Report

Abhay Vivek Kulkarni (ak6277)

The algorithm for my implementation is based on the OpenMP implementation of the gaussian elimination:

Sequential code:

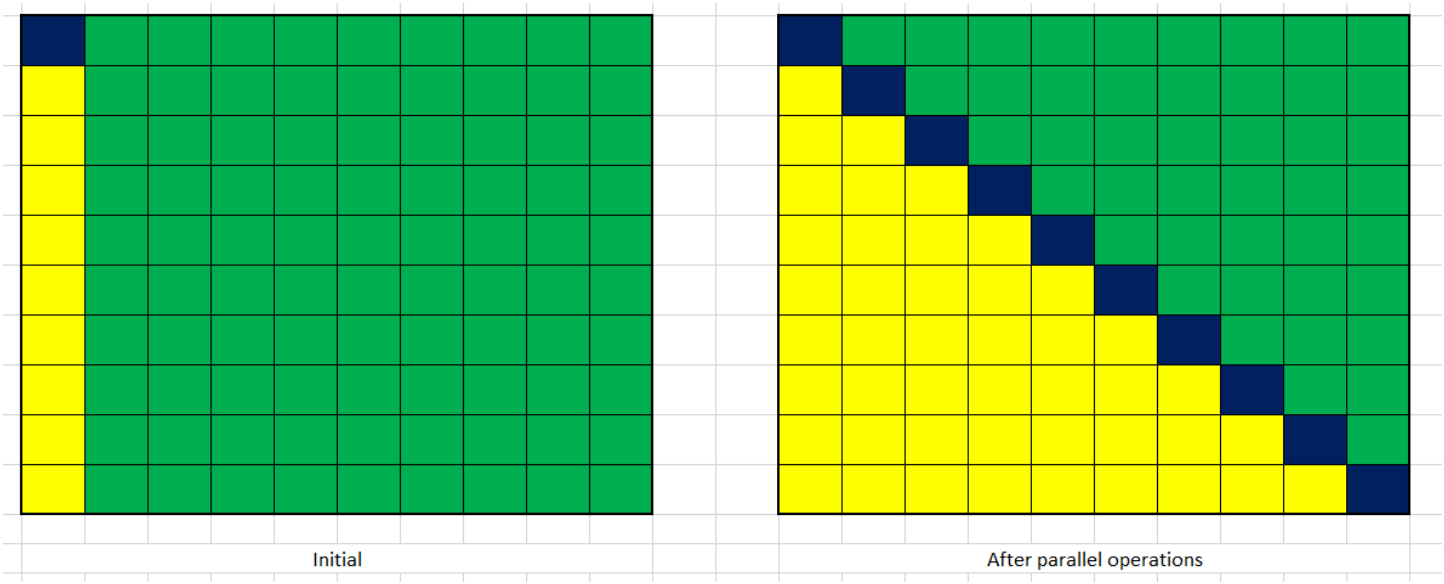
```
for (int i = 0; i < N-1; i++) {
    for (int j = i; j < N; j++) {
        double ratio = A[j][i]/A[i][i];
        for (int k = i; k < N; k++) {
            A[j][k] -= (ratio*A[i][k]);
            b[j] -= (ratio*b[i]);
        }
    }
}
```

OpenMP implementation:

```
for(int i=0;i<N-1;i++){
#pragma omp parallel for
    for(int j=i;j<N; j++){
        double ratio=A[j][i]/A[i][i];
        for(int k=i;k<N; k++){
            A[j][k]-=(ratio*A[i][k]);
            b[j]-=(ratio*b[i]);
        }
    }
}
```

