Marcantonio Soda
HW3 Report

- Background on Gauss

  The Gaussian elimination algorithm is used in many applications for solving systems of linear equations. Gaussian elimination is a fundamental algorithm in computational mathematics. Its importance stems from its versatility in solving systems of linear equations, which is a frequent problem in scientific, engineering, and mathematical contexts. The algorithm transforms the system of linear equations into an equivalent system in upper triangular form, from which solutions can be easily obtained via backward substitution. There are two main steps to the procedure: forward elimination and backward substitution.

  The purpose of forward elimination is to reduce the system to an equivalent matrix in upper triangular form. The process involves taking each row and, first, choosing a pivot element from the leading entries in the current row, followed by using row operations to create zeros below the aforementioned pivot element, then move onto the next row and repeat until the upper triangular form is reached. At which point, the backward substitution can be applied to solve the system.

  The purpose of backward substitution is to find the values of the variables. The process involves first solving for the last variable (because it is the simplest due to the nature of upper triangular matrices), then use that solution to solve for the variable in the next equation, then repeat until all solutions have been found.
- Parallel Techniques

  I took note of four places that the algorithm would potentially benefit from parallelism given some constraints on the system being solved. Finding the pivots, normalizing the pivot rows, eliminating below the pivot rows, and backward substitution as a whole.
- `find_pivot` and `par_find_pivot`:

  Starts from the current `pivot_row` and iterates over all rows below it in an attempt to find the the row with the maximum absolute value in the column of the `pivot_row`. The goal is to find the best pivot in an effort to avoid numerical instability.

  The parallel counterpart uses rayon to parallelize the search for the max absolute value from the remaining rows. It will make finding the pivot faster, but only for very large matrices.
- `normalize_pivot_row` and `par_normalize_pivot_row`:

  After the pivot row is found, this divides all of the elements in the pivot row by the pivot value, which ends up setting the pivot to `1`. It also adjusts `b` accordingly.

  The elements in the pivot row of `a` are divided by the pivot value in parallel. Again, this will only cause a speedup for large systems.
- `eliminate_below_pivot` and `par_eliminate_below_pivot`:

  For every row below the pivot, this adjusts its elements to eliminate (making it 0) the corresponding value in the pivot column. It does this by subtracting a multiple of the pivot row from the current row. `b` is also adjusted accordingly.

The parallel implementation of this will likely result in vast speedups for matrices that are relatively small compared to the ones required for the previous two parallel functions to be useful. Each thread handles the elimination of a separate row.

- `backward_substitution` and `par_backward_substitution`:

