# ECE 420
# Lab 3 Report

Cynthia Li, Michael Kwok, Samuel Lojpur

# Description of Implementation

Our goal in this lab was to parallelize a solver for a linear system of equations using OpenMP. We used a basic implementation of Gauss-Jordan elimination and applied lines of OpenMP in order to make a much faster solution of the same problem, when run on a multi-core machine.

Because the focus of the lab is on improving the speed of the algorithm over its serial counterpart, we decided to utilize the existing Gauss-Jordan algorithm provided in the development kit and add on improvement via OpenMP directives.

A basic implementation of a Gauss Jordan elimination algorithm is broken into two parts. Gaussian elimination iterates through each column, then each row in the given matrix looking for the row that has the greatest value in the currently selected column(the Maximization Step). That value is moved so that this large value is along the diagonal (the Pivoting Step), then this row is temporarily multiplied by a certain factor and then subtracted from each row beneath it, in order to ensure that all rows beneath it have a value in the given column (The Elimination Step). This process can then be repeated until the matrix is ‘upper triangular’, that is all values beneath the diagonal in the matrix are zero.

Jordan elimination is then employed to combine the gaussian-elimination matrix, and add the starting vector on the right to make one big matrix, then produce a row where all values not along the diagonal (or on the far right ‘vector row’ are 0, and all values along the diagonal are 1 by subtracting *higher* rows by *lower* ones (and then multiplying the value by whatever ratio makes it one) (The Reduction Step). This changes the values left on the far right column of the given matrix into solutions to the linear system of equations, and can be returned as the solution.

Optimization via OpenMP directives are called in as many places as possible: parallelization of the Maximization Step, the Elimination Step, the Reduction Step and the saving of the solution into the result vector. In the Maximization Step we used each thread to independently search a section of the search space (the currently selected column in the matrix) and compare each maximum value in the end, using a critical section. The Elimination Step was a large subtraction and modification of multiple rows. We chose to split the rows to be eliminated among our existing threads, since each one is a separate array in memory and therefore caused less competition with other threads. The Reduction Step is very similar to the Elimination Step, but since the rows must be reduced from top to bottom, we only parallelize the subtraction of each column, not of each row. Finally, the saving of the far right column to its own memory location to be output is also parallelized.

The biggest time save came from parallelizing the Elimination Step. This single parallelization accounted for 50% of the time saved. Most of the other parallelizations had minimal or non-existent effects on the run time of the program. Other improvements over the existing code were made by removing the use of a proxy index array and instead swap the pointers