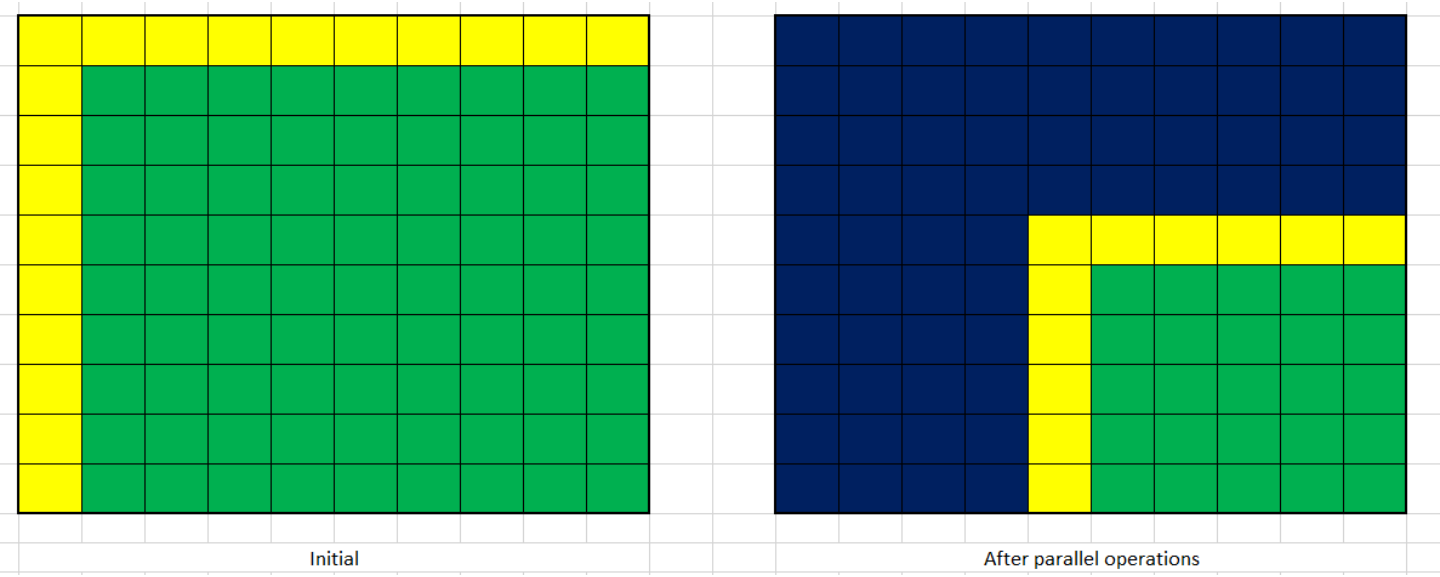
(https://www.cs.rutgers.edu/~venugopa/parallel_summer2012/ge.html)

The aim is to solve the (Ax = b) Linear equations:



All the column elements under the pivot (diagonal) line will have to be 0 for the linear equations to solve by Gaussian Elimination.

To parallelize this:



The yellow part is used for calculating the division factor.
The operations will be performed on the green part; blue part is the completed matrix.

The only part of the parallelizable code is performing operations on the green part.

Pseudo code:

```
for each row:
    get pivot element
    swap pivot with column max. value
    normalize the row w.r.t the diagonal element
    for each element in the lower green part:
            allocate partitions to threads
            perform operations parallelly
            wait for all
```

Implementation:

I used:
ExecutorService es = Executors.newFixedThreadPool(threads);
to manage the thread.start() and thread.join() operations.

```
futures[innerRow] = es.submit(() -> {
    for (int innerRow_ = begin; innerRow_ < end; innerRow_++) {
        double innerValue = matrixA[innerRow_][rowD];
        for (int innerCol = rowD + 1; innerCol < size; innerCol++) {
            matrixA[innerRow_][innerCol] -= innerValue * matrixA[rowD][innerCol];
        }
        vectorB[innerRow_] -= matrixA[innerRow_][rowD] * vectorB[rowD];
        matrixA[innerRow_][rowD] = 0.0;
    }
});
```

`java.util.concurrent.Future` is used to handle asynchronous tasks performed by the thread pool managed by the executor.

Once the entire inner part operation is finished; we can do futures[innerRow].get()
It waits for the threads to finish and then the result is used.

After the outer row has done traversing; we get an upper triangular matrix.
Perform Back-substitution on the matrix to obtain results.