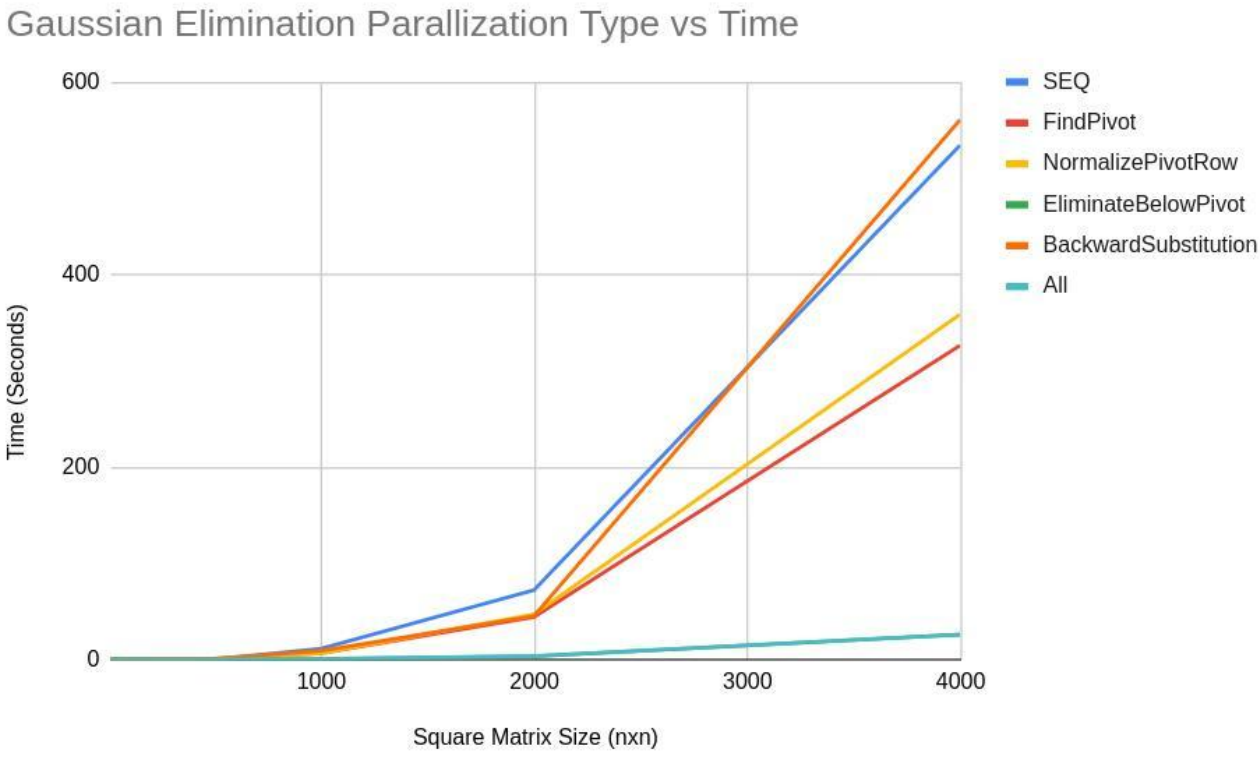
  This finds the values of the variables in the system. The process involves first solving for the last variable (because it is the simplest due to the nature of upper triangular matrices), then use that solution to solve for the variable in the next equation, then repeat until all solutions have been found.

  Its parallel counterpart distributes the summation task to multiple threads, making it faster for very large matrices.

- Benchmarks:

  `lib.rs` contains functions for one parallel and one sequential version of each of the four chunks of code that I wanted to parallelize. It contains only one `gauss` function which takes a `vector` of `Par`s. `Par` is an enum that I created to represent each of the four parallelized methods, it can be either `FindPiv`, `NormPivRow`, `ElimBeforePiv`, or `BackSub`. The `Par` vector may contain any combination of `Par`s. If a particular `Par` is present in the vector, the parallel version of the function will be called, otherwise, the sequential version will be called. This allows me to rapidly mix and match which combination of parallelized algorithms I want to call and reduces code duplication entirely. I love Rust.

  I benchmarked 6 combinations of the parallel functions. One in which only sequential methods are called, one benchmark for each parallel method in which only that parallel method is called and the rest are sequential, and one in which all parallel methods are used. For each of the 6 methods, 9 matrix sizes are benchmarked, 10x10, 25x25, 50x50, 100x100, 250x250, 500x500, 1000x1000, 2000x2000, and 4000x4000. The array elements are randomly filled.



- Analysis:

  Based on the figure, it is clear that parallelization techniques make Gaussian Elimination orders of magnitude faster for large systems. I expect that this trend will grow for even larger systems. Generally, for smaller systems, the sequential variant was able to find the solution faster because of the overhead required for spawning and managing threads. All of the parallelization types outperformed the sequential variant for square matrices of around size 250x250 and above. The only exception is BackwardSubstitution which seemed to slightly underperform compared to the sequential version for large systems. I would be willing to bet that it would overtake sequential for even larger matrices. However, I do not have the time to test this.

  The place where most of the speedup happens is within the parallelization of EliminateBelowPivot (which I predicted). This elimination step is the most computationally intensive, and it is very parallelizable, so it stands to reason that most of the speedup would happen here.

  Overall, the results are exactly what I expected.