Raul L Bernal-Gonzalez
Alonzo Castanon

Final Report

COMP-605
May 4, 2024

Parallelization of LU Decomposition

LU-decomposition is a popular method of solving a linear system of equations with out having to invert a matrix. Any invertible matrix A can be factorized into an lower and upper triangular part. These L and U matrices can then be used to solve the $A\vec{x} = \vec{b}$. This method is illustrated in the following:

$$A\vec{x} = \vec{b} \tag{1}$$
$$(LU)\vec{x} = \vec{b} \tag{2}$$
$$L\vec{y} = \vec{b} \tag{3}$$
$$\vec{y} = L^{-1}\vec{b} \tag{4}$$
$$\vec{y} = U\vec{x} \tag{5}$$

Two popular algorithms for LU decomposition are Crout's and Gauss-Elimination.

1 Crout's Algorithm

This method solves for the upper triangular part of the LU decomposition in a "⌟" shape. It first computes the upper triangular part then then uses those values to create the lower triangular part. The algorithm iterates through columns of the matrix, going down each row of the column. The algoritms solves equations 6 and 7. This method is depicted in figure 1. The algorithm is "in place" with each iteration replacing the data in A with the LU decomposition. This is illustrated in equation 8

$$\beta_{ij} = a_{ij} - \sum_{k=1}^{i-1} \alpha_{ik}\beta_{kj} \tag{6}$$
$$\alpha_{ij} = \frac{1}{\beta_{jj}}\left(a_{ij} - \sum_{k=1}^{j-1} \alpha_{ik}\beta_{kj}\right) \tag{7}$$

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \rightarrow \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ \alpha_{21} & \beta_{22} & \beta_{23} & \beta_{24} \\ \alpha_{31} & \alpha_{32} & \beta_{33} & \beta_{34} \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \beta_{44} \end{bmatrix} \tag{8}$$

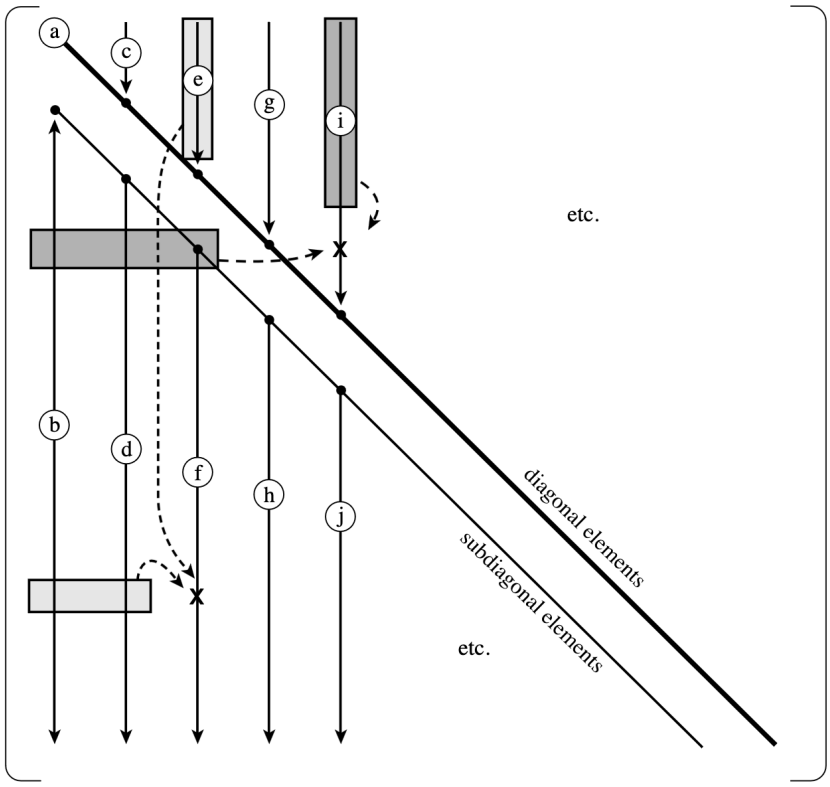

Figure 2.3.1. Crout's algorithm for *LU* decomposition of a matrix. Elements of the original matrix are modified in the order indicated by lower case letters: a, b, c, etc. Shaded boxes show the previously modified elements that are used in modifying two typical elements, each indicated by an "x".

