PARALLEL MATRIX FACTORIZATION

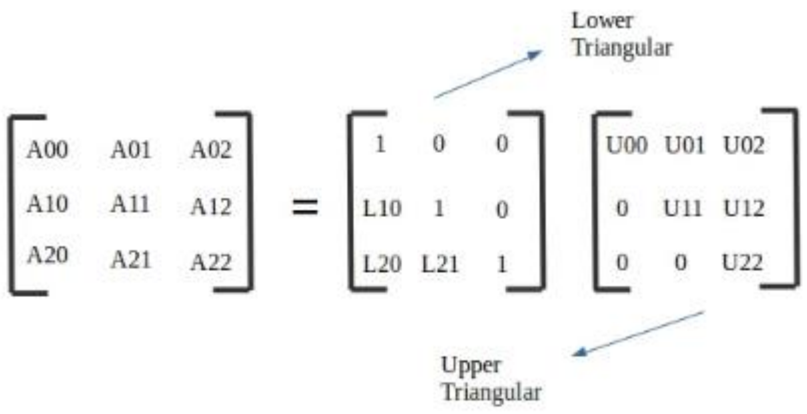
CS3006-PARALLEL AND DISTRIBUTED COMPUTING

SUPERVISOR: DR. GHUFRAN AHMED

Group Members:
21K- 4783 Abdullah Shaikh
21K-3562 Azeem Waqar
21K-3596 Abdul Ghani

Table of Contents

I. Introduction:

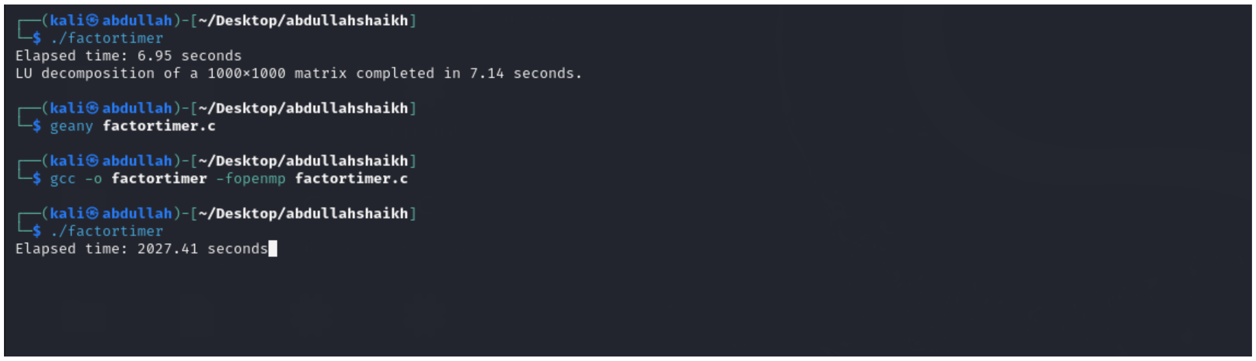


Parallel Matrix Factorization is a technique used to break down a large matrix into smaller, manageable matrices, typically in the form A≈W×HA \approx W \times HA≈W×H, where AAA is the original matrix, and WWW and HHH are factor matrices. This is particularly useful in applications like recommendation systems, where matrix factorization helps identify patterns in large datasets, such as user-item interactions.
In a parallel environment, matrix factorization becomes more efficient by distributing the workload across multiple processors. This involves dividing the matrix operations among different threads or processes, allowing them to compute in parallel rather than sequentially. By parallelizing, we reduce computation time and make it feasible to handle large-scale matrices, especially important in big data contexts.
Typically, **OpenMP** is used for shared-memory parallelization, where threads run concurrently on a single machine, and **MPI** is used for distributed-memory systems, where multiple machines (nodes) work together and communicate to perform the factorization on a large matrix. Each approach places different synchronization and communication constructs to manage parallel computation and ensure correct results.

