Parallel LU Decomposition with Row Pivoting: A Comparative Study using OpenMP and Pthreads

Naveen Meena 2019CS10378

February 15, 2024

1 Introduction

1.1 LU Decomposition

LU decomposition is a method in numerical linear algebra for decomposing a given square matrix A into the product of a lower triangular matrix L and an upper triangular matrix U. This decomposition is particularly useful for solving linear equations, inverting matrices, and computing determinants. The process can be mathematically represented as:

$$A = LU \tag{1}$$

where

- A is the input square matrix,
- L is a lower triangular matrix with ones on the diagonal, and
- U is an upper triangular matrix.

1.2 Row Pivoting and the Permutation Matrix

In practice, to improve the numerical stability of the LU decomposition, row pivoting is often employed. Row pivoting involves rearranging the rows of A based on the magnitude of the elements in each column before performing the decomposition. This is achieved using a permutation matrix P, leading to the modified equation:

$$PA = LU \tag{2}$$

Here, P is a permutation matrix that rearranges the rows of A such that:

$$P = \begin{pmatrix} 0 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 0 \end{pmatrix} \tag{3}$$

where each row and column contains exactly one entry of 1 with all others being 0, indicating the permutation of rows in A.

The matrices L and U obtained after applying P to A ensures that the decomposition accommodates the row pivoting, thereby enhancing the numerical stability of the operation. This process is crucial for matrices that may lead to large numerical errors if decomposed directly without pivoting.

2 Sequential LU Decomposition

The sequential algorithm for LU decomposition with row pivoting is critical for understanding the computational workload and identifying opportunities for parallelization. Below is the pseudocode that outlines the sequential process:

2.1 Pseudocode for Sequential LU Decomposition

```
Algorithm Sequential_LU_Decomposition
Inputs: A - an n x n matrix
Outputs: L - lower triangular matrix, U - upper triangular matrix, P - permutation matri

Begin
    Initialize L as an n x n identity matrix
    Initialize U as an n x n zero matrix
    Initialize P as an n x n identity matrix (for tracking row permutations)
    For k = 1 to n do
        max = 0
        for i = k to n do
            if abs(A[i, k]) > max then
                max = abs(A[i, k])
                k' = i
            end if
        end for
        if max == 0 then
            error "Matrix is singular"
        end if
        Swap P[k, :] and P[k', :]
        Swap A[k, :] and A[k', :]
        For i = k+1 to n do
            A[i, k] = A[i, k] / A[k, k]
            for j = k+1 to n do
                A[i, j] = A[i, j] - A[i, k] * A[k, j]
            end for
        end for
    end for
    Decompose final A into L and U
End
```

This pseudocode represents the core of the LU decomposition process, including the pivotal step of row pivoting for numerical stability. The complexity and iterative nature of this algorithm, particularly in the search for the pivot and the update of the matrix A, make it a candidate for optimization through parallelization.

2.2 Need for Parallelization

The computational complexity of the sequential execution of LU decomposition is $O(n^3)$, which leads to significant inefficiencies for large matrices.

In the subsequent sections, we explore the implementation of LU decomposition using OpenMP and Pthreads. We highlight the parallelization techniques employed to address the challenges identified in the sequential algorithm, aiming for scalable performance enhancements.

Table 1: Matrix Size vs Time Performance (Sequential Algorithm)

Matrix Size	**Time (s)**
100	0.002
500	0.125
1000	0.867
1500	2.857
2000	6.731
3000	22.758
4000	53.802
5000	105.367
6000	182.386
7000	290.645
8000	430.458

3 Pthread Implementation

The pthread implementation of LU decomposition demonstrates effective memory management techniques to ensure dynamic memory allocation and deallocation, contributing to the overall memory efficiency. Key strategies include:

- Dynamic allocation of matrices to use exactly as much memory as needed.
- In-place modifications of matrices within threads to avoid unnecessary memory usage.
- Proper deallocation of all dynamically allocated memory to prevent leaks.

