Evaluation of Parallel Gaussian Elimination Implementations

Bonor Ayambem

CSE-411: Advanced Programming Techniques

Prof. Corey Montella

October 26, 2023

1. Introduction

Gaussian Elimination is a technique used to solve a system of linear equations of the form Ax = B. A sequence of operations is performed on the augmented matrix of A and B to reduce it to its row echelon form and hence produce values for the column vector of variables x.

Gaussian elimination can be split into two parts namely, forward elimination and backward substitution. Throughout, elementary row operations of swapping two rows, multiplying a row by a nonzero number, and adding a multiple of one row to another row are performed on the augmented matrix.

This project is concerned with taking a sequential algorithm for Gaussian elimination, and parallelizing parts of it. Our goal is to evaluate how parallelism affects the performance of an algorithm such as this on matrices of different sizes.

2. Implementation Summaries

Three out of the four implementations make use of the rayon Rust library while the final implementation was completed using threads. Rayon provides data parallelism for performing parallel computations on collections, which is especially useful in a project such as this that deals with a collection of numbers. Threads are concurrent execution units that allow us to write parallel and concurrent programs, and by using them in this project we are able to compare how they affect performance versus rayon.

2.1 Parallel Backward Substitution

This implementation parallelizes the backward substitution part of Gaussian elimination, and it does this by calling `.into_par_iter()` on the reversed range from 0 to n, replacing the

for loop that handles back substitution in the sequential implementation. Hence, during back substitution, operations are carried out on each reduced row concurrently.

Since `x` is updated in the closure, `x` needs to be wrapped in an `Arc` and `Mutex` when it is first declared in the function. This also differs from the sequential implementation which does not wrap `x`.

2.2 Parallel Forward Elimination Sections

This implementation parallelizes two sections within the forward elimination part of the gaussian code. Both the section to find the pivot element and the elimination loop itself are done concurrently for each row that they operate on. This is achieved using `rayon` functions.

To work around the ownership issues that can arise in parallel programming, `a` and `b` are wrapped in `Arc` and `Mutex`, and then unlocked every time they need to be accessed. This differs from the sequential implementation which does not wrap `a` or `b`.

2.3 Parallel Full Forward Elimination

All of the operations done as part of forward elimination are done concurrently using rayon for each row. To work around the ownership issues that can arise in parallel programming, `a` and `b` are wrapped in `Arc` and `Mutex`, and then unlocked every time they need to be accessed. This differs from the sequential implementation which does not wrap `a` or `b`.

2.3 Using Threads

Parallelism here is achieved by leveraging multiple threads to perform tasks concurrently. The input matrices 'a' and 'b' are wrapped in Arc and Mutex to allow safe, shared access across threads. The parallelism is introduced in two key areas: first, in the elimination of the pivot element and the row swapping, and second, in the elimination of variables below the pivot row.

Threads are spawned to handle these operations, allowing multiple rows to be processed simultaneously.

3. Testing Setup

3.1 Testing

To ensure that all the implementations are solving the equations as expected, tests were run for each of them. For:

```
a = [
        [2.0, 1.0, -1.0],
        [-3.0, -1.0, 2.0],
        [-2.0, 1.0, 2.0],
    ];
```

and

```
b = [8.0, -11.0, -3.0];
```

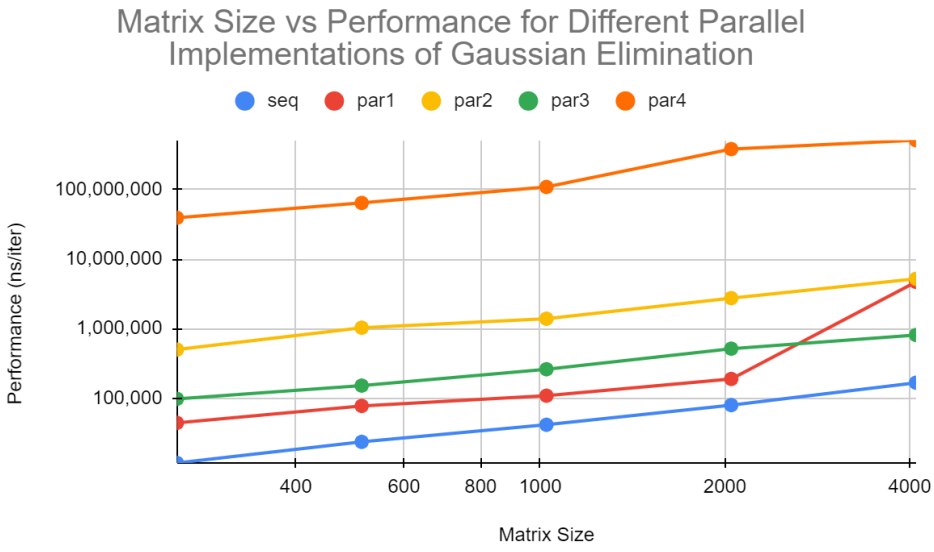
The expected result is `[2.0, 3.0, -0.9999999999999999]`, and each implementation worked as expected and passed the tests.

3.2 Benchmarking

Benchmarks are run for matrix A sizes of 256, 512, 1024, 2048, and 4096 to evaluate the performances of the different implementations.

4. Results

As the matrix size increases, the execution time for the sequential implementation also increases, which is expected.



The "par1," "par2," "par3," and "par4" benchmarks represent the different attempts at parallelizing the Gaussian elimination, using various threading and synchronization techniques. It's evident that the parallel implementations are significantly slower than their sequential counterpart, likely due to overhead introduced by thread management and synchronization.

For the first parallel implementation, there's a notable increase in execution time as the matrix size grows, suggesting that this approach does not provide a significant performance benefit for larger matrices. The second parallel implementation exhibits similar behavior, with even longer execution times compared to the first indicating suboptimal parallelization.

The third parallel implementation shows improved performance over the first and second for matrices of 1024x1024 and 2048x2048. It's a more effective parallelization approach for those sizes. The fourth implementation showcases extremely long execution times, likely due to an inefficient or incorrect parallelization strategy.

5 Conclusions

The sequential implementation shows an expected increase in execution time as the matrix size grows. This is because larger matrices require more computations, resulting in longer execution times.

The results show that not all parallelization attempts lead to improved performance. In some cases, the parallel implementations are slower than the sequential version due to the overhead introduced by threading and synchronization. The performance of parallel implementations is also highly dependent on the size of the matrix. Some parallelization strategies perform well for smaller matrices but become less effective as the matrix size increases.

The third strategy which parallelizes the entire forward elimination for each row using rayon, shows promise with improved performance for certain matrix sizes. The implementation using threads stands out as highly inefficient, with excessively long execution times. This result indicates that not all parallelization strategies are effective, and some may introduce severe performance penalties.

These results highlight the trade-offs involved in parallelizing algorithms. The benchmark results illustrate the complexity of parallelizing Gaussian elimination and emphasize the need for tailored parallelization approaches that consider the matrix size and the trade-offs associated with concurrency and synchronization.