Raul L Bernal-Gonzalez
Alonzo Castanon

Final Report

COMP-605
May 2, 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. 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 6

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \\ a_{41} & a_{42} & a_{43} & a_{44} \end{bmatrix} \rightarrow \begin{bmatrix} \beta_{11} & \beta_{12} & \beta_{13} & \beta_{14} \\ \alpha_{21} & \beta_{22} & \beta_{23} & \beta_{24} \\ \alpha_{31} & \alpha_{32} & \beta_{33} & \beta_{34} \\ \alpha_{41} & \alpha_{42} & \alpha_{43} & \beta_{44} \end{bmatrix} \tag{6}$$

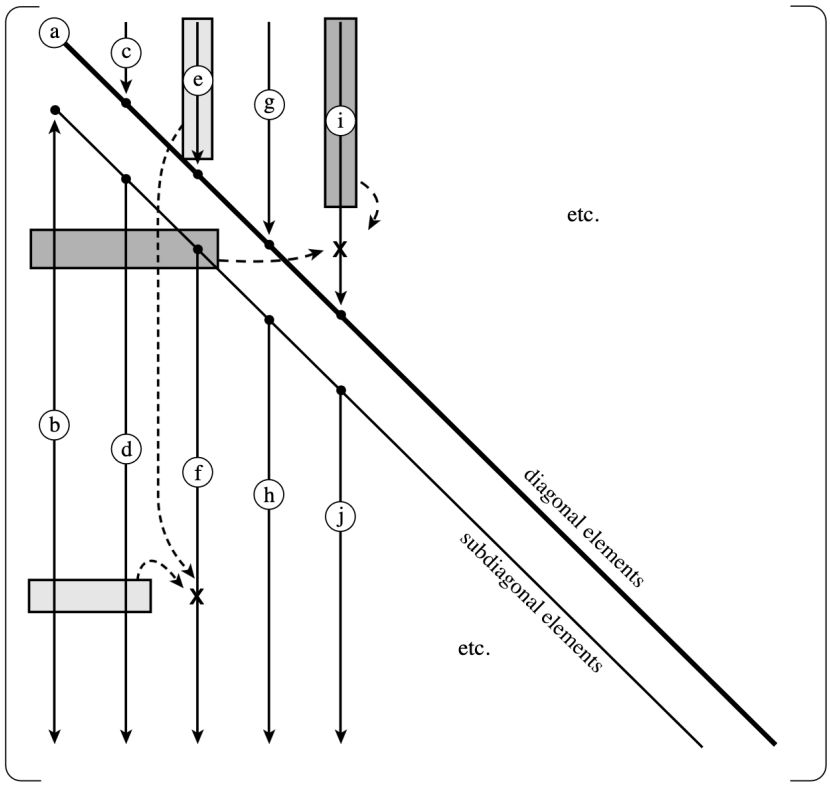

Figure 2.3.1. Crout's algorithm for *LU* decomposition of a matrix. Elements of the original matrix are modified in the order indicated by lower case letters: a, b, c, etc. Shaded boxes show the previously modified elements that are used in modifying two typical elements, each indicated by an "x".

1.1 Parallelization

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
// 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;
}
// main crout's loop. iterate by column
```

```
for (j = 0; j < n; j++) {
    // build upper trianfular part up to j-1 row since eache thread will have
       ↪ its own set of i's,
    // no overlap the a(k,j) term would have already been build in previous
       ↪ iterations of j
#pragma omp parallel for default(none) private(i, k, sum) shared(a, j)
   ↪ schedule(dynamic)
    for (i = 0; i < j; i++) {
      sum = a(i, j);
      for (k = 0; k < i; k++)
        sum -= a(i, k) * a(k, j); // we are not reducing to sum
      a(i, j) = sum;
    }
    // continue to build upper triangular part from i = j up to n rows
 #pragma omp parallel for default(none) private(i, sum, k) shared(a, j, n, big
    ↪ , imax, cout) schedule(dynamic)
    for (i = j; i < n; i++) {
      sum = a(i, j);
      for (k = 0; k < j; k++)
        sum -= a(i, k) * a(k, j);
      a(i, j) = sum;
    }
    // look for pivot. This area is not worth parallelizing since we will have
       ↪  to make the
    // conditional a critical section. We interested in the row-index of max(
       ↪ big), not it's value.
    // the operatio is scalar-scalar multiplication
    big = 0.0;
    for (i = j; i < n; i++) {
      dum = scaling[i] * abs(a(i, j));
      if (dum >= big) {
        big = dum;
        imax = i;
      }
    }
    // swap pivot rows. No parallelization
    if (j != imax) {
      a.row(imax).swap(a.row(j));
      scaling[imax] = scaling[j];
    }
    indx[j] = imax; // needed for backsubtitution if solveing Ax = b
    if (a(j, j) == 0.0){
        printf("Singular matrix in routine ludcmp");
        exit(1);
      }
    if (j != n - 1) {
      dum = (1.0 / a(j, j));
      // this last part build lower triangular part using the values we
         ↪ calculated above
#pragma omp parallel for default(none) private(i, k) shared(j, n, a, dum)
      for (i = j + 1; i < n; i++)
      a(i, j) *= dum;
    }
  }
```

The first loop simply finds the max absolute value of each row and stores it for future use. The first 2 loops use 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.

1.2 Performance and Results

The code was tested against Eigen's C++ Library LU decomposition algorithm for correctness with random matrices. The times were also measured using random matrices of size $64, 256, 512$, and 1024.

Problem 2

I use TEXstudio, but Overleaf and TEXmaker are also popular.