CS5334: Project Report
Tasked Based LU Decomposition
Md Abdul Kader

Introduction:

The task for this project was to implement an algorithm to decompose a matrix A into a lower triangular matrix and an upper triangular matrix. But to utilize the multicore parallel system, the project requirements have been extended to Tasked-Based LU Decomposition.

In this short report I am going to explain background, my implementation, design decision and performance analysis in the following sections.

Background:

LU Decomposition/Factorization:

A procedure for decomposing an $N \times N$ matrix A into a product of a lower triangular matrix L and an upper triangular matrix U.
A=LU.

Formally, you are given matrix A, find matrix L and U.

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{bmatrix} = \begin{bmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{bmatrix} \begin{bmatrix} u_{11} & u_{12} & u_{13} \\ 0 & u_{22} & u_{23} \\ 0 & 0 & u_{33} \end{bmatrix}$$

Here is an example

$$\underbrace{\begin{bmatrix} 1 & 2 & 4 \\ 3 & 8 & 14 \\ 2 & 6 & 13 \end{bmatrix}}_{A} = \underbrace{\begin{bmatrix} 1 & 0 & 0 \\ 3 & 1 & 0 \\ 2 & 1 & 1 \end{bmatrix}}_{L} \underbrace{\begin{bmatrix} 1 & 2 & 4 \\ 0 & 2 & 2 \\ 0 & 0 & 3 \end{bmatrix}}_{U}$$

Algorithm Design:

I used the idea of Divide and Conquer to solve the problem. I used it to take the advantage of independent task that the Divide and Conquer produces.

Idea:

1. Partition the matrix A into four equal (almost equal) matrices.

$$A = \begin{pmatrix} A00 & A01 \\ A10 & A11 \end{pmatrix}$$

2. Solve the small sub matrices.
3. Combine the solutions.

Procedure

$$A_{00} = L_{00}U_{00} \quad \text{STEP1}$$

$$A_{01} = L_{00}U_{01} \quad \text{STEP2}$$

$$A_{10} = L_{10}U_{00} \quad \text{STEP3}$$

$$A_{11} = L_{10}U_{01} + L_{11}U_{11} \quad \text{STEP4}$$

This is a single iteration of the decomposition recursion.

Corresponding tasked dependency graph is in figure 1.

