COL380 Assignment 1 - Parallel LU Decomposition

Pratyush Pandey (2017EE10938), Harkirat Dhanoa (2017CS50408)

19^{th} January, 2020

Contents

1 Introduction

Lower–Upper (LU) decomposition or factorization factors a given matrix as the product of a lower triangular matrix and an upper triangular matrix. Every Square Matrix A will admit an LU factorization given

- A is invertible and contains non zero leading principal minors (converse is also true).
- A is Singular, having rank k, then it's first k principal minors should be non zero (converse is not true).

However, LU decomposition isn't limited to just square matrices and also exists for a general (not necessarily invertible) matrix over any field, subject to ranks of certain sub-matrices.
A special case also exists for Symmetric (or Hermitian) positive definite matrices where a special algorithm called Cholesky's Algorithm is used to factorize the matrix. It can be shown that a unique Cholesky Decomposition will always exist for a positive definite matrix A, the computation of which is more efficient and numerically more stable than certain LU Decomposition.

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

Figure 1: LU Decomposition for a 3x3 Matrix

LU Decomposition is used by computers in the following algorithms:

- Solving system of Linear Equations
- Computing determinant of a matrix

2 Algorithm

```
Initilize A, L, U randomly using appropriate ranges of values (having the same
    dimensions)
for i = 1 to n-1
    1. find maximum absolute element in column i below the diagonal
    2. swap the row of maximum element with row i
```

The LU factorization algorithm that is most commonly used on parallel machines is simply a reorganization of classic Gaussian Elimination. The basic algorithm proceeds row by row, attempting to "eliminate" entries below the main diagonal. Multiples of row i are subtracted from rows below i in order to ensure that the part of column i below the main diagonal becomes zero. To enhance numerical stability pivoting, the swapping of rows to place a large value on the diagonal, is performed prior to each elimination step.

```
    3. scale column i below diagonal by 1/A[i][i]
    L[i][i]=1
    for j = i+1 to n
        L[j][i]=A[j][i]/A[i][i]
        4. Set row i of U
        for j = i to n
            U[i][j]=A[i][j]
```

Row interchanges (pivoting) is required to ensure existence of LU factorization and numerical stability of Gaussian elimination algorithm. After this process is completed, the solution of Ax=b can be obtained by forward and back substitution with L and U.

```
5. Perform a trailing matrix update, i.e. update the part of the matrix below and to the
    right of A[i][i]
        for j=i+1 to n
            for k = i+1 to n
                A(j,k) = A(j,k)-L(j,i)*U(i,k)
        6. This step can equivalently be expressed as a "rank-one update":
        A[i+1:n][i+1:n] = A[i+1:n][i+1:n] - L[i+1:n][i]*U[i][i+1:n]
Check : Compute the norm of the residue |PA - LU|, which should be very small
```

The Gaussian elimination step, which is the main bottleneck in the algorithm (taking up 97% of the execution time) has general form of triple-nested loop, as shown in step 5, in which entries of L and U overwrite those of A.
In our implementation, we have taken the ijk ordering of the nested for loops, but they can be taken in any order, for total of 3! = 6 different ways of arranging loops. Different loop orders have different memory access patterns, which may cause their performance to vary widely, depending on architectural features such as cache, paging, vector registers, etc. While thinking about improving the efficiency of the program via the use of multi-threads and parallelization, it becomes very important to order these loops in a way that promises efficient parallel implementation. These forms only differ in accessing matrix by rows or columns, respectively.

2.1 Runtime Analysis

We will ignore pivoting here for convenience. The Gaussian elimination step requires about $n^3/3$ paired additions and multiplications. About $n^2/2$ divisions also required, but we ignore this lower-order term. Hence the algorithm used is $O(n^3)$.
This is also evident from the data of table 1.6 as we double n, the running time becomes 8 times.