Figure 1: Numerical Recipes in C 2nd Edition

1.1 Parallelization OpenMP

We decided to apply parallelization to the Numerical Recipes version of the Crouts's algorithm. The parallelization of this algorithm is fairly straight forward with OpenMP. We parallelized over the inner loops that go through the rows of each column. This was the best way to parallelized with OpenMP because it minimizes communication between threads. If we had parallelized over the main loop that iterates over the columns, we would have had to pass messages between threads because we need the all the columns $< j$ to build the j^{th} column. The code for the openMP parallelization is shown below.

Listing 1: Crout's algorithm parallelize with OpenMP. Adapted from Numerical Recipes

```
#pragma omp parallel for default(none) private(i, big) shared(scaling, a, n)
for (i = 0; i < n; i++) {
// Find the max abs(value) of each row and detect if matrix is singular
    big = a.row(i).cwiseAbs().maxCoeff();
    if (big == 0.0) {
        printf("Singular matrix in routine ludcmp");
        exit(1);
    }
    scaling[i] = 1.0 / big;
```

```
10 }
11 // main crout's loop. iterate by column
12 for (j = 0; j < n; j++) {
13     // build upper trianfular part up to j-1 row since eache thread will have
       ↪ its own set of i's,
14     // no overlap the a(k,j) term would have already been build in previous
       ↪ iterations of j
15 #pragma omp parallel for default(none) private(i, k, sum) shared(a, j)
   ↪ schedule(dynamic)
16     for (i = 0; i < j; i++) {
17       sum = a(i, j);
18       for (k = 0; k < i; k++)
19         sum -= a(i, k) * a(k, j); // we are not reducing to sum
20       a(i, j) = sum;
21     }
22     // continue to build upper triangular part from i = j up to n rows
23  #pragma omp parallel for default(none) private(i, sum, k) shared(a, j, n,
    ↪ cout)
24     for (i = j; i < n; i++) {
25       sum = a(i, j);
26       for (k = 0; k < j; k++)
27         sum -= a(i, k) * a(k, j);
28       a(i, j) = sum;
29     }
30     // look for pivot. This area is not worth parallelizing since we will have
       ↪  to make the
31     // conditional a critical section. We interested in the row-index of max(
       ↪ big), not it's value.
32     // the operatio is scalar-scalar multiplication
33     big = 0.0;
34     for (i = j; i < n; i++) {
35       dum = scaling[i] * abs(a(i, j));
36       if (dum >= big) {
37         big = dum;
38         imax = i;
39       }
40     }
41     // swap pivot rows. No parallelization
42     if (j != imax) {
43       a.row(imax).swap(a.row(j));
44       scaling[imax] = scaling[j];
45     }
46     indx[j] = imax; // needed for backsubtitution if solveing Ax = b
47     if (a(j, j) == 0.0){
48         printf("Singular␣matrix␣in␣routine␣ludcmp");
49         exit(1);
50       }
51     if (j != n - 1) {
52       dum = (1.0 / a(j, j));
53       // build lower triangular part using the values we calculated above
54 #pragma omp parallel for default(none) private(i, k) shared(j, n, a, dum)
55       for (i = j + 1; i < n; i++)
56       a(i, j) *= dum;
57     }
58   }
```

The first loop simply finds the max absolute value of each row and stores it for future use. The first loop uses a dynamic scheduling because the number of iterations in the inner loops depend on the outer index; therefore threads with larger assigned indices will do less iterations. The pivoting part was left in serial because the conditional statement needs to be run by a single thread at a time to ensure the last thread to update imax is the thread with the max value. The swapping of the rows is also easier to handle in serial.

One problem I ran into was the if-statement checking for the index of the pivot. Originally, these statement was inside the $i = j$ loop inside an omp critical region. This caused a bug in the code where the output would sometimes be wrong. After extracting the if-statement to its own unparalleled loop, the code worked again consistently. The problem could have been that imax was not always the index of the global max value, but instead the index of the max value of the last thread that updated it.

1.2 Performance and Results

The code was tested against Eigen's C++ Library LU decomposition and the serial version of the above code for correctness with random matrices. The times were also measured using random matrices of size $64, 256, 512$, and 1024. LU decomposition was performed 3 times on each matrix and the minimum time was recorded. The results can be seen in figures 2 and 3. Overall, there was a noticeable speed-up in the parallel version. Since Tuckoo's CPU have 16 cores each, it makes sense that 16 is the optimal thread count. The efficiency follows an expected downward path. The smallest n value has the steepest curve while the largest n has the highest efficiency. Since Tuckoo's CPU do not have 32 physical threads, there is dramatic decrease of both speed-up and efficiency at 32 threads. Despite this, data before 32 meets our expectations of increase efficiency with increase problem size. Similarly, speed-up is more drastic with bigger problem but flattens out quickly with small problems.