3.1 Code Snippet for Memory Allocation

```
double** allocateMatrix(int n) {
    double** matrix = new double*[n];
    for (int i = 0; i < n; ++i) {
        matrix[i] = new double[n];
    }
    return matrix;
}
```

Listing 1: Dynamic Memory Allocation for Matrices

3.2 Computational Efficiency

The computational efficiency of the pthread-based LU decomposition stems from its ability to parallelize the row swapping and matrix updating processes. This approach maximizes the utilization of multi-core processors, reducing computation time significantly.

3.3 Parallel Implementation Code Snippet

The following code snippet demonstrates the parallel part of the LU decomposition algorithm, focusing on workload distribution among threads, parallel row swapping, and matrix updating.

3.3.1 Thread Workload Distribution

```
// Calculate rows per thread and distribute the remainder
rowsPerThread = n / t;
remainder = n % t;
startRow = 0;
for (int i = 0; i < t; ++i) {
    threadData[i] = {startRow, startRow + rowsPerThread + (i <
    remainder ? 1 : 0), pivotRow, k, a, l, u, n};
    pthread_create(&threads[i], NULL, &parallelRowSwap, (void*)&
    threadData[i]);
    startRow += rowsPerThread + (i < remainder ? 1 : 0);
}
```

Listing 2: Thread Workload Distribution

3.3.2 Parallel Row Swapping

```
void* parallelRowSwap(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    double temp;
    for (int i = data->startRow; i < data->endRow; ++i) {
        temp = data->a[data->pivotRow][i];
        data->a[data->pivotRow][i] = data->a[data->currentCol][i];
        data->a[data->currentCol][i] = temp;
```

Listing 3: Parallel Row Swapping

3.3.3 Parallel Matrix Update

```c
void* parallelMatrixUpdate(void* arg) {
    ThreadData* data = (ThreadData*)arg;
    for (int i = data->startRow; i < data->endRow; ++i) {
        if (i > data->currentCol) {
            for (int j = data->currentCol + 1; j < data->n; ++j) {
                data->a[i][j] = data->a[i][j] - data->l[i][data->
    currentCol] * data->u[data->currentCol][j];
            }
        }
    }
    pthread_exit(NULL);
}
```

Listing 4: Parallel Matrix Update

Table 2: Performance of Pthread Implementation Across Different Thread Counts and Matrix Sizes (Time in Seconds)

Matrix Size	1 Thread (s)	2 Threads (s)	4 Threads (s)	8 Threads (s)	16 Threads (s)
100	0.01	0.011	0.015	0.027	0.040
500	0.181	0.121	0.096	0.112	0.152
1000	1.125	0.630	0.392	0.397	0.477
2000	8.521	4.512	2.482	2.243	2.375
3000	28.678	14.729	7.912	6.976	7.163
4000	67.430	34.809	19.044	16.652	16.223
5000	132.020	69.313	39.557	32.863	33.360
6000	225.442	121.395	69.006	60.965	58.701
7000	357.943	197.361	139.834	119.937	91.971
8000	536.212	293.257	196.001	175.521	132.864

4 OPENMP Implementation

Memory management is crucial in the OpenMP implementation of LU Decomposition. Dynamic memory allocation and deallocation are carefully handled to ensure efficient use of resources.

4.1 Dynamic Memory Allocation

Memory for matrices is dynamically allocated at the beginning of the program, ensuring that only the necessary amount of memory is used based on the matrix size n.

4.2 Memory Deallocation

Once the computation is complete, all dynamically allocated memory is properly freed, preventing memory leaks and ensuring efficient memory usage throughout the program's lifecycle.

4.3 Computational Efficiency

The OpenMP implementation enhances computational efficiency through parallel processing. Key operations in the LU Decomposition algorithm are parallelized to take advantage of multi-core architectures.

4.3.1 Parallelization Strategy

The algorithm uses OpenMP directives to distribute computational work across multiple threads, significantly reducing the execution time for large matrices.

4.4 Code Snippets

This section presents key snippets from the OpenMP implementation, highlighting parallelization and efficient memory management.

