**COMP – 534**
**Parallel Computing**
**Swanand Sanjay Khonde – sk218**

**Assignment 2 – LU Decomposition**

# LU Decomposition using row pivoting.

## 1) Program Execution:

The program takes in input – matrix_size and nworkers, which is the number of threads. After that the program begins by initializing a square matrix of matrix size with random floating-point numbers (double precision) like mentioned in the assignment description. Here I have also parallelized the matrix initialization. To prevent race condition and generate the matrix efficiently each thread initializes a different part of the matrix. This is achieved by using a random generator for each thread which is seeded uniquely.

Once the matrix is initialized, the program will execute the L U Decomposition function, which performs LU decomposition with partial row pivoting.

This involves performing below steps:

1) The program searches the pivot for each iteration (the largest element In the current column).
2) The program then swaps the rows in the matrix and the permutation vector to move the pivot into position. Although we can parallelize the pivot selection but swapping needs to be done in a sequential manner to avoid data races.
3) As we want to factorize the matrix into Lower and Upper triangular matrix. The program then computes L and U matrix. Here we can parallelize the factorization step so that each thread computes a part of L and U matrix.
4) Finally, the program then updates the matrix for the next iteration, which involves subtracting the outer product from the remaining sub matrix. This update is independent for each row and therefore can be parallelized.

After the computation is completed, the program then prints the pi or the P vector and L (Lower triangular matrix) and U (Upper triangular matrix).

## 2) Pseudo Code of parallelization strategy:

<u>Initialize the input matrix with random numbers.</u>

```
Initialize Matrix (matrix, seed):
  Parallel for each row i in matrix:
    Set up random number generator with unique seed for each thread.
    For each column j in row i:
      matrix[i][j] = Generate random number
```

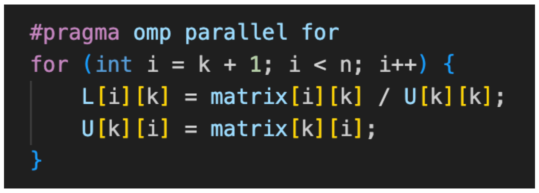
Perform LU decomposition.

```
LU_Decomposition(matrix, size):
  Initialize permutation vector pi with sequential indices

  For k from 0 to size - 1:
    max_val = 0
    k_prime = k


    for each i from k to size:
      If floatabs(matrix[i][k]) > max_val:
        max_val = floatabs(matrix[i][k])
        Update k_prime to i

    If max_val is 0:
      Print error message and exit

    Swap pi[k] with pi[k_prime]

   For i from 0 to n:
    Swap matrix rows k with k_prime in matrix

    For i from 0 to k-1:
      Swap rows k and k_prime in L

   Parallel for each i from k + 1 to size:
      Compute L[i][k] and U[k][i]

    Parallel for each i from k + 1 to size:
      For j from k + 1 to size:
        Update matrix[i][j] with computed L and U values.
```

## 3) Parallelization Strategy

I have parallelized Matrix Initialization. Parallelizing matrix initialization with large matrix size will reduce the time required to fill the matrix with random values. The program initializes the matrix with random values in a parallel section of OpenMP. Each thread initializes a distinct portion of the matrix to ensure workload is distributed evenly.

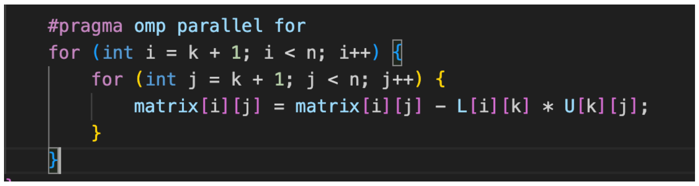
Next, I have parallelized computations in the LU Decomposition function.



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

This section of code is involved in parallelizing the operations involved in computing and updating the L and U matrix. This is done after the pivot has been chosen and the rows have been swapped.

The update of L matrix depends upon the original input matrix and the U matrix. U[k][k] is already set before entering the parallel section. This means that every thread access U[k][k] in a read-only manner, which does not introduce race conditions.
Similarly for the U matrix the update U[k][i] = matrix[k][i]; is independent for each i. The source data matrix[k][i] is not modified within this loop. This ensures that there will be no race condition if we parallelize this for loop.



```
#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] - L[i][k] * U[k][j];
    }
}
```

For this section, the parallel for loop will distribute the iterations of outer loop across the available threads. Each thread is assigned a subset of range from k + 1 to n-1, that are the set of rows of the matrix. This means that each thread will work on the set of assigned rows independently. While the inner loop runs sequentially. For each row i assigned to the thread, it iterates over columns j from k + 1 to n - 1.