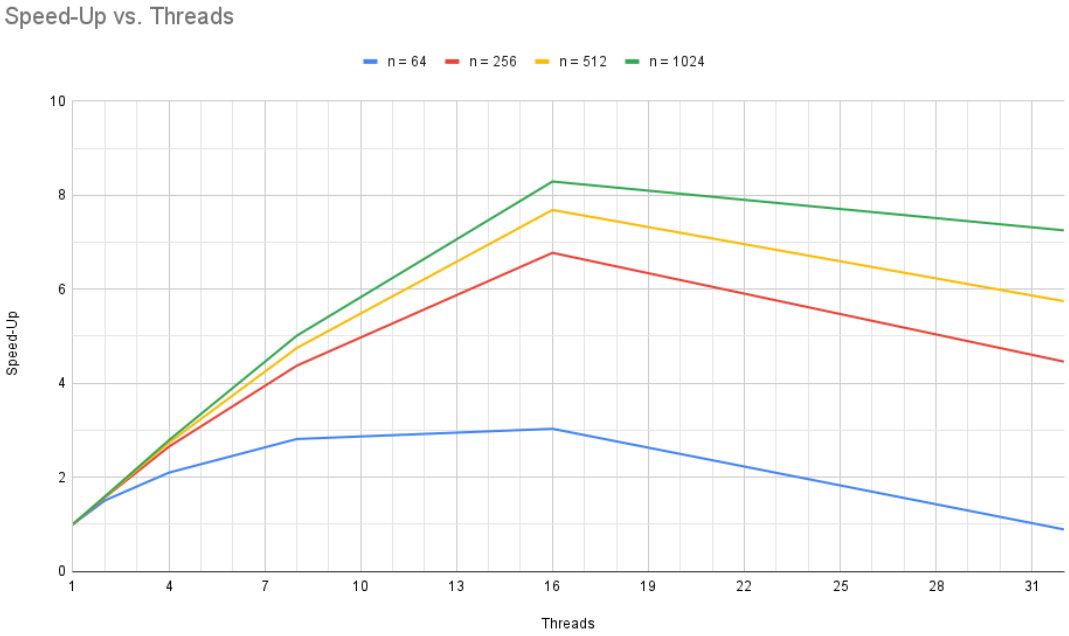

Figure 2: Speed-Up per number of processors per problem size. n is the dimensions of the square $n \times n$ matrix

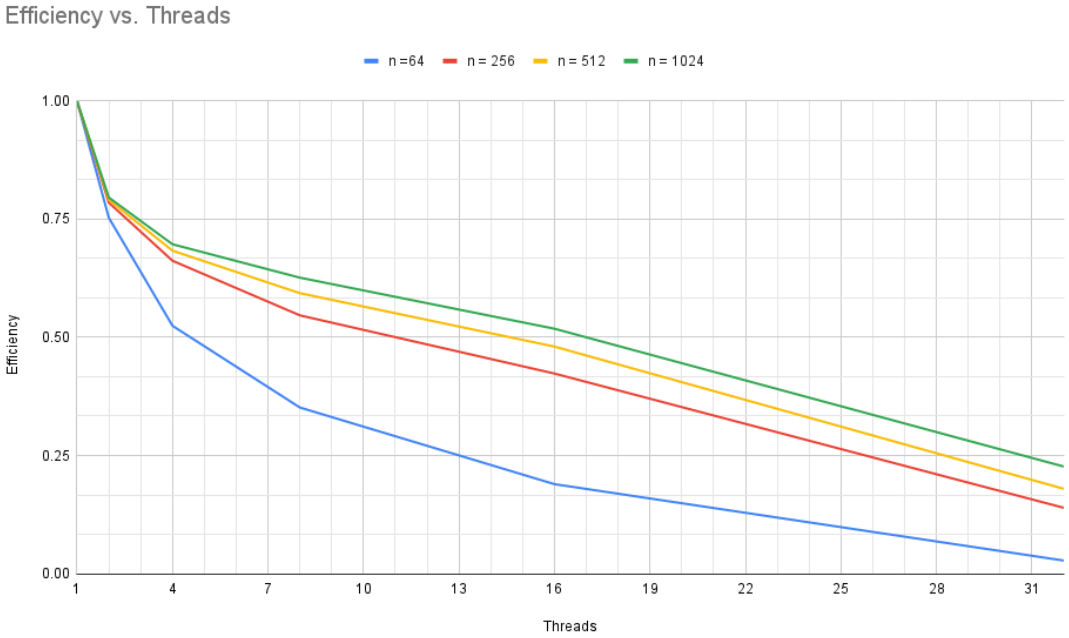

Figure 3: Efficiency per number of processors per problem size. n is the dimensions of the square $n \times n$ matrix