1. <u>OpenMP Code Execution</u>

1.1 Serial code execution Time:

- 2027.41 / 60 = 33.7 minutes
- The same code on a PC with a 12 GB ram core i7 12 generation takes 30 minutes approx to complete.



```
┌──(kali㉿abdullah)-[~/Desktop/abdullahshaikh]
└─$ ./factortimer
Elapsed time: 6.95 seconds
LU decomposition of a 1000×1000 matrix completed in 7.14 seconds.

┌──(kali㉿abdullah)-[~/Desktop/abdullahshaikh]
└─$ geany factortimer.c

┌──(kali㉿abdullah)-[~/Desktop/abdullahshaikh]
└─$ gcc -o factortimer -fopenmp factortimer.c

┌──(kali㉿abdullah)-[~/Desktop/abdullahshaikh]
└─$ ./factortimer
Elapsed time: 2027.41 seconds
```

Figure 2

1.2 Steps taken to parallelize the code with OpenMP:

In the serial version of the LU decomposition, the program initializes and decomposes the matrix row by row and column by column. This approach is straightforward but slow for large matrices because it performs each operation sequentially. In the OpenMP version, we used parallelization to distribute these tasks across multiple threads, speeding up the process significantly. Here is a breakdown of how and why we applied parallelism:

Matrix Initialization Parallelization

Where: We added #pragma omp parallel for before the outer loop in the initializeMatrix function.

Why: Initializing each element of the matrix is independent of others, making it an ideal candidate for parallel processing. By assigning different parts of the matrix to different threads, the initialization completes much faster.

1. **Parallelizing LU Decomposition Steps**
   - LU decomposition involves creating two matrices, L (lower triangular) and U (upper triangular), by iterating over rows and columns.
   - **Updating Matrix U**:
     - **Where**: We added #pragma omp parallel for before the loop that calculates elements in U[k][j] (for each column j from k onward) in each iteration of the outer loop.
     - **Why**: Each element in U[k][j] depends only on previous values that are already calculated, so each column can be computed independently in parallel. This allows different threads to compute columns simultaneously, speeding up the process.

- **Updating Matrix L**:
  - **Where**: We placed #pragma omp parallel for before the loop that computes elements of L[i][k] (for each row i from k+1 onward).
  - **Why**: Like U, each row in L can be calculated independently once the necessary pivot value is determined. By assigning different rows to different threads, we can significantly reduce the computation time.

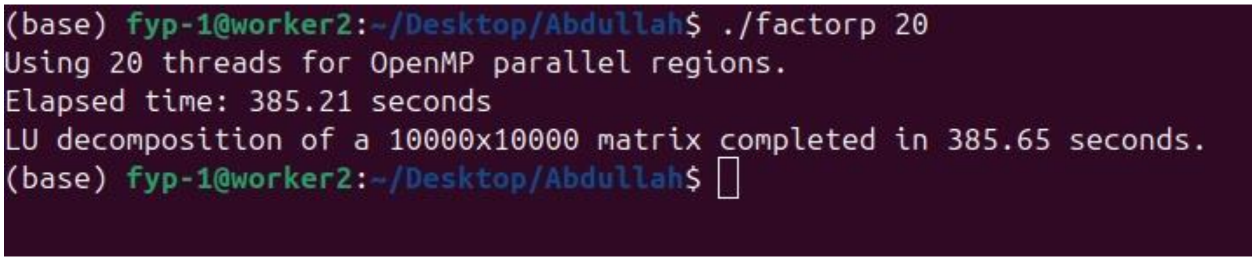
2. **Ensuring Correct Order with Synchronization**
   - **Where**: We used #pragma omp barrier at the end of each step in the main LU decomposition loop.
   - **Why**: A barrier ensures that all threads complete their calculations for the current step before moving on to the next one. Without this, threads could access data that hasn't been fully updated, leading to incorrect results. This synchronization maintains the necessary sequential dependencies in parallel processing.

Summary

By adding #pragma omp parallel for in sections where calculations are independent (matrix initialization and row/column updates), and using barriers to ensure correct sequencing, we distributed the workload efficiently across threads. This reduces the time required for LU decomposition, especially for large matrices, while maintaining the accuracy and integrity of the results.