Issues faced:

When selecting the pivot element; get the maximum element from the pivots' column

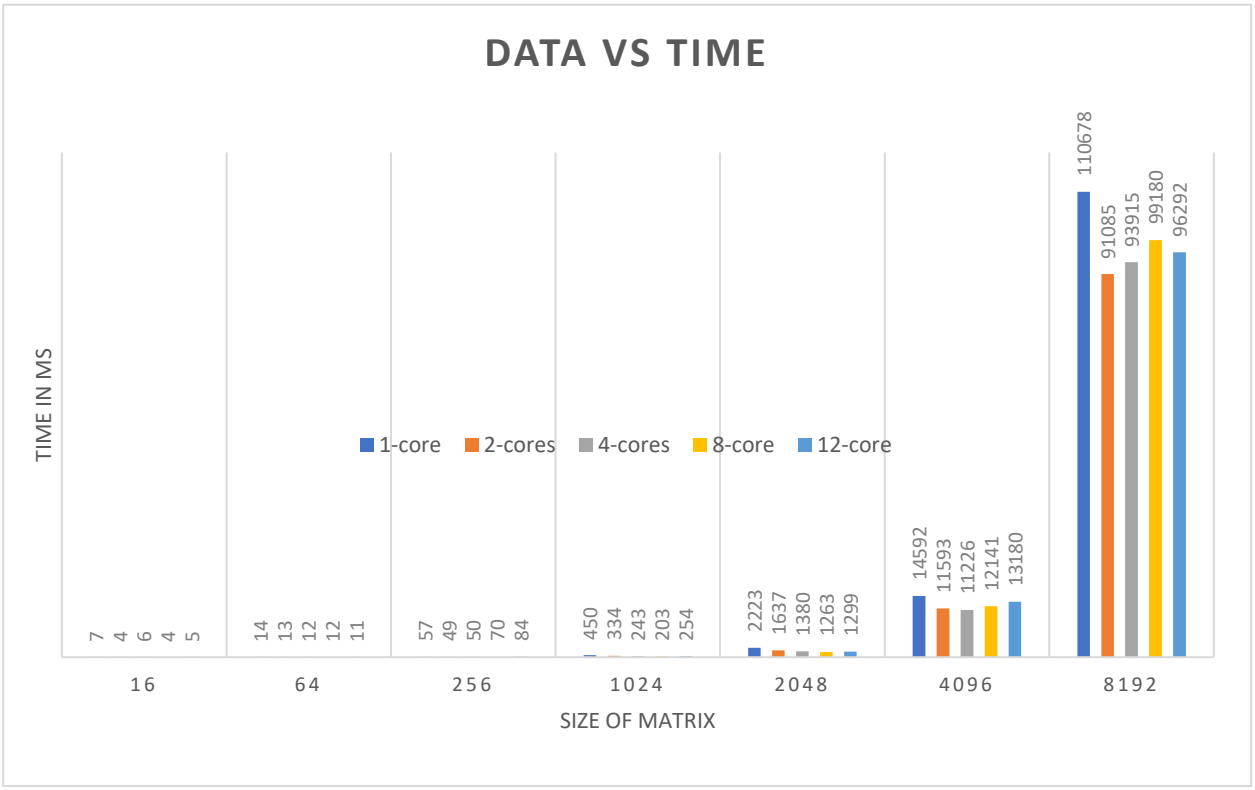
a) It makes sure the pivot is a non-zero number, this prevents DivideByZero errors.
b) Provides the best normalized row elements for further divisions.

Swap the rows before allocating the row ranges to the threads. Prevents race conditions and makes sure the transactions are atomic.

Results :

To check the efficiency of the algorithm; the program iterates through two lists:

```
for matrix size in [16, 64, 256, 1024, 2048, 4096, 8192]:
        for number of Threads in [2, 4, 8, 12]:
                sequentialTime = 0; parallelTime = 0;
                for I in range(20):
                        Create a random matrix
                        perform sequential elimination
                        perform parallel elimination
                        match results
                        log time
                        add times to the counters
                return sequentialTime/20, parallelTime/20
```



(Maxed out the threads at 12 as this was done on my laptop which has 12 cores; Time is in 'ms')

We start seeing significant changes on larger datasets.

Additional runs were conducted on kraken. The results cap at 2048x2048 size matrix due to a resource limit. Threads have a range (2-100)

Observations :

Based on the results from performing multiple iterations on various data sizes and resources:

The parallel code is a poor choice for low-size datasets.

At a certain point, the parallel implementation shows some significant improvements over the sequential; but as we keep on increasing the number of threads the extra overhead cancels out the performance gained by parallel implementation.