4.4.1 Matrix Initialization and Memory Allocation

```
double** allocateMatrix(int n) {
    double** matrix = new double*[n];
    for (int i = 0; i < n; ++i) {
        matrix[i] = new double[n];
    }
    return matrix;
}

void init(double** a, int n) {
    srand(time(nullptr));
    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            a[i][j] = drand48();
```

Listing 5: Matrix Allocation and Initialization

4.4.2 Parallel LU Decomposition

```
void LU_Decom(double** a, vector<int>& pi, int n, double** l,
    double** u) {
    for (int k = 0; k < n; ++k) {
        // Parallel region for computing l and u elements
        #pragma omp parallel for
        for (int i = k + 1; i < n; ++i) {
            l[i][k] = a[i][k] / u[k][k];
            u[k][i] = a[k][i];
            #pragma omp atomic
            a[i][j] -= l[i][k] * u[k][j];
```

Listing 6: OpenMP Parallel LU Decomposition

Table 3: Performance of OpenMP Implementation Across Different Thread Counts and Matrix Sizes (Time in Seconds)

Matrix Size	1 Thread (s)	2 Threads (s)	4 Threads (s)	8 Threads (s)	16 Threads (s)
100	0.002	0.001	0.002	0.005	0.004
500	0.131	0.084	0.058	0.060	0.064
1000	0.930	0.509	0.292	0.299	0.309
2000	7.257	3.821	2.082	1.920	1.874
3000	24.462	13.081	7.201	6.166	6.138
4000	57.644	30.793	17.428	15.393	14.117
5000	113.379	62.367	37.543	32.644	30.509
6000	196.983	112.334	66.340	55.627	54.790
7000	314.150	177.798	126.327	88.686	85.973
8000	464.872	270.845	204.739	129.909	140.219

5 Conclusion

In comparing Pthreads and OpenMP for parallelizing LU decomposition, both methods offer unique advantages: Pthreads provides fine-grained control over parallel execution, allowing for customized optimization, while OpenMP simplifies development with high-level abstractions for automatic thread management and synchronization.