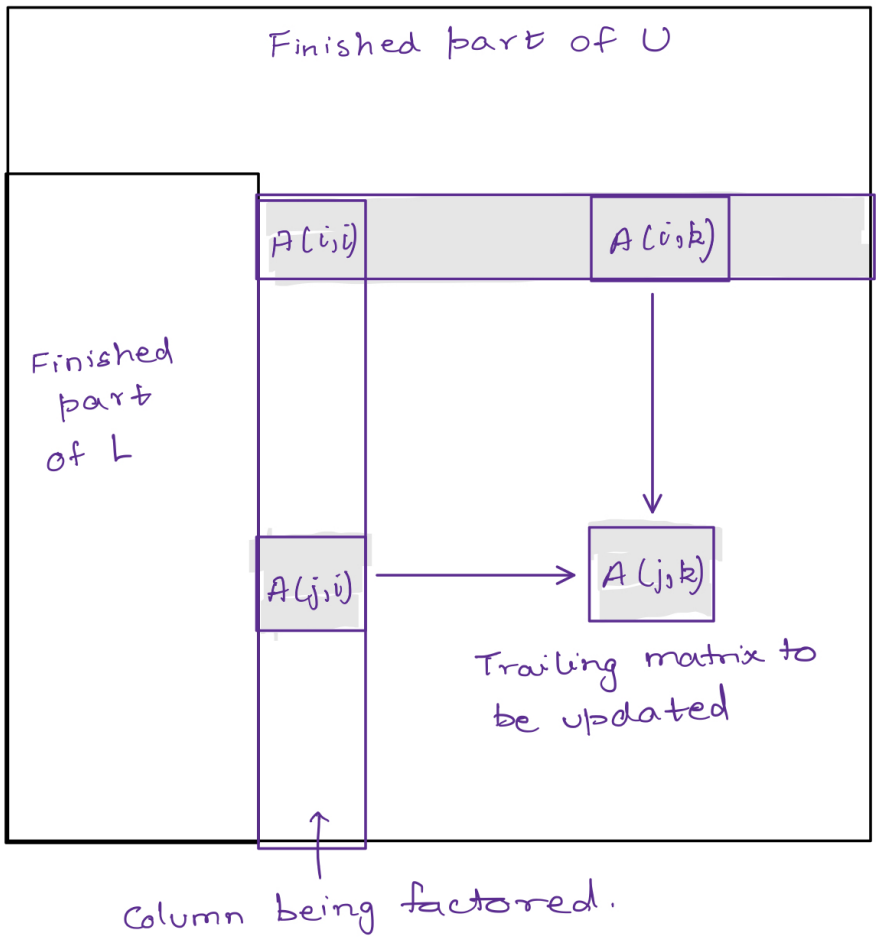


Figure 2: Dataflow in LU Factorization. The computation proceeds from left to right

3 Other Details

3.1 Data Structure

We used *double* ** to represent 2D matrices and *double* * for 1D matrix. This was helpful in passing references to the original matrix instead of creating copies every time which lead to unnecessary memory allocation. This also led to O(1) row swapping instead of swapping each and every single element from the rows.

3.2 Optimizations

We used pointers to pass matrices to avoid duplicacy of function arguments. Instead of swapping rows element by element, we instead swapped addresses of these rows (*double* *) in $O(1)$ time.

3.3 Memory

Declaring an array of suppose size $n = 4000$ will lead to $4000 * 4000 * 8$ *bytes* of memory being allocated, since datatype *double* takes 8 *bytes*. This is 128 MB per such matrix, and in the code we have 5 matrices of size $n * n$. So for $n = 4000$, 640 MB of RAM is being used. Which makes it extremely important to free up memory by deleting this allocated memory once the requirement for it ceases finishes.

3.4 Performance

In both the parallelization strategies, we observed nearly two times speed-up when using *2* threads and nearly four times using $4, 8$ and 16 threads. This was because the computer we used was quad-core. Also, the fraction of code that was parallelized was $1 - f = 0.97$. So according to *Amdahl's Law*, $S(n) = \frac{n}{1+(n-1)f}$, the speedup will be nearly n times (as $f \approx 0$).
Note : The runtimes aren't exactly calculable from amdahl's law due to two reasons:

1. The matrices have been initialized randomly and since arithmetic operations such as multiplication aren't exactly O(1) so the total time can't be exactly pre-determined
2. All the cores aren't fully available for running the code due to other applications running on the computer. Therefore the efficiency of each core is less which affects the runtime of the code which use less number of threads more than that of the code which uses more number of threads.