2 Gauss-Elimination

The second method we tested was Gauss-Elimination. This methods uses a series of row-operations to generate L and U. The algorithm first finds a pivot for column i, scales it by the pivot, then subtracts the $i+1$ row. For Gaussian's MPI implementation, we ran into issues with sending and receiving the data within Eigen data, which is the library we used to represent our matrices. Firstly, Eigen matrices are more complex data structures than MPI natively supports. For any communication to happen we have to ensure that we are extracting native C arrays and sending those instead. Eigen does have a Matrix.row(number).data() function that can be used for this purpose. The next issue was that Eigen by default stores data internally in column major order, which means data is stored such that columns are traversed before rows. I experimented with storing the data in Row Major order, but it only caused issues on the data ingestion and didn't seem to improve accuracy or performance, but there is some impact there that can be explored. Finally, we ran into an issue with the allocation of memory between processes. Our data loader (matrix_reader.cpp) dynamically reads in a CSV file and allocates the memory space using a dynamically sized matrix object on the fly. This approach, by default, is not going to work for broadcasting data between processes for MPI. We solved this by requiring size as an input for the Gaussian MPI program, and pre-allocating the space in each process before ingesting the data into the allocated space.

Listing 2: Gauss-Elimination MPI

```
for (int k = 0; k < n; ++k) {
// broadcast current pivot row if its my turn
MPI_Bcast(mat.row(k).data(), n, MPI_FLOAT, k % procs, MPI_COMM_WORLD);

    for (int i = k + 1; i < n; ++i) {
        if (i % procs == rank) {
            double factor = mat(i, k) / mat(k, k);
                for (int j = k; j < n; ++j) {
                    mat(i, j) -= factor * mat(k, j);
                }
                mat(i, k) = factor;
        }
    }
}

int main(){
    // give data to processes
    MPI_Bcast(mat.data(), n*n, MPI_FLOAT, 0, MPI_COMM_WORLD);
    // run it!
    lu_parallel(rank, procs, mat, n);
    // gather, send and recieve are the same buffer for rank 0 so skip it with
        ↪ MPI_IN_PLACE
    MPI_Gather(mat.data(), (n*n) / procs, MPI_FLOAT, mat.data(), (n*n) / procs
        ↪ , MPI_FLOAT, 0, MPI_COMM_WORLD);
    // print result
}
```

2.1 Performance and Results

The parallel Gaussian implementation was run at a $1, 2, 4, 8, 16$ and 32 process counts using MPI, and with randomly generated square matrices sized $64, 256, 512$, and 1024. The matrices were generated such that they were gauranteed to be invertible. We also wrote a seperate serial only Gaussian implementation to validate our outputs and compare correctness, as it was difficult to achieve correct results particularly at larger matrix sizes. This is likely due to a combination of incorrectly implementing the algorithm in the parallel code and testing on matrices that are not stable with Gaussian LU decomposition.

As for the performance, using figures 4 and 5 we can see that up to 4 processes we achieve speedups nearly equal to the number of processes. Efficiency does appear to go over 1 slightly for the 2 process run due to the poor performance when running the MPI version of the algorithm with a single process. Generally, the observed performance up to 4 processes is indicative of potential issues in the implementation, especially when comparing the correctness. As we increase the process count to 16, we see a decrease in performance for small size matrices, as the overhead in setting up all the processes outweighs any benefit. For larger matrices, performance levels off after 8 processes due to the aforementioned hardware limitations of the Tuckoo environment.