## 4) Data Partition.

The data partitioning strategy primarily relies on row-wise division among the threads. This strategy can be efficient where we need to calculate the LU of a large matrix with matrix size of almost 8000. The row-wise partitioning strategy minimizes the use of data already present in the cache and minimizes cache coherence. This also improves the overall efficiency of the parallel program.

## 5) Work Partition

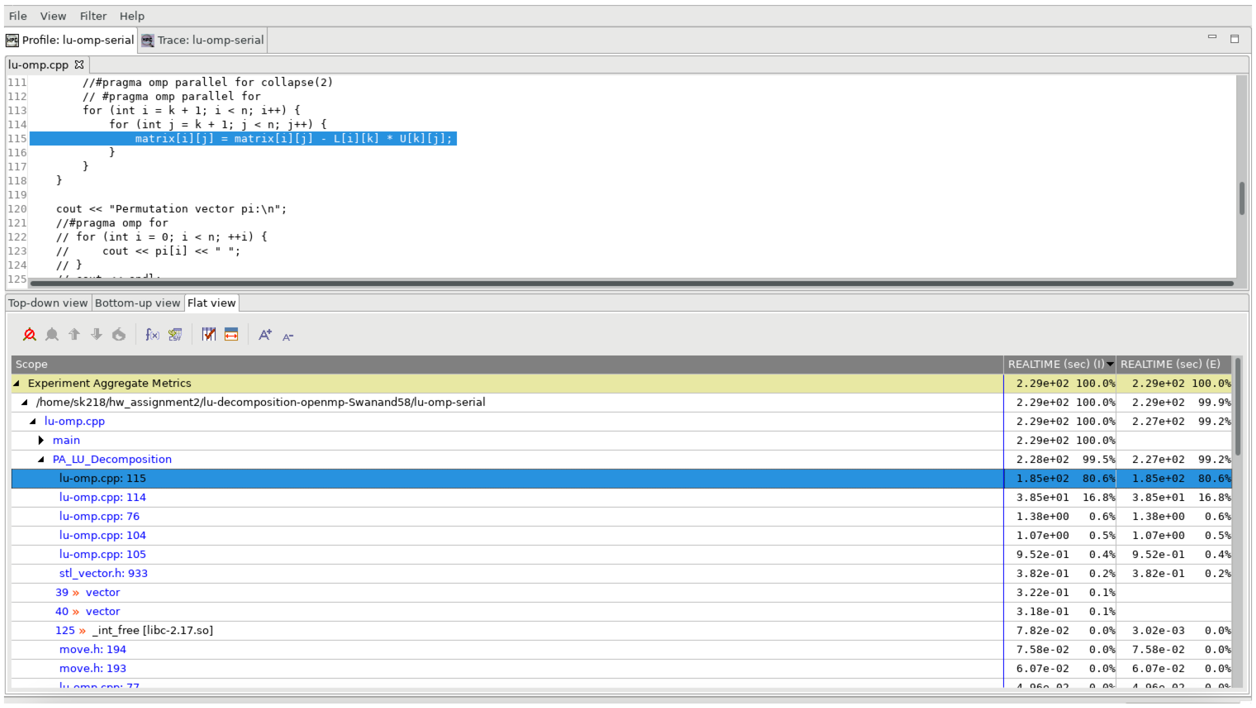
As mentioned in the 3rd point, In the initialization of the matrix section, each thread generates random numbers to fill a portion of the matrix, ensuring that the initialization workload is evenly distributed among all available threads.

After finding the pivot and updating the permutation vector, L and U matrix is calculated in a parallel fashion. As mentioned in the 3rd point, each thread calculates a subset of the values for L and U, which are independent of each other between iterations which are suitable for parallelization. Similarly, the outer for loop is parallelized to update the original matrix. Here each thread is allotted a set of rows which the work on. The inner loop remain sequential for each thread.