3.5 Correctness

Correctness of the algorithm was verified by calculating the norm of $\|PA - LU\|$ which was of the order 10^{-13} for matrix size = 8000 and smaller (10^{-18}) for smaller sizes of n

4 Analysis of OpenMP

OpenMP allows for parallelization of loops with no intra-loop data dependency. We observed that for chunks with less time complexity such as initialization of matrices (O(n^2)), or row swapping/duplicating (also O(n^2)) parallelization won't lead to any significant improvement in the overall running time because its fraction of time taken is very less as compared to the O(n^3) part of the code. Instead if we parallelize this part then we will get more overhead of creating, forking and joining the threads after every such trivial computation. This in fact lead to an increase in the total time of execution.
Therefore, we parallelized the triple-nested for loop towards the end of LU decomposition algorithm.
We kept i (row index) private i.e. unchanged at the end of parallel block, and the three matrices: A(input), U(upper triangular) and L (lower triangular) shared.

```
# pragma omp parallel for num_threads(num_thread) private(i) shared(A,L,U)
```

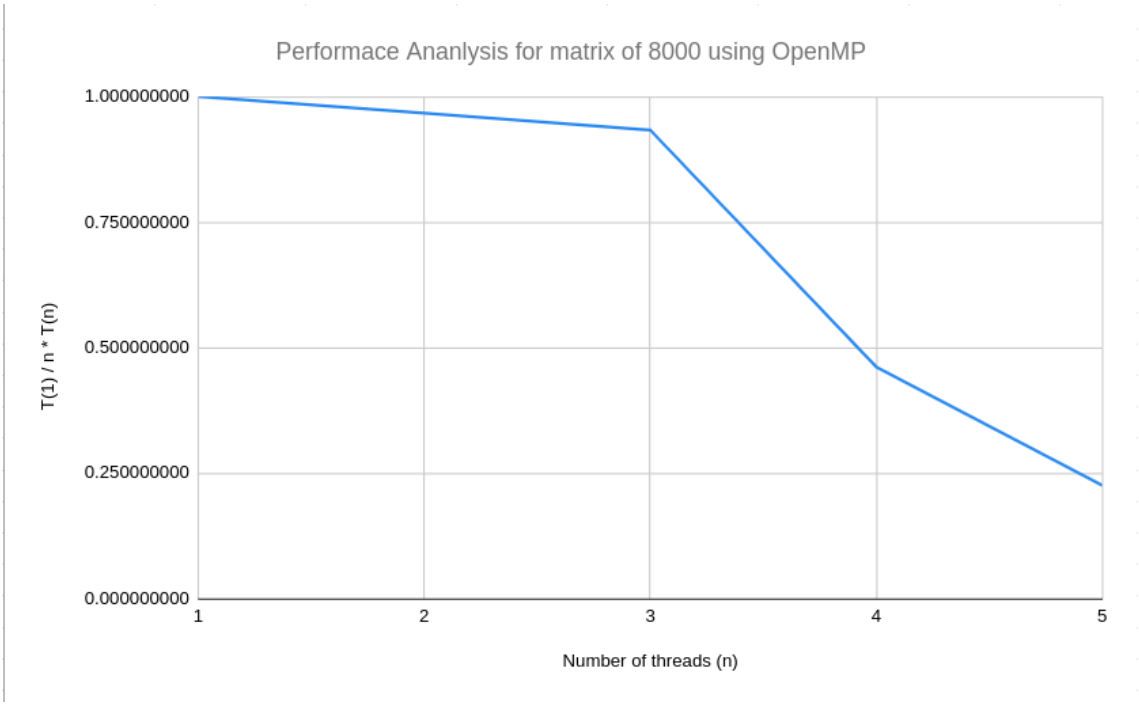
We used the *collapse*() function to unroll the nested loop structure into a single loop but the process itself consumed more time than what it improved overall since the outer loop was of length (*n-k-1*) where k increased from *0* to *n-1*, which is not very fruitful.
We tested using various parallelization schemes such as static scheduling and dynamic scheduling with chunk sizes varying from $1, 2, ..., 64$. We also tried interleaved scheduling (follows round-robin manner). The default scheduling (having *n/num_thread* chunks) gave best results so we stick with that.