1.3 Parallelized code execution time:

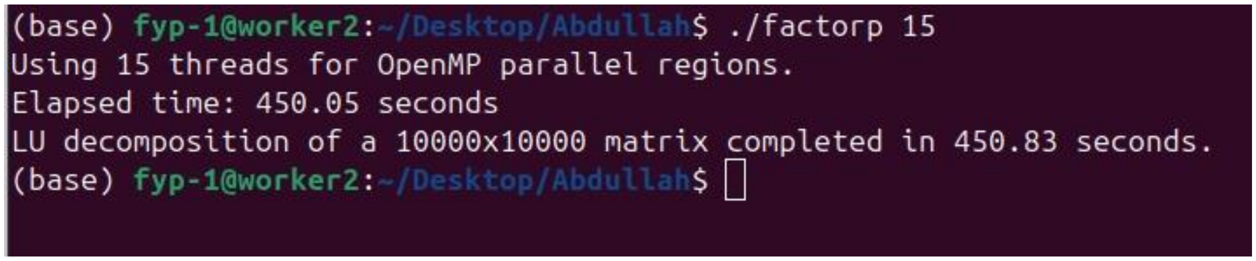
- No of cores: 20
- Time = 6.4 minutes



```
(base) fyp-1@worker2:~/Desktop/Abdullah$ ./factorp 20
Using 20 threads for OpenMP parallel regions.
Elapsed time: 385.21 seconds
LU decomposition of a 10000x10000 matrix completed in 385.65 seconds.
(base) fyp-1@worker2:~/Desktop/Abdullah$ 
```

Figure 3

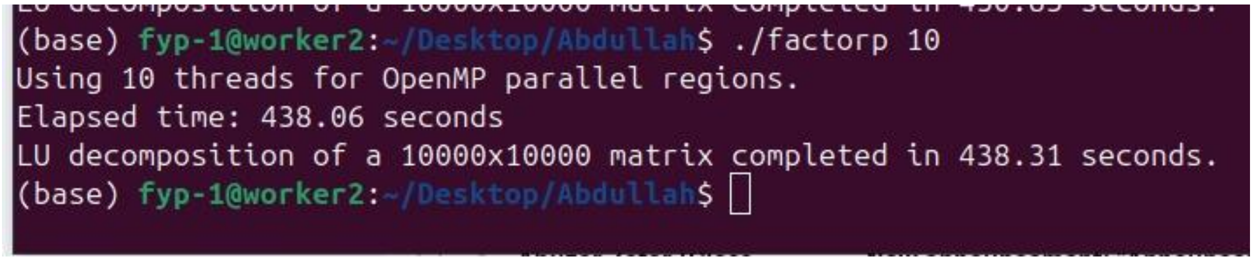
- No of cores: 15
- Time = 7.5 minutes



```
(base) fyp-1@worker2:~/Desktop/Abdullah$ ./factorp 15
Using 15 threads for OpenMP parallel regions.
Elapsed time: 450.05 seconds
LU decomposition of a 10000x10000 matrix completed in 450.83 seconds.
(base) fyp-1@worker2:~/Desktop/Abdullah$
```

Figure 4

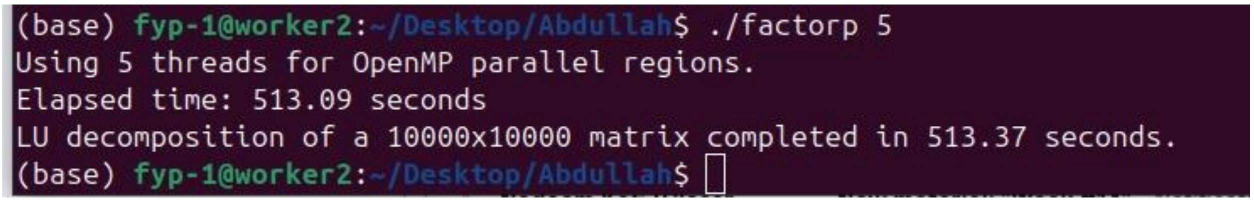
- No of cores: 10
- Time = 7.3 minutes



```
(base) fyp-1@worker2:~/Desktop/Abdullah$ ./factorp 10
Using 10 threads for OpenMP parallel regions.
Elapsed time: 438.06 seconds
LU decomposition of a 10000x10000 matrix completed in 438.31 seconds.
(base) fyp-1@worker2:~/Desktop/Abdullah$
```