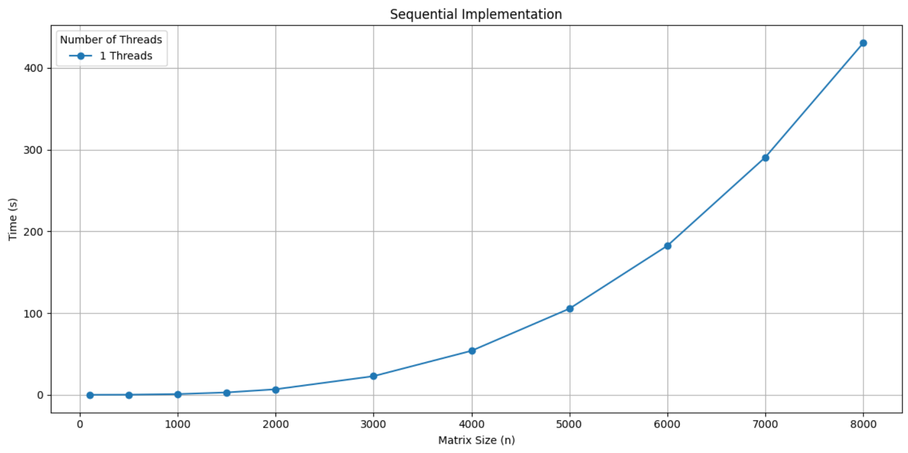

Figure 1: Sequential Implementation

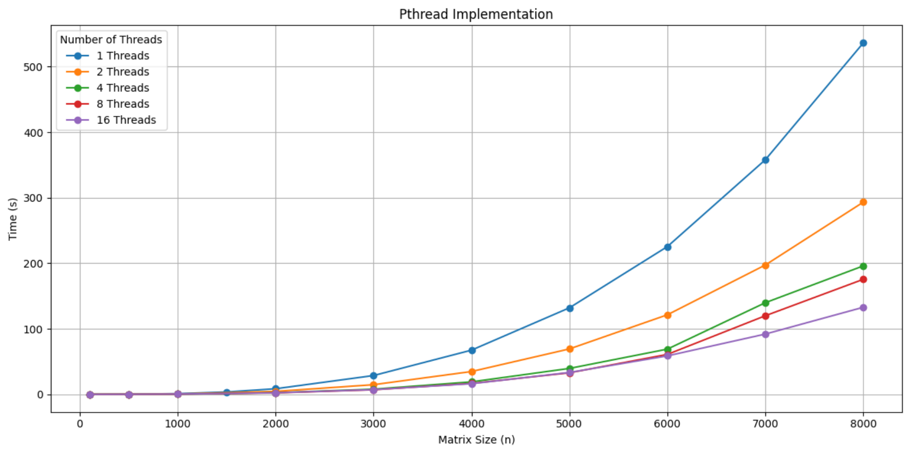

Figure 2: Pthread Implementation

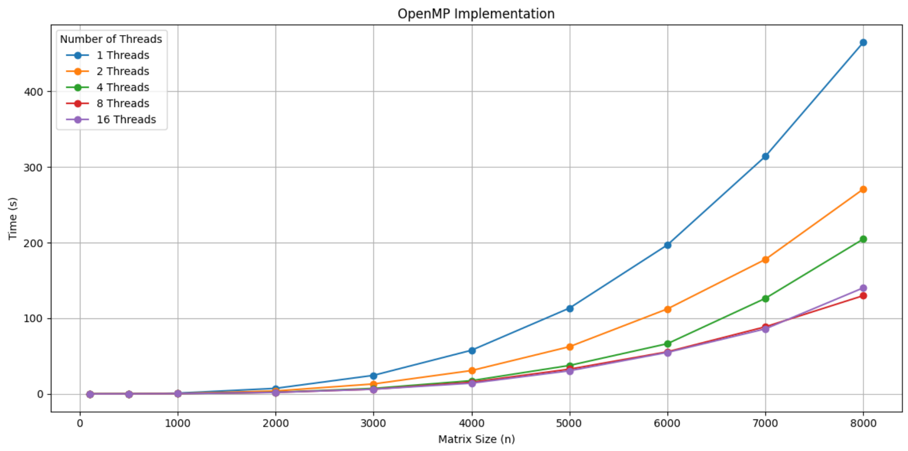

Figure 3: OpenMP Implementation

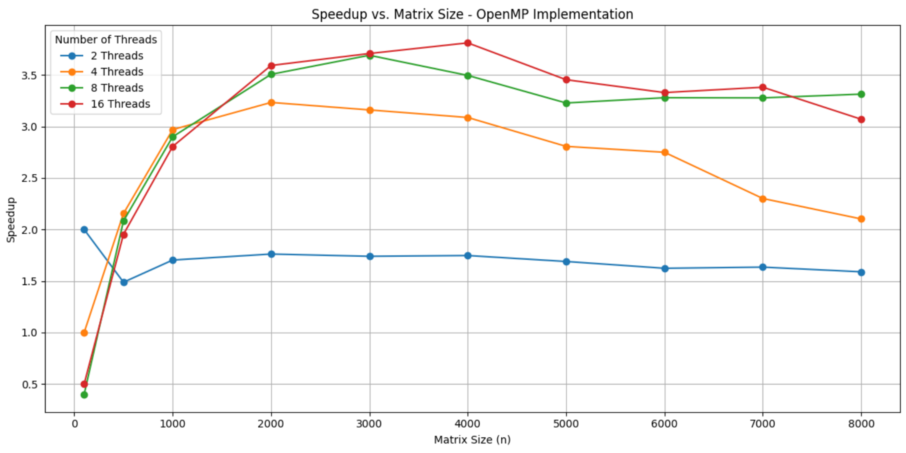

Figure 4: Speed vs. Matrix Size - OpenMP Implementation