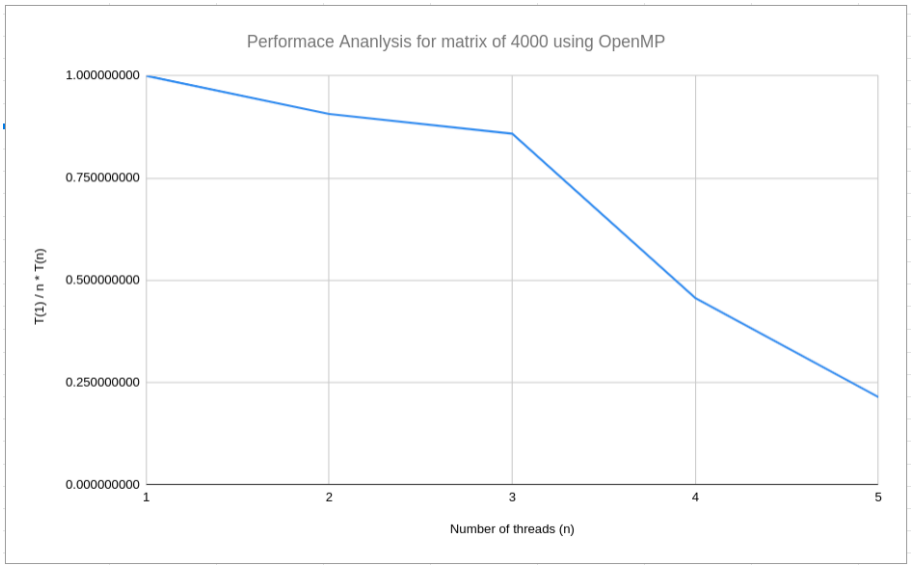
4.1 Run Times

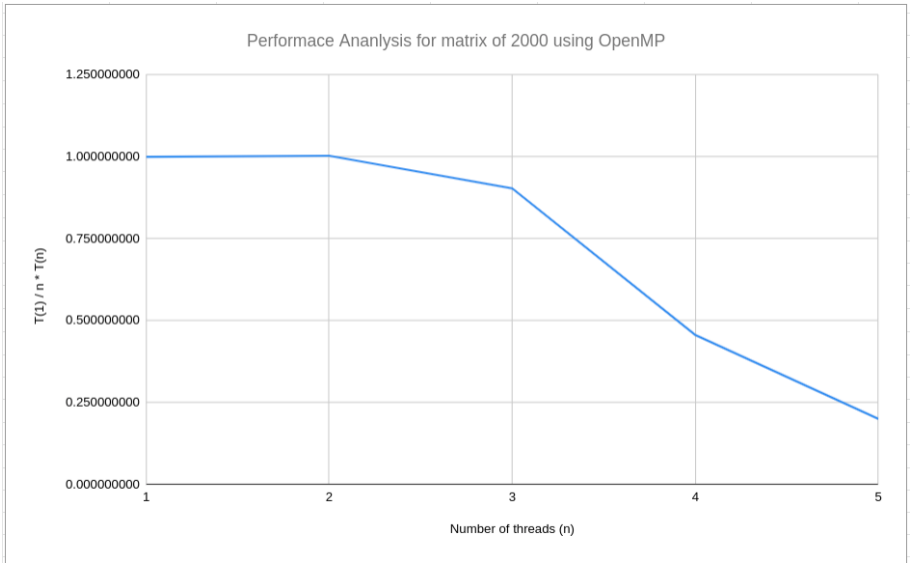
No. of Threads	N (size of Matrix)	Execuation Time (seconds)
1	500	0.090474
	1000	1.182
	2000	11.128675
	4000	84.0098
	8000	682.592595
2	500	0.085395
	1000	0.667363
	2000	5.546747
	4000	46.348368
	8000	353.00991
4	500	0.073444
	1000	0.41433
	2000	3.078534
	4000	24.472246
	8000	182.856062
8	500	0.054294
	1000	0.38286
	2000	3.048087
	4000	23.009764
	8000	185.08032
16	500	0.074628
	1000	0.461238
	2000	3.473308
	4000	24.495561
	8000	189.532123