Figure 5

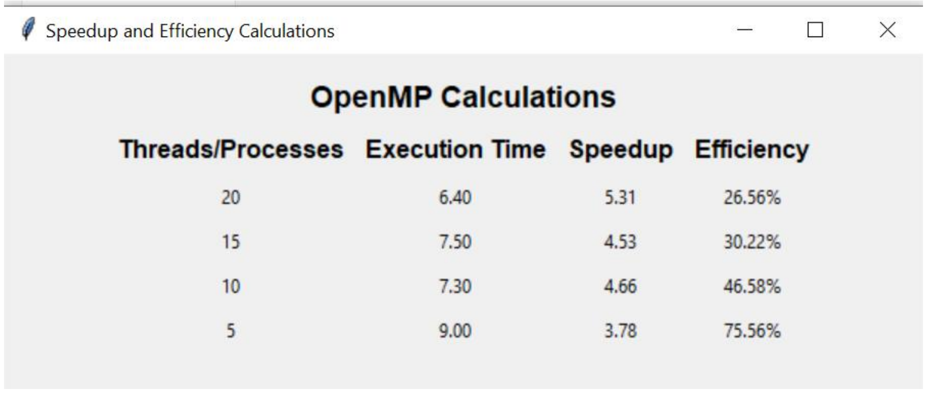
- No of cores: 5
- Time: 9 minutes



```
(base) fyp-1@worker2:~/Desktop/Abdullah$ ./factorp 5
Using 5 threads for OpenMP parallel regions.
Elapsed time: 513.09 seconds
LU decomposition of a 10000x10000 matrix completed in 513.37 seconds.
(base) fyp-1@worker2:~/Desktop/Abdullah$
```

Figure 6

1.4 Numerical calculations:



OpenMP Calculations

Threads/Processes	Execution Time	Speedup	Efficiency
20	6.40	5.31	26.56%
15	7.50	4.53	30.22%
10	7.30	4.66	46.58%
5	9.00	3.78	75.56%