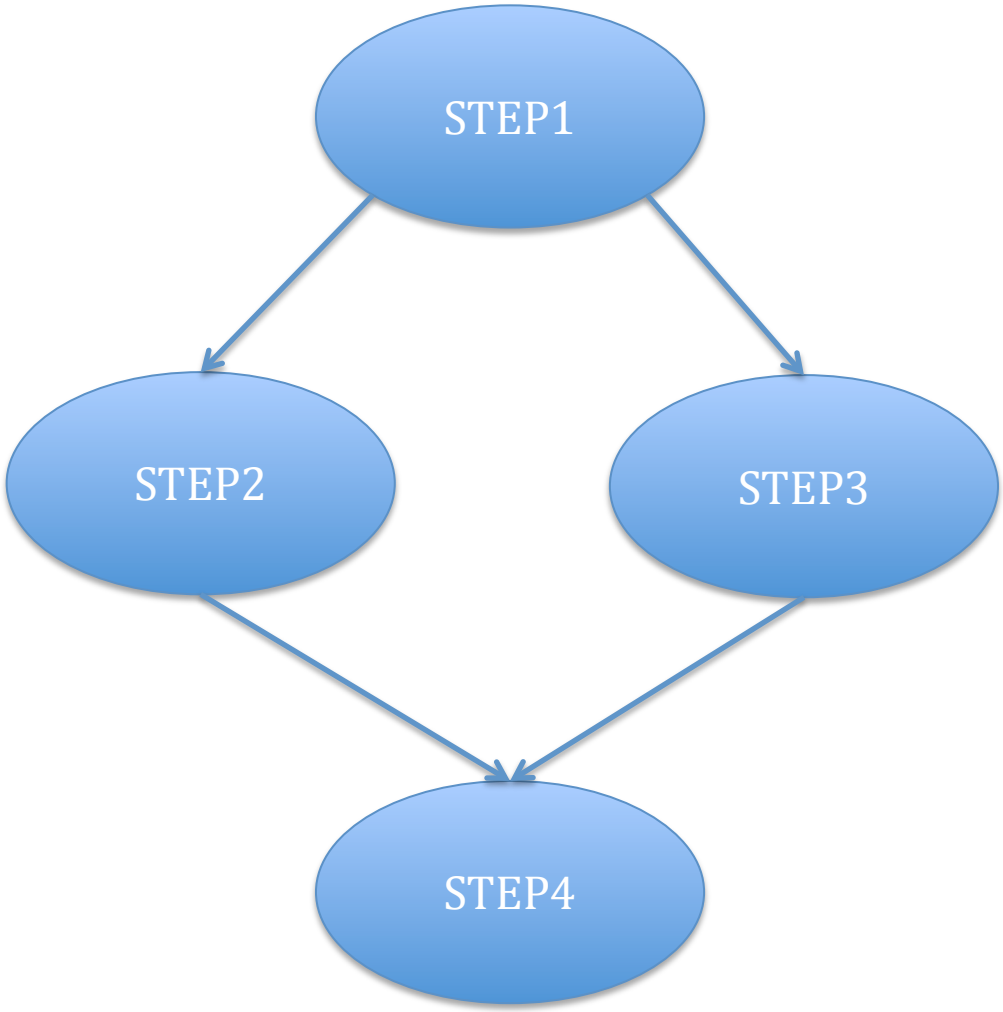

Figure 1 Task dependency graph

It has concurrency two only. But fortunately it gets more concurrency in STEP2 and STEP3 if I apply same Divide and Conquer idea in STEP2 and STEP3.

STEP2:
We need to find matrix C given matrices B and L such that B=LC, where L is a lower triangular matrix.

STEP3:
We need to find matrix C given matrices B and U such that B=CU, where U is an upper triangular matrix. This step is similar to STEP2.

Pseudo code: [1]

```
procedure LU(A, L, U)
    LU( A00, L00, U00 );
    SolveLower( A01, L00, U01 );
    SolveUpper( A10, L10, U00 );
    Multiply( −1, L10, U01, A11 );
    LU( A11, L11, U11 );
```

Pseudo-code 1: LU decomposition

```
solve_lower( B00, L00, C00 )
solve_lower( B01, L00, C01 )
B10 := B10 − L10C00;
B11 := B11 − L10C01;
solve_lower( B10, L11, C10 )
solve_lower( B11, L11, C11 )
```

Pseudo-code 2: Solve Lower (STEP2)

Implementation:

Approach:

- Implemented Non-threaded version (Sequential version).
- Threaded version with OpenMP task directive
  - It was crashing because of thread unsafe C++ STL vector.
  - Then I replaced vector with my own implementation of dynamic array.
- Threaded version using Cilk

Let's see two code snippets.

```
#pragma omp task shared(U01)
{
    U01=solveLower(A01,L00);
}
#pragma omp task shared(L10)
{
    L10=solveUpper(A10,U00);
}
#pragma omp taskwait
```

Code Snippet 1: OpenMP Implementation

```
U01=cilk_spawn solveLower(A01,L00);
L10=cilk_spawn solveUpper(A10,U00);
cilk_sync;
```

Code Snippet 2: Cilk Implementation

Cilk is easy and concise. The above code snippet was taken from the main LU decomposition recursion.

Sources:

I have created a repository in my github account for this project. All codes are available publicly there.
I have a well-documented readme file there. Please see the link https://github.com/sreezin/LU-Decomposition.

