Bidirectional LU Factorization

PREPARED FOR

DR. SURYA PRAKASH

PREPARED BY

PUSHPENDRA KUMAR : 160001046

KANISHKAR J : 160001028

Introduction

In numerical analysis and linear algebra, lower–upper (LU) decomposition or factorization factors a matrix as the product of a lower triangular matrix and an upper triangular matrix. The product sometimes includes a permutation matrix as well. The LU decomposition can be viewed as the matrix form of Gaussian elimination. Computers usually solve square systems of linear equations using the LU decomposition, and it is also a key step when inverting a matrix or computing the determinant of a matrix.

As, we can see how important LU decomposition is in scientific computations. And the computing power of a single processor is not always enough for processing very big data. Hence, In this project we shall try to Parallelize the LU decomposition method (Bidirectional LU factorization), and we shall also compare its performance metrics with the normal LU decomposition method.

In this project we will implement a parallel *bidirectional algorithm*, based on *LU factorization*, for the solution of general sparse system of linear equations having *non symmetric* coefficient matrix. As with the sparse symmetric systems, the numerical factorization phase of our algorithm is carried out in such a manner that the entire back substitution component of the substitution phase is replaced by a single step division. However, due to absence of symmetry, important differences arise in the ordering technique, the symbolic factorization phase, and message passing during numerical factorization phase. The bidirectional substitution phase for solving general sparse systems is the same as that for sparse symmetric systems.

Project Roadmap

Implementation

First, We will be implementing the sequential and parallel algorithms for LU Decomposition problem in c++ using OpenMP/CUDA .

Verification

We shall then verify the outputs of both the procedures for different matrices and compare them.

Analysis

- Compare Speed-ups for Matrices of different sizes and compare plot a graph of the data obtained.
- Compare Speed-ups while increasing the number of processors used and plot a graph with it.

Testing

We will test our algorithm on randomly generated matrices of various densities.

Solve System of Linear Equations

We will use LU decomposition to solve a system of linear equations. Here we will be using both the sequential and the parallel algorithm and then note down the speed-up.

Methodology

LU Factorization

It is always possible to factor a square matrix into a lower triangular matrix and an upper triangular matrix. That is, [A] = [L][U]. Doolittle's method provides an alternative way to factor A into an LU decomposition without going through the hassle of Gaussian Elimination. For a general n×n matrix A, we assume that an LU decomposition exists, and write the form of L and U explicitly. We then systematically solve for the entries in L and U from the equations that result from the multiplications necessary for A=LU.

After implementing the sequential algorithm, We will parallelize the above mentioned algorithms to achieve better performance.

Bidirectional Substitution

1. Forward elimination: reduction to row echelon form. Using it one can tell whether there are no solutions, or unique solution, or infinitely many solutions.
2. Back substitution: further reduction to reduced row echelon form.

We will be implementing methods for forward and backward substitution. Then we will be applying forward substitution and then backward substitution to solve the system of linear equations.

Code

Sequential Algorithms

```
// Decomposing matrix into Upper and Lower triangular matrix
for (int i = 0; i < n; i++) {

   // Upper Triangular
   for (int k = i; k < n; k++) {
       // Summation of L(i, j) * U(j, k)
       int sum = 0;
       for (int j = 0; j < i; j++)
           sum += (lower[i][j] * upper[j][k]);
       // Evaluating U(i, k)
       upper[i][k] = mat[i][k] - sum;
   }

   // Lower Triangular
   for (int k = i; k < n; k++) {
       if (i == k)
           lower[i][i] = 1; // Diagonal as 1
       else {
           // Summation of L(k, j) * U(j, i)
           int sum = 0;
           for (int j = 0; j < i; j++)
               sum += (lower[k][j] * upper[j][i]);
           // Evaluating L(k, i)
           lower[k][i] = (mat[k][i] - sum) / upper[i][i];
       }
   }
}
```

Parallel Algorithm

```
for(int k=0;k<n;k++)
   {
       #pragma omp parallel for
       for(int i=k+1;i<n;i++)
           matrix[i][k]=matrix[i][k]/matrix[k][k];

       #pragma omp parallel for
       for(int i=k+1;i<n;i++)
       {
           for(int j=k+1;j<n;j++)
           {
               matrix[i][j]=matrix[i][j]-matrix[i][k]*matrix[k][j];
           }
       }
   }

   #pragma omp parallel for
   for(int i=0;i<n;i++)
   {
       #pragma omp parallel for
       for(int j=0;j<=i;j++)
       {
           lower[i][j]=matrix[i][j];
           upper[j][i]=matrix[j][i];
       }
       lower[i][i]=1;
   }
```

Forward Substitution

```
y.resize(n);
for (int i=0 ; i<n ; i++)
{
    y[i]=b[i];
    for (int j=0 ; j<i ; j++) {
        y[i]-=lower[i][j]*y[j];
    }
}
```

Backward Substitution

```
x.resize(n);

   for(int i=n-1;i>=0;i--)
       {
       x[i]=y[i];
           for(int j=i+1;j<n;j++)
               x[i]-=upper[i][j]*x[j];
           x[i]/=upper[i][i];
       }
```