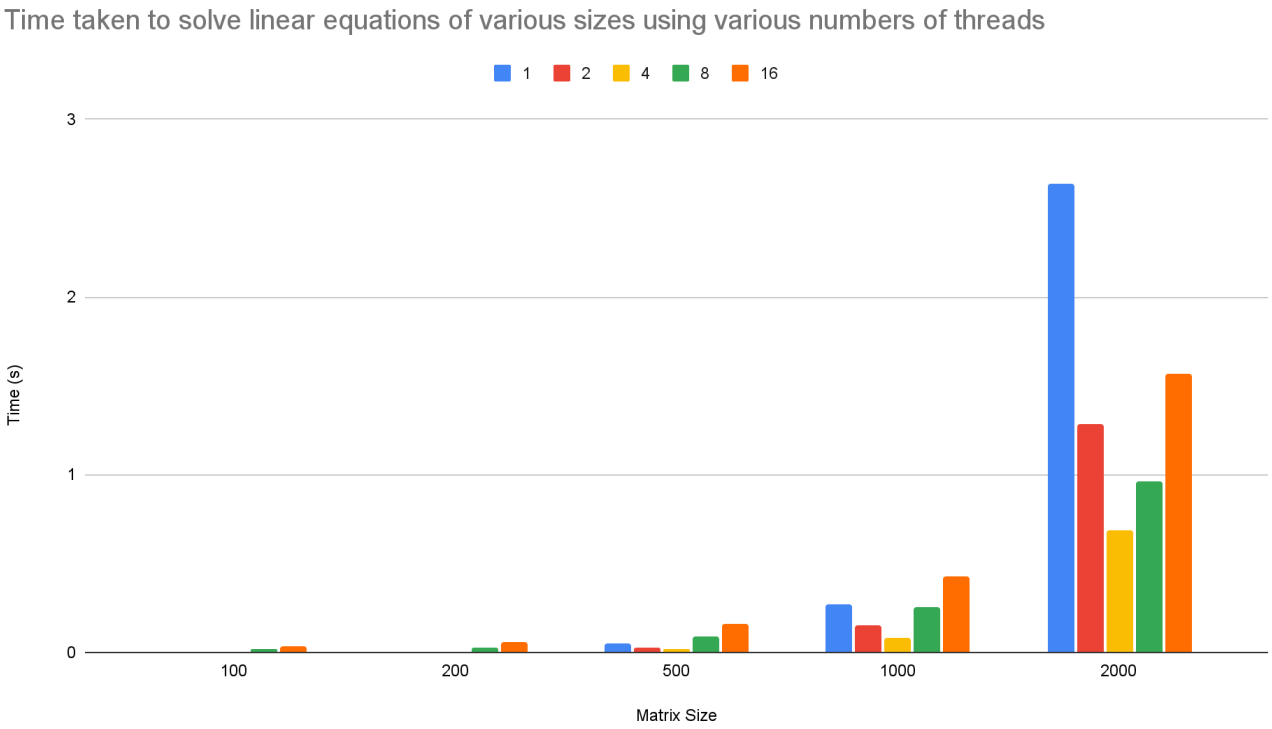
directly in the matrix, in addition to using the absolute method from the standard library instead of squaring the value which reduced a lot of computational overhead.

We attempted to incorporate the collapse directive into the code, but the overhead of collapsing made the Elimination step take much longer than the non collapsed loop in all of our test cases, and there were no other loops that satisfied both the condition of rectangular iteration space as well as the perfect loop. We also looked into parallelizing other existing loops but they needed to be done in a specific order, or else were dependent on the results of the previous iteration. We also attempted to keep the threads used for one process (such as the Maximization step) open to reuse them in a later step (the Elimination Step) but this proved to be inefficient, especially since the Pivoting Step between them had to be performed with a single thread, and thus required all other threads to wait on that.

# Performance Discussion

In the serial program the execution time for a 200x200 matrix was approximately 0.02s. When all parallelizable sections were optimized, the time was cut down to 0.006s. This is a speedup of over 3x over the single threaded case (on matrix sized > 500, using 4 threads), which is significant enough that we believe that our improvements worked.

| **Threads\Matrix Size** | **100** | **200** | **500** | **1000** | **2000** |
|---|---|---|---|---|---|
| **1** | 0.000892 | 0.004055 | 0.048125 | 0.273016 | 2.636748 |
| **2** | 0.001244 | 0.003464 | 0.027573 | 0.155156 | 1.281261 |
| **4** | 0.001134 | 0.002713 | 0.015978 | 0.077833 | 0.689265 |
| **8** | 0.016528 | 0.028641 | 0.085433 | 0.256167 | 0.960759 |
| **16** | 0.030796 | 0.057395 | 0.15824 | 0.425572 | 1.569492 |
| **32** | 0.063836 | 0.115217 | 0.282403 | 0.706157 | 2.096789 |




The above graph illustrates the benchmark of our algorithm against different numbers of threads and different matrix sizes. The runtime of the algorithm is directly proportional to the matrix size; however, the optimal number of threads varies depending on the size of the matrix. At lower sizes, fewer threads provide a benefit over many threads due to overhead such as thread creation and data contention. However, as the data set grows larger, a clear trend emerges in that there exists an optimal number of threads such that the trade off of performance degradation of having too many threads is balanced with the performance benefits of dividing work among multiple threads.