### 6) How parallelism is exploited

The program divides the overall computation into smaller chunks that can be executed concurrently. This is effective in LU decomposition because it contains operations that can be performed independently. Also, OpenMP can dynamically balance the workload among the threads ensuring that all the processor cores are utilized. The primary advantage of this parallelization is when the matrix size is very large as just swapping the values at let’s say level 7267 * 7267 can take a lot of time.

### 7) Justify your implementation choices.



```
111        //#pragma omp parallel for collapse(2)
112        // #pragma omp parallel for
113        for (int i = k + 1; i < n; i++) {
114            for (int j = k + 1; j < n; j++) {
115                matrix[i][j] = matrix[i][j] - L[i][k] * U[k][j];
116            }
117        }
118    }
119
120    cout << "Permutation vector pi:\n";
121    //#pragma omp for
122    // for (int i = 0; i < n; ++i) {
123    //     cout << pi[i] << " ";
124    // }
```

| Scope | REALTIME (sec) (I) | | REALTIME (sec) (E) | |
|---|---|---|---|---|
| Experiment Aggregate Metrics | 2.29e+02 | 100.0% | 2.29e+02 | 100.0% |
| /home/sk218/hw_assignment2/lu-decomposition-openmp-Swanand58/lu-omp-serial | 2.29e+02 | 100.0% | 2.29e+02 | 99.9% |
| lu-omp.cpp | 2.29e+02 | 100.0% | 2.27e+02 | 99.2% |
| main | 2.29e+02 | 100.0% | | |
| PA_LU_Decomposition | 2.28e+02 | 99.5% | 2.27e+02 | 99.2% |
| lu-omp.cpp: 115 | 1.85e+02 | 80.6% | 1.85e+02 | 80.6% |
| lu-omp.cpp: 114 | 3.85e+01 | 16.8% | 3.85e+01 | 16.8% |
| lu-omp.cpp: 76 | 1.38e+00 | 0.6% | 1.38e+00 | 0.6% |
| lu-omp.cpp: 104 | 1.07e+00 | 0.5% | 1.07e+00 | 0.5% |
| lu-omp.cpp: 105 | 9.52e-01 | 0.4% | 9.52e-01 | 0.4% |
| stl_vector.h: 933 | 3.82e-01 | 0.2% | 3.82e-01 | 0.2% |
| 39 » vector | 3.22e-01 | 0.1% | | |
| 40 » vector | 3.18e-01 | 0.1% | | |
| 125 » _int_free [libc-2.17.so] | 7.82e-02 | 0.0% | 3.02e-03 | 0.0% |
| move.h: 194 | 7.58e-02 | 0.0% | 7.58e-02 | 0.0% |
| move.h: 193 | 6.07e-02 | 0.0% | 6.07e-02 | 0.0% |

I chose to parallelize the 2 for loops based on the HPC toolkit results that I got on the serialized code. As seen in the above image most of the time taken by the code was on the nested for loops which performs computation of the matrix, and some of the computations to update L and U matrix. If the matrix size is very small then using parallel for loop for calculating L and U matrix will not make sense but when the Size of the matrix becomes very high that is 7000, 8000 then calculating and updating the rows of such a huge matrix can consume a lot of time.

## 8) Synchronizing the parallel Work

OpenMP automatically inserts a barrier at the end of parallel regions and work-sharing constructs. That is, for matrix initialization, updating L and U matrices, and updating the matrix for the next iteration of decomposition, OpenMP automatically includes an implicit barrier at the end of the for loop. This means that the computation of each thread is already completed before reaching the next point in the code. This barrier ensures that all updates to the matrix or the L and U matrices are completed before moving on to the next step in the algorithm, maintaining the correctness of the computation. For other operations which are outside the OpenMP parallel region (finding pivot and swapping rows) have inherent data dependency that require them to be executed in a specific order. By executing these operations serially, your program ensures that all necessary conditions are met before parallel sections begins. There is no explicit synchronization needed for this section.