Figure 7

1.5 Graphical analysis

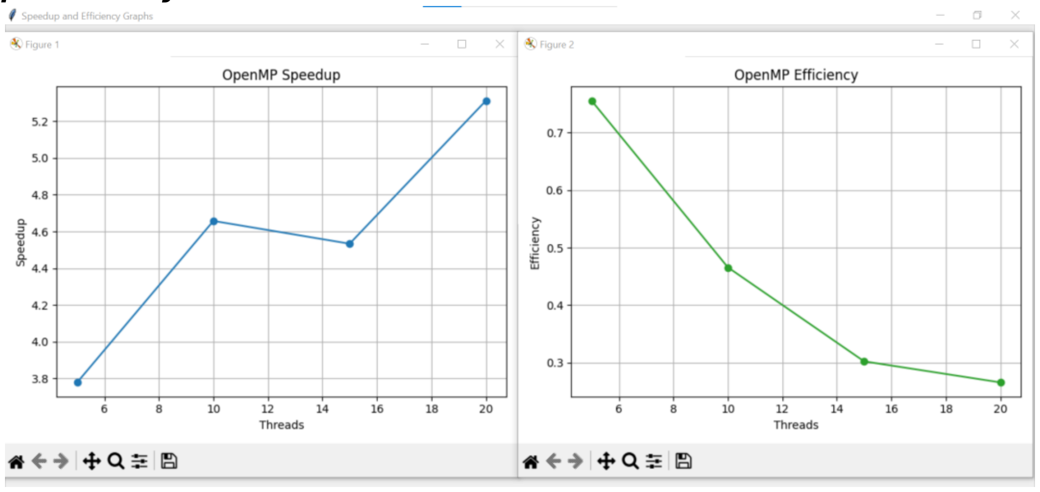


Figure 8

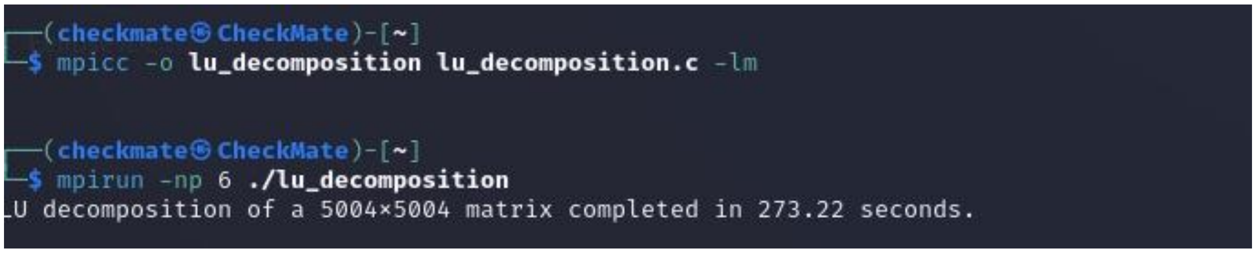
2. <u>Mpi code execution</u>

2.1 Parallelized code execution time:

Total processes: 6

Execution time: 4.5 minutes for 5000x5000

12 + minutes for 10000 x 10000



```
┌──(checkmate㉿CheckMate)-[~]
└─$ mpicc -o lu_decomposition lu_decomposition.c -lm

┌──(checkmate㉿CheckMate)-[~]
└─$ mpirun -np 6 ./lu_decomposition
LU decomposition of a 5004×5004 matrix completed in 273.22 seconds.
```

Figure 9

2.2 Steps taken to parallelize the code with MPI

In a serial LU decomposition, each matrix element is computed in a sequence, which is slow for large matrices. With MPI, we used multiple processes to split up the work, enabling simultaneous computations and inter-process communication to accelerate the LU decomposition.

1. **Matrix Initialization and Distribution**

- **Where**: The initializeMatrix function is called by process 0 to initialize the matrix, and then MPI's MPI_Bcast function broadcasts the entire matrix from process 0 to all other processes.
- **Why**: Distributing the initialized matrix to all processes ensures that each one has access to the complete matrix, allowing for parallel processing of LU decomposition on different rows. Broadcasting minimizes data transfer time, making sure each process gets a consistent version of the matrix at the start.

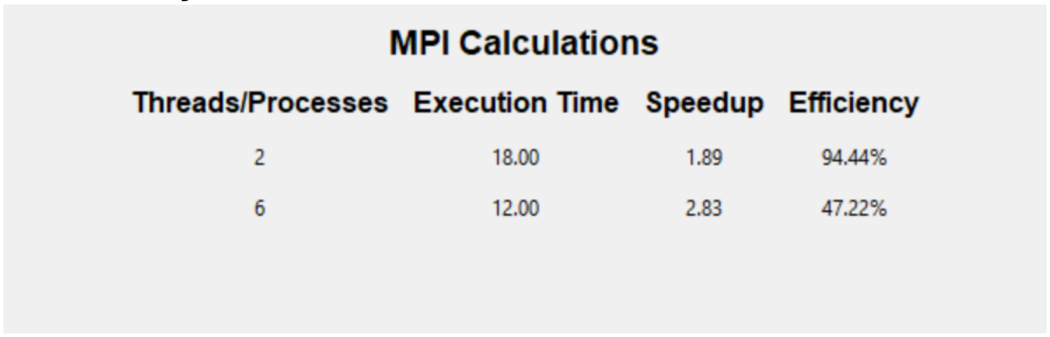
2. **Row-Based Work Division**
   - **Where**: In the luDecomposition function, we calculated rows_per_proc, which defines the number of rows each process will handle. Each process calculates its start_row and end_row based on its rank and the total number of processes.
   - **Why**: By dividing the matrix rows among processes, we can have each process independently work on its portion of the matrix without interference from others. This row-based division allows us to make efficient use of each process's computational resources.
3. **Parallel Calculation of U and Communication with MPI_Bcast**
   - **Where**: For each step k in the outer loop, the process handling row k computes U[k][j] values, which is then broadcasted to all other processes using MPI_Bcast.
   - **Why**: Broadcasting the updated row of U ensures that all processes have the latest values needed to continue their computations. This communication is essential because each process needs accurate U values for subsequent calculations in its assigned rows.
4. **Parallel Calculation of L**
   - **Where**: Each process computes elements of L[i][k] for rows assigned to it, which are updated in parallel across all processes.
   - **Why**: Since the calculation of each row in L only depends on completed rows in U, each process can compute its rows independently after receiving the latest U data. This approach distributes the work of constructing L across processes, speeding up the decomposition.
5. **Ensuring Synchronization Across Processes**
   - **Where**: By using MPI_Bcast to communicate updated rows and MPI_Barrier to synchronize where necessary, each process is synchronized to prevent moving to the next step before all processes have the required data.
   - **Why**: Synchronization maintains the accuracy of the LU decomposition across all processes. Without these communications, processes would proceed with incomplete data, leading to errors. Broadcasting and barriers

