# CMPT 431 Project Report: Parallel Linear Equation Solver Implementation Using C++ Threads and MPI

**Group 8**


| Name | Student Number | Email |
|---|---|---|
| Curtis Lui | 301304667 | cwlui@sfu.ca |
| Adam Liang | 301360695 | ala193@sfu.ca |

# Introduction

Solving a system of equations is considered an $O(n^3)$ problem. Our approach to solving a system of linear equations with 1000 variables is Gaussian elimination followed by back substitution. The algorithm naturally leads itself to parallelism as reducing all the rows below the pivot row can be done in parallel. We have applied different row mapping techniques to see the overall effects and speedup with threads and MPI.

# Background

All implementations compile successfully and produce a correct result that is verified to be within 0.5% error of the actual answer. There are 6 working versions of our linear equation solver:

1. Serial version
2. Parallel - Static Mapping
3. Parallel - Dynamic Mapping
4. Parallel - Dynamic Equal Mapping
5. Distributed - Static Mapping
6. Distributed - Dynamic Equal Mapping

Each implementation sees a speedup when increasing the number of threads or processes.

# Implementation Details

## Serial

The serial implementation performs Gaussian elimination to obtain the row echelon form of the matrix followed by back substitution to solve the variables. All operations done to the matrix are performed serially. During forward elimination, the pivot row and column is checked to see if the value is 0. If it is 0, a row below the current pivot whose value is not 0 will be swapped. If there is no such row, then the matrix is singular.

## Parallel

The parallel versions differ in how the rows below the pivot row are calculated. In all versions, row swapping and back substitution is done by the root thread. Also, two barriers have been

added to synchronize the threads. One barrier is used for the row swapping and the other is used to ensure all threads wait until all the rows below the pivot have been eliminated.

The static mapping (`--strategy 1`) implementation distributes all rows of the matrix equally amongst the threads once before computation begins. This means that for every pivot, every thread performs computation at their assigned start row up to the respective end row. For a thread where the pivot row is below their end row, no computation is done since that thread's assigned rows are in row echelon form.

The dynamic mapping (`--strategy 2 --granularity k`) implementation assigns rows when requested. The number of rows that are assigned for each request is the granularity value, k. The granularity value can be set as an input argument when running the program. Each thread computers the rows assigned until all rows below the pivot for that iteration are processed.

The equal mapping (`--strategy 3`) implementation dynamically assigns rows for every pivot iteration. In each iteration, all the rows below the pivot are distributed equally to every thread, any remainder is assigned to the last thread. Essentially, this strategy will statically map the rows 999 times.

## Distributed

The distributed program uses the static strategy similar to strategy 1 in the parallel program to minimize the communication required between mpi processes. All rows of the matrix are equally distributed amongst the process before computation begins. For each pivot, every process performs elimination for each row from their start to end row. Processes who's end row is before the pivot row will exit. For each pivot, the process that contains the pivot within its start and end rows broadcasts it to processes that have a higher world rank than it (created a vector of custom mpi communicators that only include processes with world rank greater or equal to the index before computations). Since synchronization is not required between the different sections of the matrix for gaussian elimination an asynchronous broadcast is used. To reduce the amount of data sent in the broadcast only the non-zero elements of the pivot are broadcasted. Since the end result is assumed to be an upper triangular matrix, all of the bottom left values are not looked at and treated as 0.

## Verification Details

For all implementations, the resulting vector is verified by substituting the result into the original set of equations, $A\overline{x} = \overline{b}$. The result is then asserted to check if the result is within 0.5% of the original value of $\overline{b}$. The result is verified by the following codeblock:

```
for (int i = 0; i < size; i++) {
    double sum = 0;
    for (int j = 0; j < size; j++) {
        sum += mat[i][j] * x[j];
    }
    assert(abs(mat[i][size] - sum) / mat[i][size] <= 0.005);
}
```

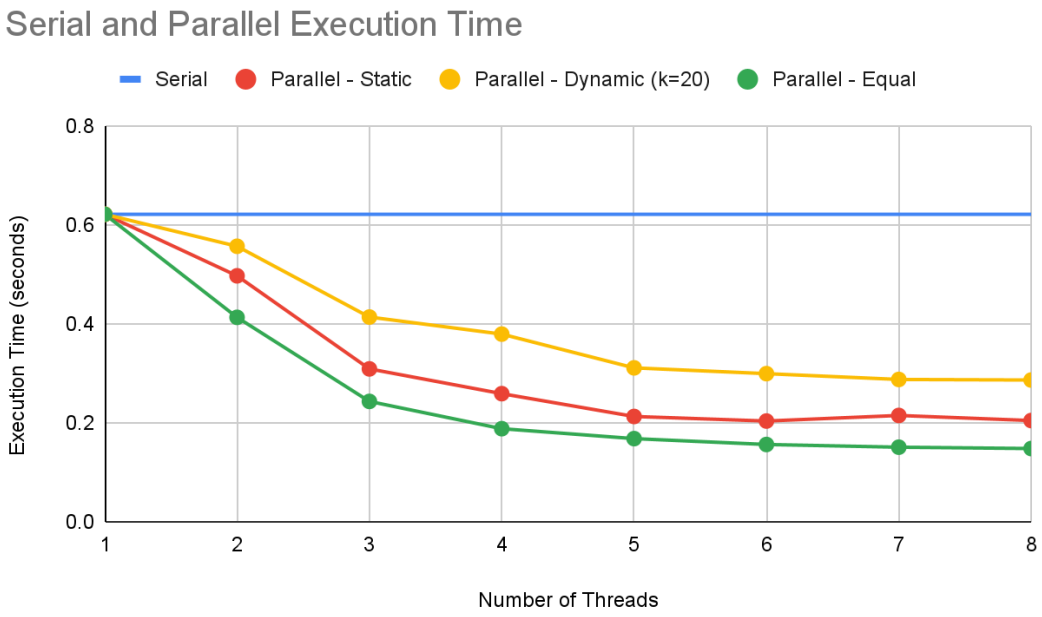
# Evaluation

For all tests we used the same 1000 × 1000 and took 10 timed tests to obtain an average solve time. Since the parallel and distributed versions were derived from the serial implementation, it is natural that the run time for the serial version is our control and benchmark. The average solve time for the serial implementation was 0.622 seconds.