Table 1: Run Times of LU Decomposition code using varied number of threads and size of A Matrix optimised using OpenMP

4.2 Performance graphs







5 Analysis of pthreads

Using pthreads we parallelized the outermost for-loop in the triple-nested part towards the end of LU decomposition algorithm. We split it into *num_threads* number of chunks and used one thread each to loop over one chunk by calling a function *thread_computation*.
We passed separate arguments into each thread by creating a *struct*. This was done to remove data dependency amongst the loops.

```
pthread_t threads[num_thread];
for k = 1 to n
    .
    .
    for i = 0 to num_thread
        args *in = (args *)malloc(sizeof(args));
        /* initializing arguments to be passed into the i-th thread */

        pthread_create(&threads[i], NULL, thread_computation, (void*)(in));

    for i=0 to num_thread
```

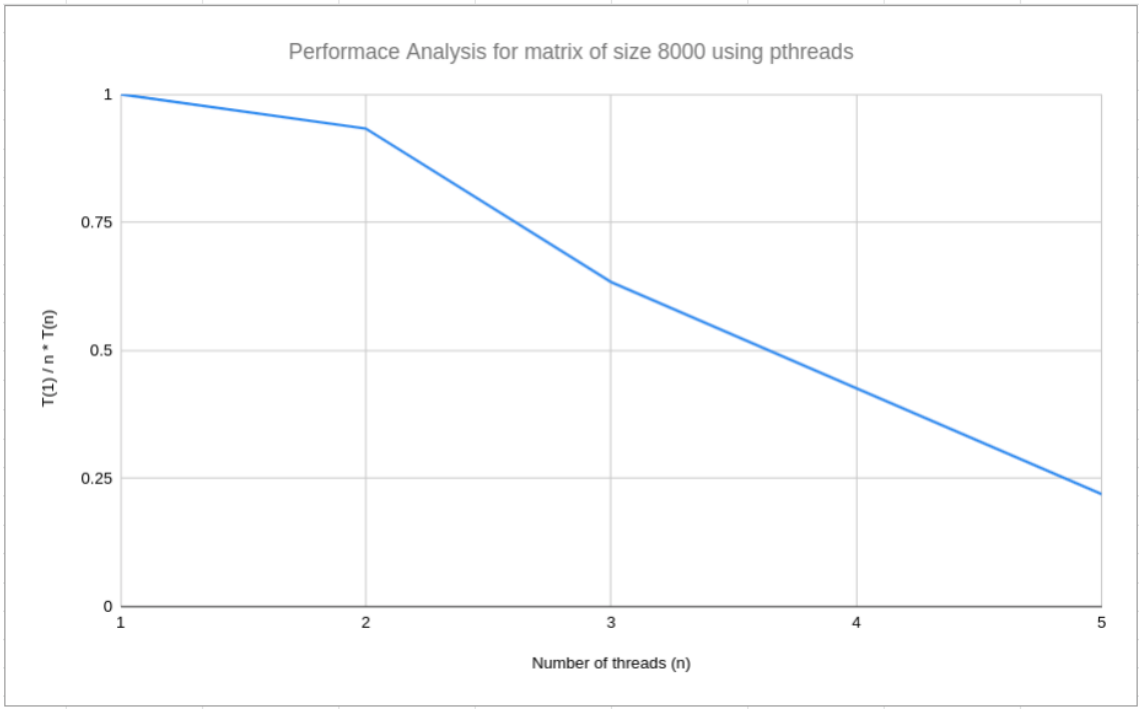
```
        pthread_join(threads[i], NULL);

thread_computation:
    /* gaussian eliminiation step with these constraints on the outer loop */

    for i = (k+1)+core*(n-(k+1))/num_thread to (k+1)+(core+1)*(n-(k+1))/num_thread
        .
        .
```