ensure that each step in the decomposition is completed in a coordinated manner.

Summary

Through MPI, we parallelized LU decomposition by dividing rows among processes, using MPI_Bcast to communicate necessary data (updated rows in U), and ensuring that all processes are synchronized before each new step. This setup allows processes to work on different sections simultaneously, accelerating the decomposition while ensuring accuracy through essential inter-process communication.

2.3 Numerical Analysis:



MPI Calculations

Threads/Processes	Execution Time	Speedup	Efficiency
2	18.00	1.89	94.44%
6	12.00	2.83	47.22%

Figure 10

2.4 Graphical Analysis:

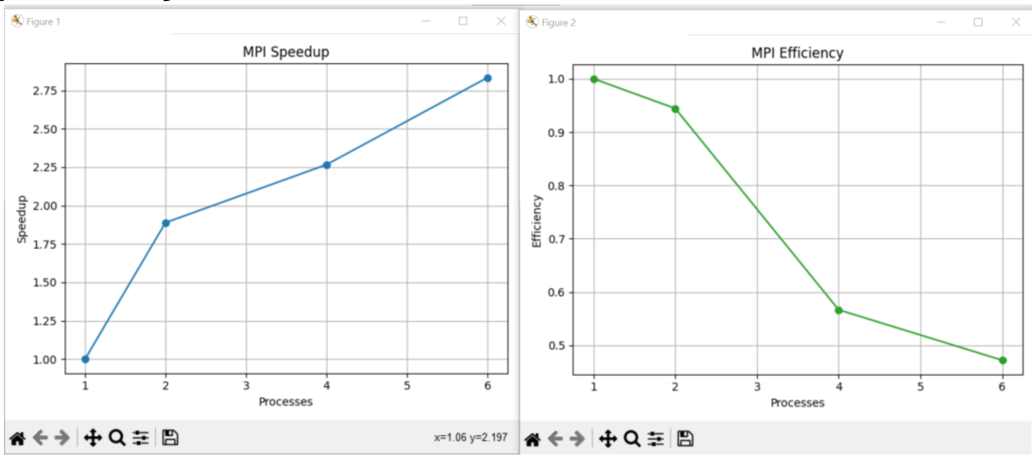


Figure 11

3.0 Codes:

3.1 Serial code

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 10000 // Define a large matrix size

void initializeMatrix(double** matrix) {
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      matrix[i][j] = (i == j) ? 1.0 : rand() % 100 + 1; // Initialize with random values and
diagonal 1s.
    }
  }
}

void luDecomposition(double** matrix, double** L, double** U) {
  int i, j, k;

  // Initialize L to identity matrix and U to zero
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      L[i][j] = (i == j) ? 1.0 : 0.0;
      U[i][j] = 0.0;
    }
  }

  // Perform LU Decomposition
  for (k = 0; k < N; k++) {
    // Compute U
    for (j = k; j < N; j++) {
      U[k][j] = matrix[k][j];
      for (i = 0; i < k; i++) {
        U[k][j] -= L[k][i] * U[i][j];
      }
    }

    // Compute L
```

```
        for (i = k + 1; i < N; i++) {
            L[i][k] = matrix[i][k];
            for (j = 0; j < k; j++) {
                L[i][k] -= L[i][j] * U[j][k];
            }
            L[i][k] /= U[k][k];
        }
    }
}

int main() {
    // Allocate memory for matrices
    double** matrix = (double*)malloc(N * sizeof(double));
    double** L = (double*)malloc(N * sizeof(double));
    double** U = (double*)malloc(N * sizeof(double));

    for (int i = 0; i < N; i++) {
        matrix[i] = (double*)malloc(N * sizeof(double));
        L[i] = (double*)malloc(N * sizeof(double));
        U[i] = (double*)malloc(N * sizeof(double));
    }

    initializeMatrix(matrix);

