ECE 420 Lab 3 Report

Andrew Juwon Park - 1442834
John Tyler Beckingham - 1500702
Zhijie(Max) Shen - 1494084

Description of Implementation:

In our main() code, we initially get the matrix values by calling Lab3Loadinput which was given to us. Our main operation is divided into three different sections: Gaussian elimination, Jordan elimination, and getting results.

Starting with the Gaussian elimination, we try to eliminate the elements below the diagonal to zero one column. First, we will iterate through the rows from i to end and find the maximum absolute value on the i'th column. Then, we switch the row with the highest absolute value with the i'th row. Then, for all of the rows below the i'th row, we multiply the values so that they equal to the value on the column of the i'th row and subtract them so it becomes 0. For loop iterates through i and then we end up with a matrix that has all 0s below the diagonal line.

For Jordan Elimination, the process is more simple. We iterate through the matrix starting from the furthest right column k, and inside another for loop, we iterate through each row of the column and subtract the last column k in the last row k. Then, it goes to the 2nd furthest row / column from the right. This process repeats and eventually, we are left with a matrix with values only on the diagonal.

For the last step, we simply divide the row, i, of the resulting matrix by the value at (i, i) in the matrix.

Attempted Methods of Optimization:

Here are some of the optimization methods we have tried:

1. Adding parallelization to the pivoting operations
2. Adding parallelization to loops that perform expensive operations (multiplications and divisions)
3. Adding parallelization to the loop that fetches the resulting vector

We found that adding parallelization to simple operations (pivoting and fetching the resulting vector) increased the running time. We measured the time with and without parallelization for each of those loops and noticed 2 things:

1. The first iteration, when parallelized, takes significantly longer than the rest of the iterations.
2. The for loop, when parallelized, was slower than the none-parallelized for loop.

For the first case, when omp parallel is executed, it forks and creates the threads. This could explain why the first loop took significantly longer than the other iterations.
For the second case, we need to take into account the time for the thread to join after it finishes processing. This could explain why the parallized calculation took longer than the unparalleled calculation.

On the expensive operation loops, we found that parallelizing did result in a significant speedup. Our final solution with the best speedup was to parallelize the for loops performing the Gaussian elimination and the Jordan Elimination.

Performance Discussion

The following table is the time for the Jordan-Gaussian elimination for differing numbers of threads and matrix sizes for our best implementation, averaged over 5 runs on the lab machine:

Matrix Size / # of Thread	**1**	**5**	**10**	**15**	**20**
100	0.002151	0.012824	0.019918	0.026766	0.030652
500	0.205596	0.277824	0.264812	0.275682	0.312489
1000	1.410506	1.346048	1.259213	1.286041	1.391086
1500	4.792601	4.259245	3.664867	3.451965	3.494410
2000	11.307834	9.073328	7.932383	7.525235	7.517152

Our best speedup of 1.504 happens when we use a large data input and many threads, specifically, the 2000 X 2000 matrix with 20 threads. This is because on larger input sizes, each thread can handle more computation, reducing the amount of serial computation, and each additional thread is still helpful as the existing threads still have a lot of work.

Some interesting things to note are that in the cases where the matrices were small, parallelizing only makes the performance worse. This is because there is significant overhead involved if forking and joining, as well as scheduling, and the amount of work done in parallel by these threads is not large enough to compensate. This is shown the most directly in the 100 X 100 matrix case, where adding any parallelization at all results in a dramatic slowdown, and increasing the number of threads slows it down even further. For example, increasing from 5 to 10 threads in the size 100 case only reduces the workload of each thread by 10 operations, while also requiring the system to handle the overhead of 5 more threads (very bad tradeoff).

Parallelizing at all only seems to become useful for matrices of size >= 1000, and increasing the number of threads up to 20 was only useful for matrices of size >= 2000. Ignoring the specific numbers, this pattern is generally expected, as parallelizing for performance would only be useful for input sizes large enough to overcome the overhead, and adding more threads generates decreasing returns while adding linear amounts of overhead. We would expect that increasing the number of threads up to 50 or 100 in the size 2000 case would also begin to make the performance worse due to this pattern.