**Results:**

For matrix size - 8000

| No of Workers | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
|---|---|---|---|---|---|---|---|---|
| Serial | 186 s | | | | | | | |
| | | | | | | | | |
| Parallel | 179 s | 97 s | 73 s | 63 s | 58 s | 56 s | 54 s | 53 s |

| No of Workers | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
|---|---|---|---|---|---|---|---|---|
| Parallel | 50 s | 52 s | 51 s | 53 s | 51 s | 53 s | 53 s | 55 s |

| No of Workers | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 |
|---|---|---|---|---|---|---|---|---|
| Parallel | 55 s | 57 s | 57 s | 57 s | 56 s | 56 s | 57 s | 58 s |

| No of Workers | 25 | 26 | 27 | 28 | 29 | 30 | 31 | 32 |
|---|---|---|---|---|---|---|---|---|
| Parallel | 56 s | 56 s | 56 s | 56 s | 57 s | 58 s | 59 s | 60 s |

Parallel Efficiency on 16 processors compute cluster nots

1 Worker = 186 / (1 * 179) * 100 = 103.91%

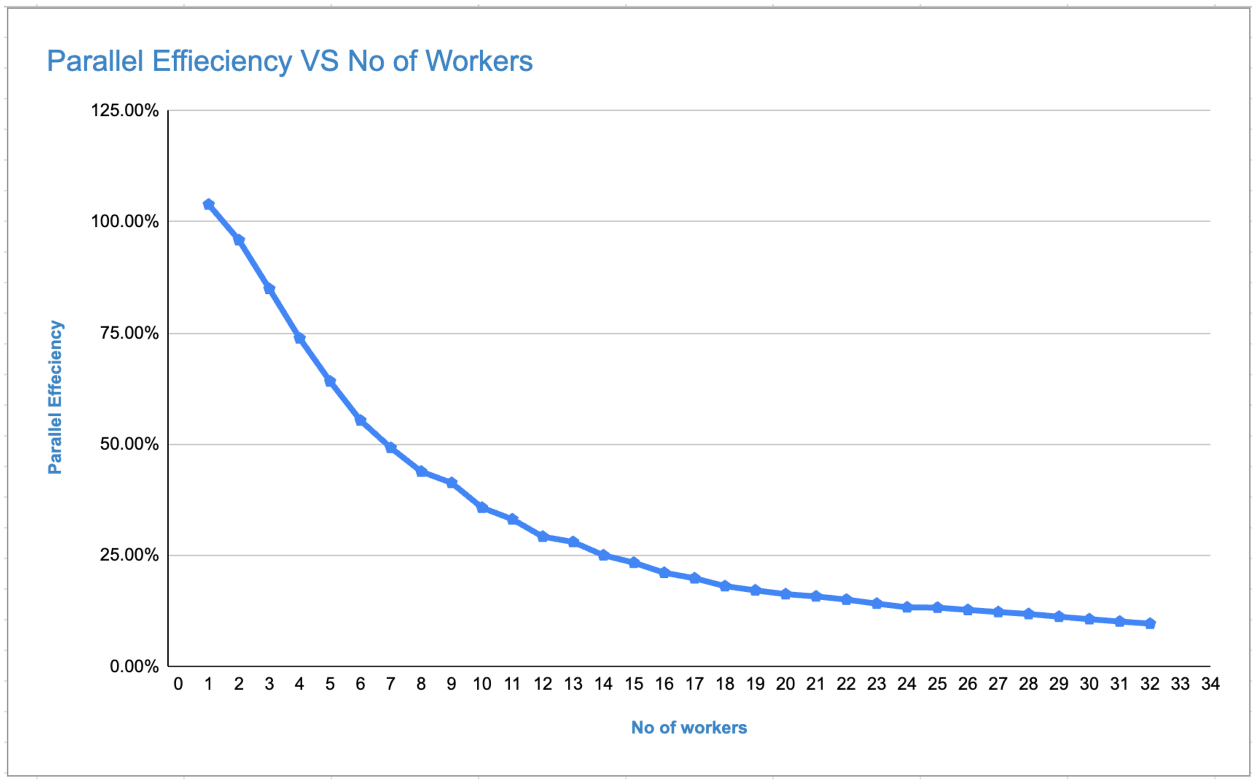
2 Workers = 186 / (2 * 97) * 100 = 95.88%

4 Workers = 186 / (4 * 63) * 100 = 73.81%

8 Workers = 186 / (8 * 53) * 100 = 43.87%

16 Workers = 186 / (16 * 55) * 100 = 21.14%

32 Workers = 186 / (32 * 60) * 100 = 9.69%



NOTE: I have added a flag in the main function. “Bool l2_norm_flag = true”. This will calculate the l2 norm to check the correctness of the program. To disable this implementation please change it to false.