    // Measure the time taken for LU decomposition
    clock_t start = clock();
    luDecomposition(matrix, L, U);
    clock_t end = clock();

    double time_taken = (double)(end - start) / CLOCKS_PER_SEC;
    printf("LU decomposition of a %dx%d matrix completed in %.2f seconds.\n", N, N,
time_taken);

    // Free allocated memory
    for (int i = 0; i < N; i++) {
        free(matrix[i]);
        free(L[i]);
        free(U[i]);
    }
    free(matrix);
    free(L);
    free(U);

    return 0;
```

```
}
```

3.2 OpenMP parallelized code:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
#include <unistd.h> // For sleep function
#include <pthread.h> // For threading

#define N 10000 // Define a large matrix size

// Structure to pass arguments to the timer thread
typedef struct {
  int running;
} TimerArgs;

void* displayTimer(void* args) {
  TimerArgs* timerArgs = (TimerArgs*)args;
  double start_time = omp_get_wtime();

  while (timerArgs->running) {
    double elapsed = omp_get_wtime() - start_time;
    printf("\rElapsed time: %.2f seconds", elapsed);
    fflush(stdout); // Ensure the output is displayed immediately
    sleep(1); // Update every second
  }

  return NULL;
}
void initializeMatrix(double** matrix) {
  #pragma omp parallel for
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
      matrix[i][j] = (i == j) ? 1.0 : rand() % 100 + 1; // Initialize with random values and
diagonal 1s.
    }
  }
}

void luDecomposition(double** matrix, double** L, double** U) {
  int i, j, k;
  // Initialize L to identity matrix and U to zero
  #pragma omp parallel for private(j)
  for (i = 0; i < N; i++) {
    for (j = 0; j < N; j++) {
      L[i][j] = (i == j) ? 1.0 : 0.0;
      U[i][j] = 0.0;
    }
  }

  // Perform LU Decomposition
  for (k = 0; k < N; k++) {
    // Compute U (parallelize outer loop)
    #pragma omp parallel for private(i)
    for (j = k; j < N; j++) {
      U[k][j] = matrix[k][j];
      for (i = 0; i < k; i++) {
        U[k][j] -= L[k][i] * U[i][j];
      }
    }

    // Compute L (parallelize outer loop)
    #pragma omp parallel for private(j)
    for (i = k + 1; i < N; i++) {
      L[i][k] = matrix[i][k];
      for (j = 0; j < k; j++) {
        L[i][k] -= L[i][j] * U[j][k];
      }
      L[i][k] /= U[k][k];
    }
  }
}

int main(int argc, char* argv[]) {
  // Check if the number of threads is provided
  if (argc != 2) {
    fprintf(stderr, "Usage: %s <number_of_threads>\n", argv[0]);
    return EXIT_FAILURE;
  }

  // Parse the number of threads from the command line
  int num_threads = atoi(argv[1]);
  if (num_threads <= 0) {
    fprintf(stderr, "Number of threads must be greater than 0.\n");
```

```
        return EXIT_FAILURE;
    }

    // Set the number of threads for OpenMP
    omp_set_num_threads(num_threads);

    printf("Using %d threads for OpenMP parallel regions.\n", num_threads);

    // Allocate memory for matrices
    double** matrix = (double*)malloc(N * sizeof(double));
    double** L = (double*)malloc(N * sizeof(double));
    double** U = (double*)malloc(N * sizeof(double));

    for (int i = 0; i < N; i++) {
        matrix[i] = (double*)malloc(N * sizeof(double));
        L[i] = (double*)malloc(N * sizeof(double));
        U[i] = (double*)malloc(N * sizeof(double));
    }

    initializeMatrix(matrix);

    // Timer thread setup
    TimerArgs timerArgs = {1}; // Set running to true
    pthread_t timerThread;
    pthread_create(&timerThread, NULL, displayTimer, (void*)&timerArgs);

    // Measure the time taken for LU decomposition
    double start = omp_get_wtime();
    luDecomposition(matrix, L, U);
    double end = omp_get_wtime();

    // Stop the timer
    timerArgs.running = 0;
    pthread_join(timerThread, NULL); // Wait for the timer thread to finish

    double time_taken = end - start;
    printf("\nLU decomposition of a %dx%d matrix completed in %.2f seconds.\n", N, N,
time_taken);

    // Free allocated memory
    for (int i = 0; i < N; i++) {
        free(matrix[i]);
        free(L[i]);
        free(U[i]);
    }
    free(matrix);
    free(L);
    free(U);

    return 0;
}
```