Tuning:

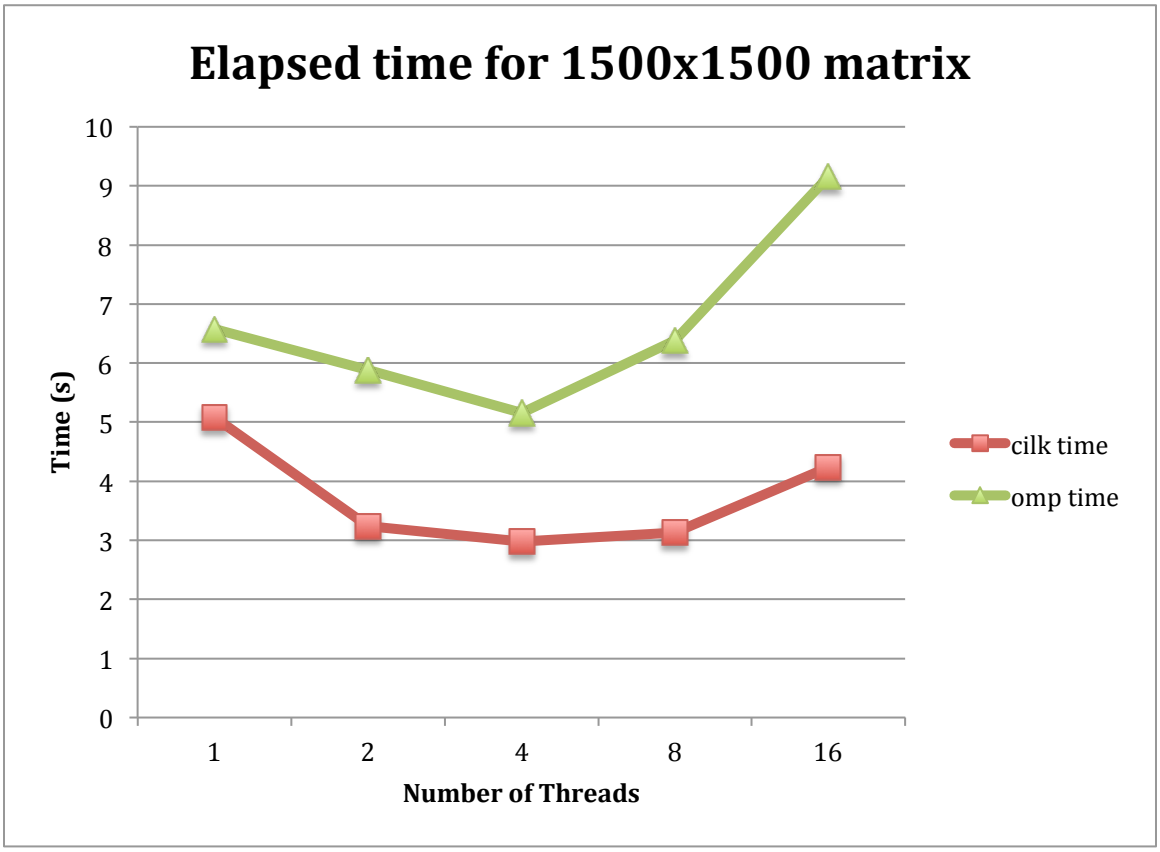

Figure 2: Tune my Algorithms.

Number of threads:

After observing the figure 2, I decided to use 4 threads for OpenMP implementation and 4 workers for Cilk implementation.

Magic Number:

I have a magic number for this project and that is 4. Number 4 is working best for several cases.

1. No spawning if sub problem size is less than or equal 4.
2. Switching to sequential version of SolveLower or SolveUpper if sub problem size is less than 4.