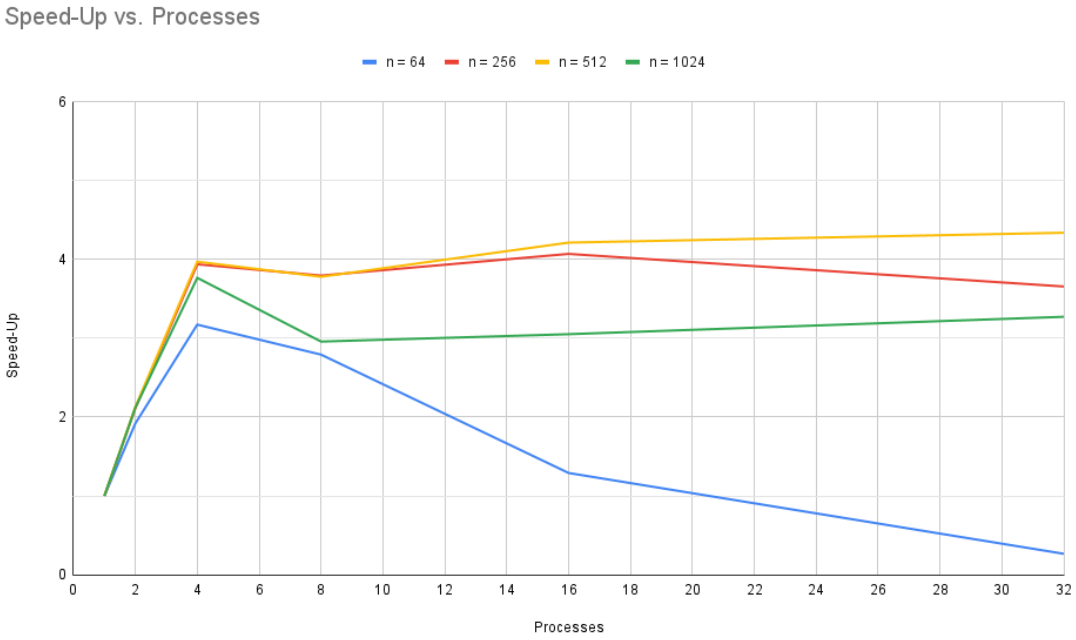

Figure 4: Speed-Up per number of processors per problem size. n is the dimensions of the square $n \times n$ matrix

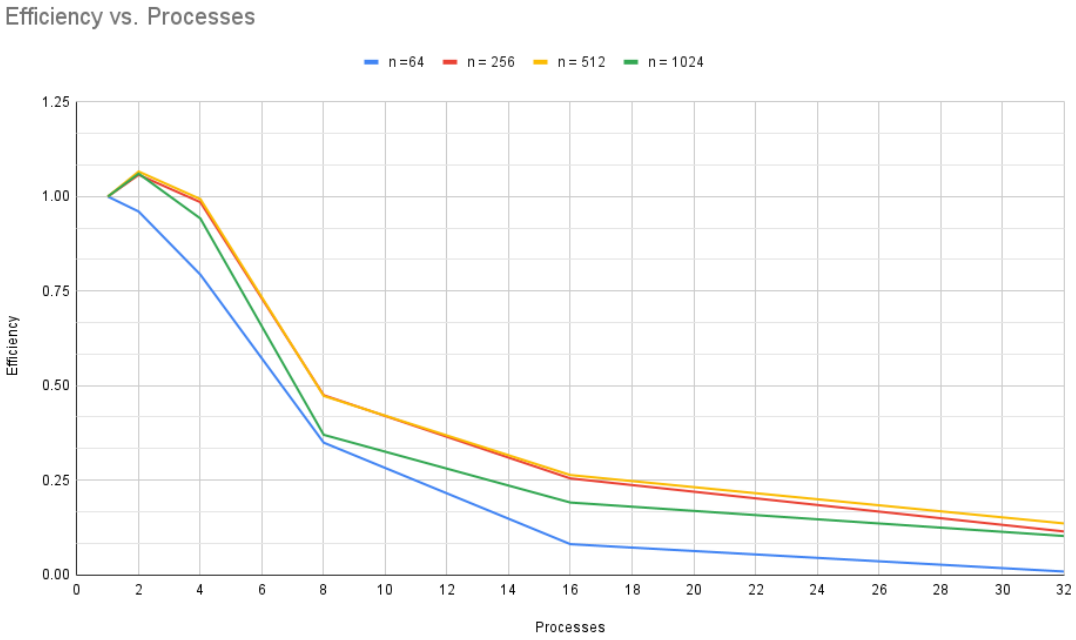

Figure 5: Efficiency per number of processors per problem size. n is the dimensions of the square $n \times n$ matrix