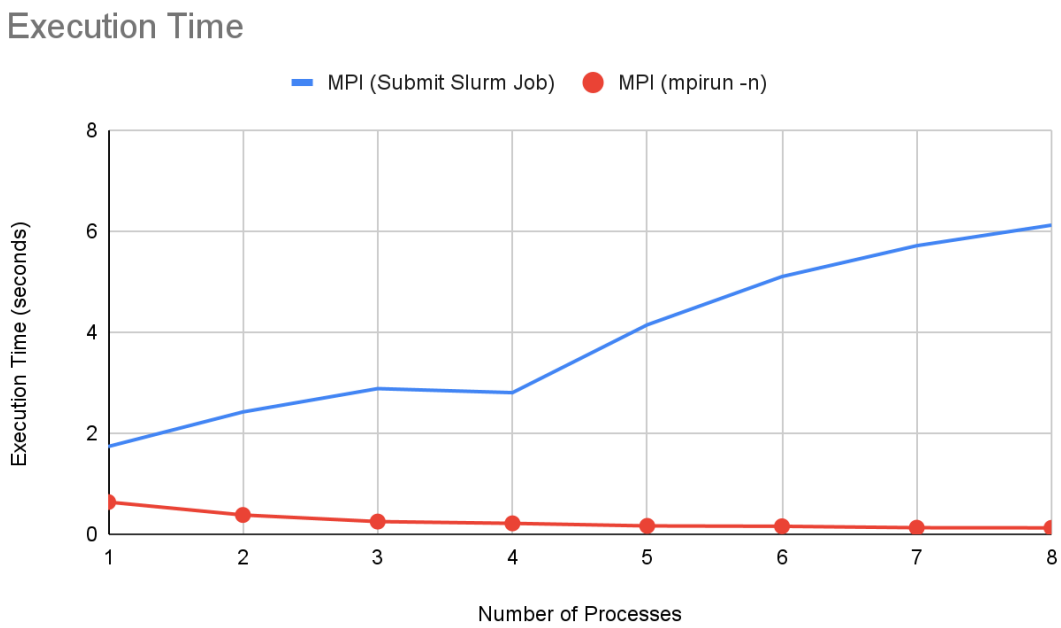
The parallel version with static mapping had decreasing solve times as the number of threads increased. At around 5 threads, the solve times plateaued to around 0.2 seconds. After investigating the barrier times, we confirmed that the constant 0.2 seconds is due to the communication with the two barriers used to synchronize the threads. Also, due to the nature of the mapping, certain threads will be performing no work once the pivot row is below their assigned end row.

The parallel version with dynamic mapping was run with a granularity value of 20 and also had a decreasing solve times as the number of threads increased. However, the decrease was not as great as the static mapping. Again, at 5 threads the solve times plateaued at around 0.29 seconds. The total number of rows that each thread computed and the barrier times was consistent across all threads. We also identified that the slowdown compared to the statically mapped version was due to the extra communication required when threads requested rows to work on.

The equally mapped parallel version performed the best out of all threaded implementations. Similar to the other versions, the speedup started to plateau at 5 threads which had a solve time of about 0.15 seconds. The time spent at the barriers is decreased since all threads are given equal amounts of rows to work on for every pivot. The speedup compared to the dynamically mapped version is due to the removal of communication for requesting work. Compared to the statically mapped version, the speedup comes from having all threads performing work during all pivots.




The mpi version is very dependent on the communication overhead. When submitted as a slurm job, tasks can be distributed on machines with high communication times, thus causing communication overhead to be greater than the speed up. However, when communication overheads are low as shown in the "`mpirun -n`" cases, there is a very good speed up. For example, the speedup with 8 tasks using mpirun = 0.638924/ 0.127741 = 5.002



Evaluation Data Spreadsheet:
https://docs.google.com/spreadsheets/d/1lmC6oXf_PoHKwDa_nfLsKZqiZskdh44K0pt3phDWK6M/edit?usp=sharing

# Conclusion

For a linear equation solver of size 1000 × 1000, threaded approaches work the best because the entire matrix can be loaded into memory and efficiently shared between threads. Threaded programs have low overhead relative to distributed programs, and thus are less impacted by the low serial time (~0.6 seconds) of the problem. For a given problem size of 1000 × 1000, the mpi approaches submitted as slurm jobs do not benefit from using distributed computing due to the overhead of communication between nodes being higher than the work being distributed.

For much larger linear equation problems, the distributed solution can scale further than the threaded solution. Since the distributed solution can work on multiple machines, it can take advantage of more processors than the threaded version. The distributed solution can also handle linear equation problems that are too large to fit on the memory of one machine, with each distributed process only reading its initially assigned chunk of the matrix.