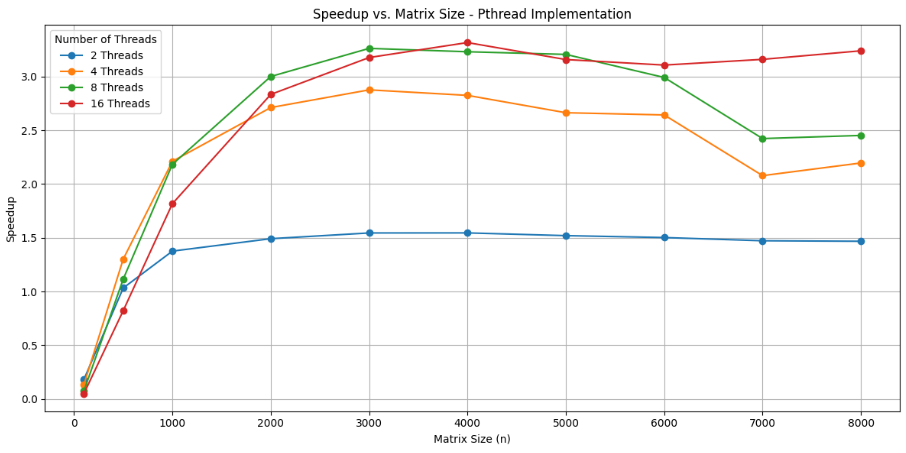

Figure 5: Speed vs. Matrix Size - Pthread Implementation

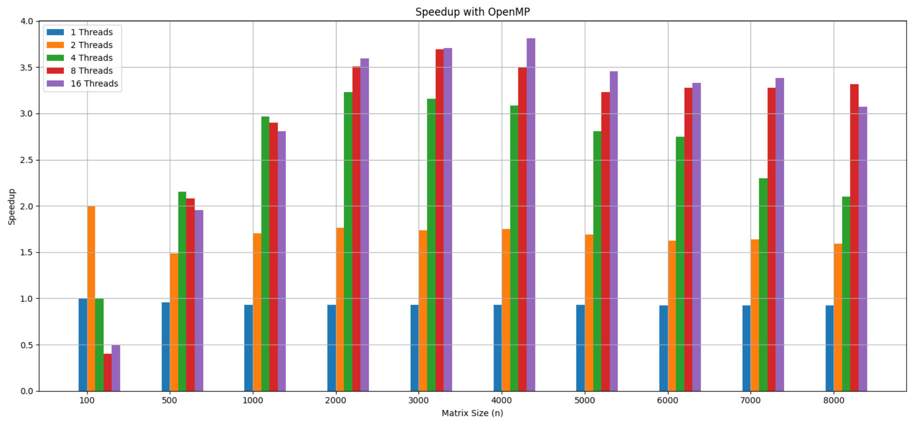

Figure 6: Speed vs. Matrix Size - OpenMP Implementation

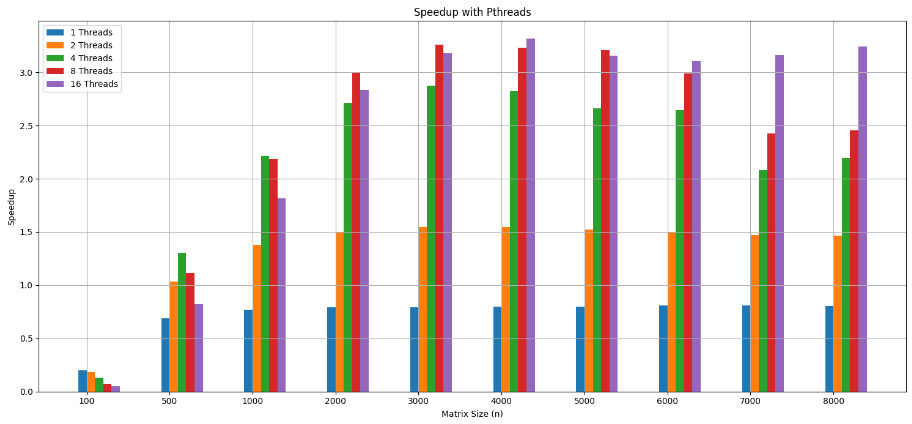

Figure 7: Speed vs. Matrix Size - Pthread Implementation