Complexity

Sequential

$T(n) = O(n^3)$

Parallel

$T(n,p) = O(n^2)$

$W(n,p) = O(n^3)$

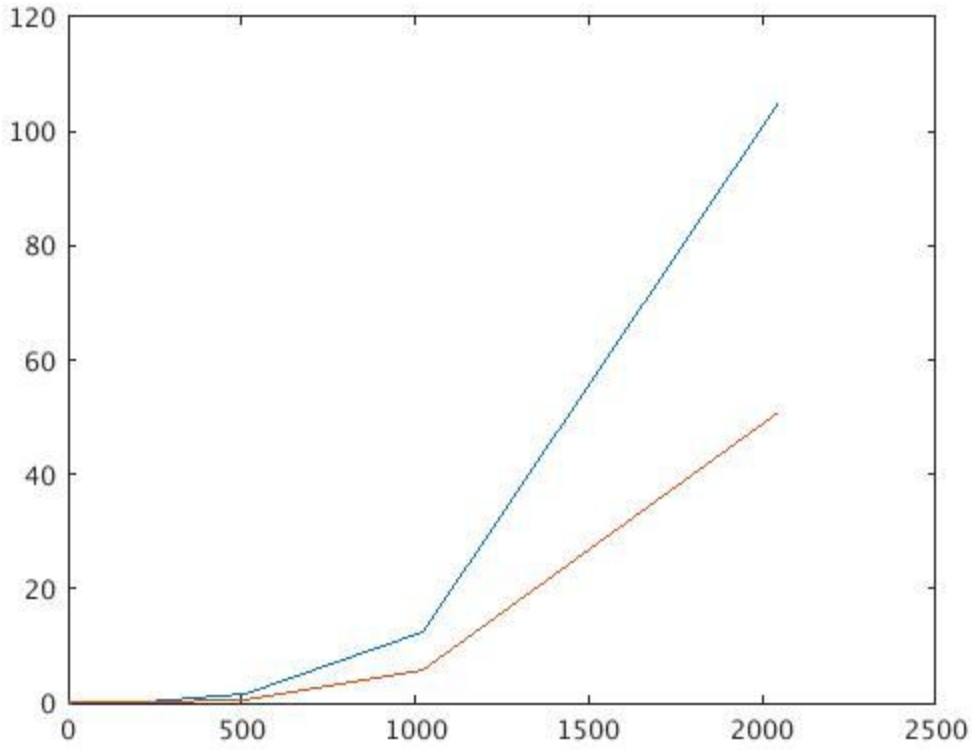
No. of processors (p) = n^3

Results

Size of Matrix	Sequential Time	Parallel Time	Speed-Up
4	0.000002	0.00228639	0.0008747414046
8	0.000008	0.00571875	0.001398907104
16	0.000043	0.00404808	0.01062231972
32	0.000441	0.00229588	0.1920832099
64	0.004174	0.00337355	1.23727231
128	0.025038	0.012258	2.042584435
216	0.103554	0.049646	2.085847802
512	1.51956	0.651968	2.330727888
1024	12.5911	5.89344	2.1364602
2048	105.183	50.9156	2.065830512

Graph

Matrix Size vs Execution time



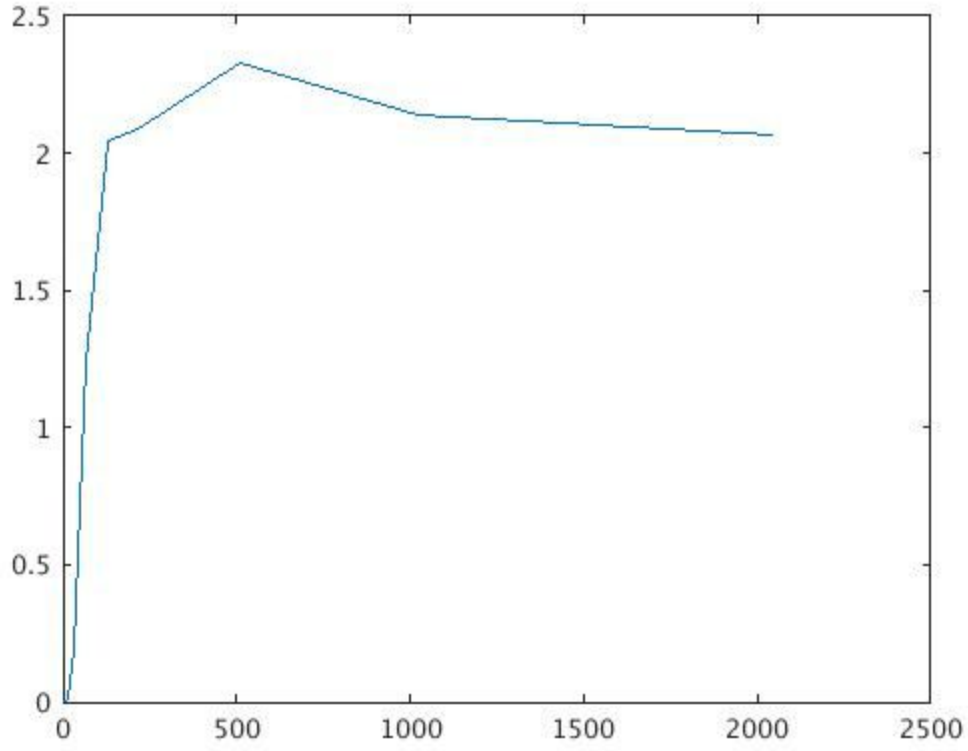
_____ **Sequential**

_____ **Parallel**

x-axis : Matrix size

y-axis : Execution Time

Matrix Size vs Speed-up



x-axis : Matrix Size

y-axis : Speed-up

Conclusion

From the Readings we can conclude that the speedup increases above 1 after the matrix size exceeds 32. Till then the sequential performs better than the parallel.