 5 | 10 | 20 | 30 |
|---|---|---|---|---|---|
| 1 | 0.97 | 0.9564 | 0.7967 | 0.426 | 0.2898 |




## Conclusion

This program was a good exercise in shared memory programming. I thought it was kind of a pain in the ass to compete with other students over MTL resources for the timing runs. I did like how MTL showed how things like CPU usage and pagefaults though. Next time I would make more of an effort to do timing way way way early before everybody else tries to do it all at the same time.