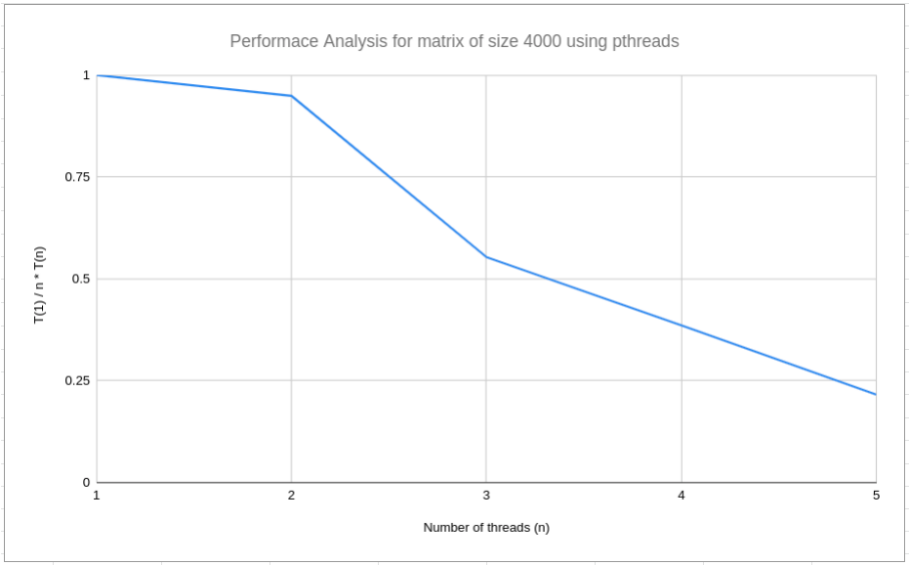
We maintained all threads in an array declared once before the beginning of the outermost loop. Then in its every iteration, we declare new arguments for each thread and create the threads. Once all threads finish execution they are joined together and the outermost loop moves to the next iteration.

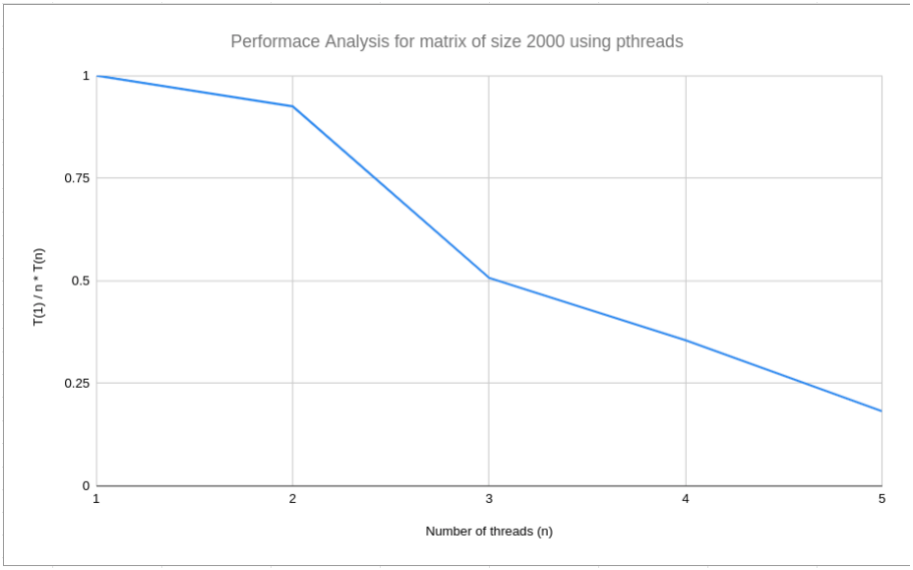
5.1 Run Times

No. of Threads	N (size of Matrix)	Execuation Time (seconds)
1	500	0.33799
	1000	2.196433
	2000	12.488279
	4000	95.865181
	8000	750.239139
2	500	0.184659
	1000	1.040094
	2000	6.751338
	4000	50.503975
	8000	402.090099
4	500	0.18658
	1000	0.992238
	2000	6.16012
	4000	43.337454
	8000	296.208879
8	500	0.137782
	1000	0.71423
	2000	4.399711
	4000	31.087262
	8000	220.125232
16	500	0.207544
	1000	0.818238
	2000	4.287706
	4000	27.776949
	8000	213.664072

5.2 Performance graphs







Appendix

The following is the machine information on which the execution run times were calculated
model name : Intel(R) Core(TM) i7-7700HQ CPU @ 2.80GHz
Ram : 16GB
Number of cores : 4