3.3 Mpi parallelized code

```
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

#define N 10000 // Define a large matrix size

void initializeMatrix(double* matrix) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            matrix[i * N + j] = (i == j) ? 1.0 : rand() % 100 + 1;
        }
    }
}

void luDecomposition(double* matrix, double* L, double* U, int rank, int size) {
    int i, j, k;
    int rows_per_proc = N / size;
    int start_row = rank * rows_per_proc;
    int end_row = (rank == size - 1) ? N : start_row + rows_per_proc;

    // Initialize L to identity and U to zero
    for (i = start_row; i < end_row; i++) {
        for (j = 0; j < N; j++) {
            L[i * N + j] = (i == j) ? 1.0 : 0.0;
            U[i * N + j] = 0.0;
        }
    }

    for (k = 0; k < N; k++) {
        // Broadcast the k-th row of U to all processes
        if (rank == k / rows_per_proc) {
            for (j = k; j < N; j++) {
                U[k * N + j] = matrix[k * N + j];
            }
        }
        MPI_Bcast(&U[k * N + k], N - k, MPI_DOUBLE, k / rows_per_proc, MPI_COMM_WORLD);

        // Compute L
        for (i = (k >= start_row) ? k + 1 : start_row; i < end_row; i++) {
          L[i * N + k] = matrix[i * N + k] / U[k * N + k];
          for (j = k + 1; j < N; j++) {
            matrix[i * N + j] -= L[i * N + k] * U[k * N + j];
          }
        }

        // Compute U
        for (j = k + 1; j < N; j++) {
          if (rank == k / rows_per_proc) {
            for (i = 0; i < k; i++) {
              U[k * N + j] -= L[k * N + i] * U[i * N + j];
            }
          }
        }
      }
    }

    int main(int argc, char* argv[]) {
      int rank, size;

      MPI_Init(&argc, &argv);
      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &size);

      if (N % size != 0) {
        if (rank == 0) {
          fprintf(stderr, "Matrix size must be divisible by the number of processes.\n");
        }
        MPI_Finalize();
        return EXIT_FAILURE;
      }

      double* matrix = (double*)malloc(N * N * sizeof(double));
      double* L = (double*)malloc(N * N * sizeof(double));
      double* U = (double*)malloc(N * N * sizeof(double));

      if (rank == 0) {
        initializeMatrix(matrix);
      }
```

```
  MPI_Bcast(matrix, N * N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  double start_time = MPI_Wtime();
  luDecomposition(matrix, L, U, rank, size);
  double end_time = MPI_Wtime();

  if (rank == 0) {
    printf("LU decomposition of a %dx%d matrix completed in %.2f seconds.\n", N, N,
end_time - start_time);
  }

  free(matrix);
  free(L);
  free(U);

  MPI_Finalize();
  return 0;
}
```

4.0 Conclusion

The implementation of LU decomposition using **OpenMP** and **MPI** demonstrates the effectiveness of parallelization for speeding up computations on large matrices.

- **OpenMP**:
  - The **speedup** consistently increased as more threads were utilized, showcasing the efficiency of dividing tasks across multiple cores.
  - However, the **efficiency** declined with an increasing number of threads. This is due to factors like thread synchronization overhead and diminishing returns from splitting smaller workloads.
- **MPI**:
  - Like OpenMP, **speedup** improved with an increase in processes, reflecting the advantage of distributing the computation across multiple nodes.
  - The **efficiency** decreased as the number of processes increased, mainly because of communication overhead and idle time in synchronization among processes.

The End