Performances:

Matrix Dimension (NxN)	Sequential Time (s)	OMP Time (s)	Cilk Time (s)
500x500	0.235685	0.217672	0.176355
1000x1000	**1.416806**	**1.537784**	1.126999
1500x1500	6.030336	5.027035	3.159244
2000x2000	14.08154	11.523919	6.24722
2500x2500	31.599557	26.912589	11.554249
3000x3000	61.703155	44.40926	19.438269
4000x4000	Mem Overflow	Mem Overflow	Mem Overflow

Table 1: Performances of three implementations

The visual representation of the above table is given blow that tells Cilk is super fast for my implementation of the LU decomposition algorithm.

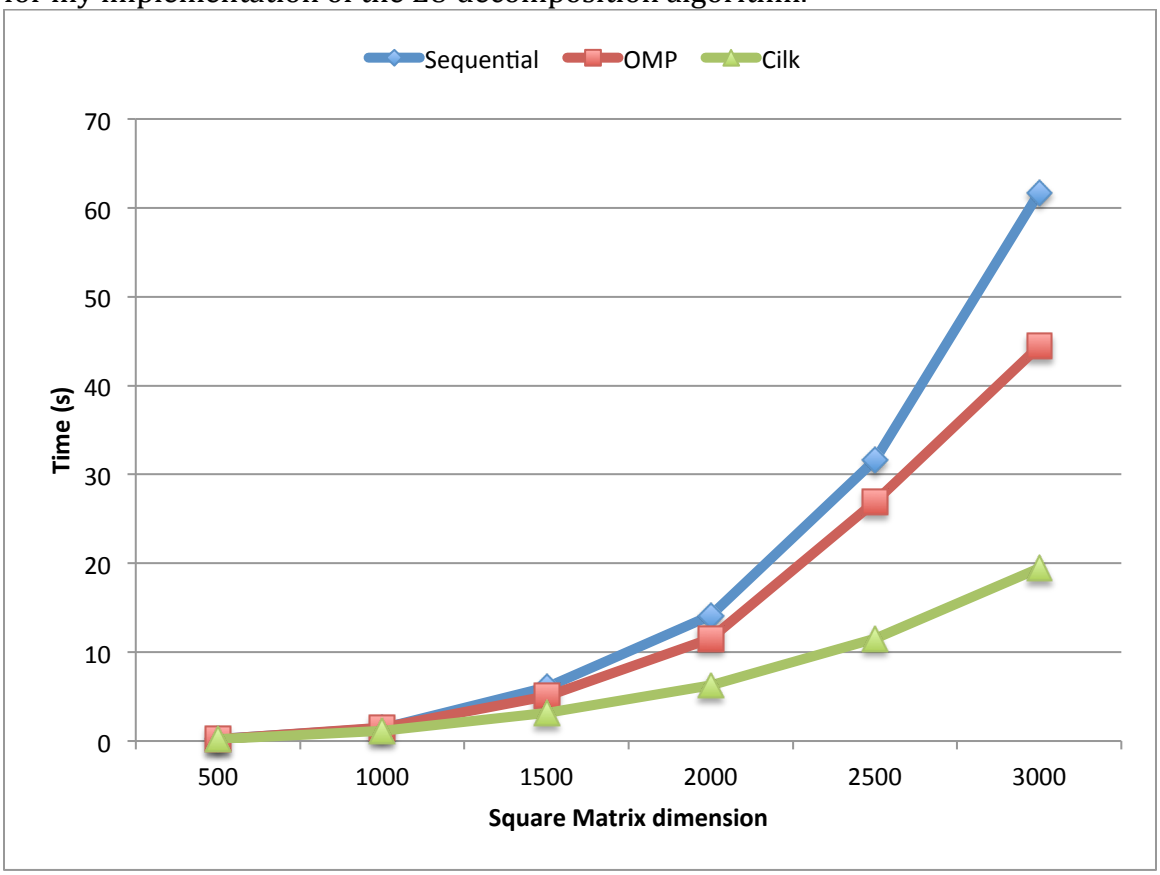

Figure 3: Performance comparison for sequential, OpenMP and Cilk

References:

[1] http://www.mis.mpg.de/preprints/2014/preprint2014_5.pdf