LU Decomposition using Serial, OpenMp and MPI

Project members:

1. 21K-4896 – Abdullah Islam

2. 21K-3347 – Talha Minhaj

3. 21K-4896 – 21K-4911 – Abdul Moiz

Submitted to:

Dr Ghufran

Project Background

The project aimed to implement LU Decomposition using Doolittle method in C, exploring three different approaches: serial, OpenMP, and MPI. The goal was to decompose a matrix efficiently and compare the performance of these parallelization techniques.

Objectives

Implement LU Decomposition using Doolittle method in C.

Explore and compare the performance of serial, OpenMP, and MPI approaches.

Analyze the efficiency and speed of each implementation.

Scope and Deliverables

Scope:

Implementation of LU Decomposition using Doolittle method.

Serial implementation for baseline comparison.

OpenMP implementation for shared-memory parallelism.

MPI implementation for distributed-memory parallelism.

Deliverables:

Functional code for each implementation.

Performance analysis and comparison report.

Documentation detailing the process and results.

Methodology

The project utilized C programming language along with:

Serial implementation: Utilized basic C programming techniques.

OpenMP: Leveraged shared-memory parallelism through OpenMP directives.

MPI: Employed message-passing interface for distributed-memory parallelism.

Tools: OpenMP libraries, MPI libraries.

Code Screenshots

Sequential:

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

#define MAX_N 100  // Maximum value for N

void doolittle_LU_decomposition(int A[MAX_N][MAX_N], float L[MAX_N][MAX_N], float U[MAX_N][MAX_N], int N) {
    int i, j, k;

    // Initialize matrices L and U
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            U[i][j] = A[i][j];
            if (i == j)
                L[i][j] = 1.0;
            else
                L[i][j] = 0.0;
}
    // Perform Doolittle LU decomposition
    for (k = 0; k < N; k++) {
        for (i = k + 1; i < N; i++) {
            if (U[k][k] == 0.0) {
                printf("Singular matrix. LU decomposition cannot be performed.\n");
                return;
            }
            float factor = U[i][k] / U[k][k];
            L[i][k] = factor;
            for (j = k; j < N; j++) {
                U[i][j] -= factor * U[k][j];
            }}}}

int main() {
    int N, i, j;
    int A[MAX_N][MAX_N];

    // Seed for random number generation
    srand(time(NULL));
```

```c
    // Input matrix size N
    printf("Enter the size of the matrix (N): ");
    scanf("%d", &N);

    if (N <= 0 || N > MAX_N) {
        printf("Invalid matrix size. Exiting...\n");
        return 1;
    }
    float L[MAX_N][MAX_N] = {0}, U[MAX_N][MAX_N] = {0};
    clock_t start_time = clock(); // Start time measurement
    // Generate random matrix A between -100 and 100
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            A[i][j] = rand() % 201 - 100;
        }
    }
    doolittle_LU_decomposition(A, L, U, N);
    clock_t end_time = clock(); // End time measurement for LU decomposition
    // Displaying L, U matrices
    printf("\nMatrix L:\n");
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            printf("%.2f ", L[i][j]);
        }
        printf("\n");
    }
    printf("\nMatrix U:\n");
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            printf("%.2f ", U[i][j]);
        }
        printf("\n");
    }
    // Display time taken for LU decomposition
    double execution_time = ((double) (end_time - start_time)) / CLOCKS_PER_SEC;
    printf("\nTime taken for Doolittle LU decomposition: %.6f seconds\n", execution_time);
    return 0;
}
```

OpenMP:

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

#define MAX_N 100  // Maximum value for N

void lu_decomposition_doolittle(int A[MAX_N][MAX_N], float L[MAX_N][MAX_N], float U[MAX_N][MAX_N], int N) {
    int i, j, k;

    for (i = 0; i < N; i++) {
        L[i][i] = 1;

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

int main() {
    int N, num_threads, i, j;
    int A[MAX_N][MAX_N];
    // Input number of threads
    printf("Enter the number of threads: ");
    scanf("%d", &num_threads);

    omp_set_num_threads(num_threads);

    // Seed for random number generation
    srand(time(NULL));
    // Input matrix size N
    printf("Enter the size of the matrix (N): ");
    scanf("%d", &N);
    if (N <= 0 || N > MAX_N) {
        printf("Invalid matrix size. Exiting...\n");
        return 1;
    }
    double start_time, end_time;

    float L[MAX_N][MAX_N] = {0}, U[MAX_N][MAX_N] = {0};
    // Generate random matrix A between -100 and 100
    #pragma omp parallel for shared(A, N) private(i, j) schedule(static)
    for (i = 0; i < N; i++) {
        #pragma omp parallel for shared(A, N) private(j) schedule(static
        for (j = 0; j < N; j++) {
            A[i][j] = rand() % 201 - 100;
        }
    }
    // Display Matrix A
    printf("\nMatrix A:\n");
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            printf("%d ", A[i][j]);
        }
        printf("\n");
    }
    start_time = omp_get_wtime();
    // Perform LU decomposition
    lu_decomposition_doolittle(A, L, U, N);
    end_time = omp_get_wtime();
    // Displaying L, U matrices
    printf("\nMatrix L:\n");
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            printf("%.2f ", L[i][j]);
        }
        printf("\n");
    }
```

```
    }
    printf("\nMatrix U:\n");
    for (i = 0; i < N; i++) {
        for (j = 0; j < N; j++) {
            printf("%.2f ", U[i][j]);
        }
        printf("\n");
    }
    // Display time taken for LU decomposition
    printf("\nTime taken for LU decomposition: %.6f seconds\n", end_time - start_time);
    return 0;
}
```

MPI:

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

#define MAX_N 100  // Maximum value for N
#define BLOCK_SIZE 10  // Block size for decomposition
void doolittle_LU_decomposition(int A[MAX_N][MAX_N], float L[MAX_N][MAX
    int i, j, k;

    for (i = 0; i < N; i++) {
        for (j = i; j < N; j++) {
            float sum = 0.0;
            for (k = 0; k < i; k++) {
                sum += (L[i][k] * U[k][j]);
            }
            U[i][j] = A[i][j] - sum; }
        for (j = i; j < N; j++) {
            if (i == j)
                L[i][i] = 1;
            else {
                float sum = 0.0;
                for (k = 0; k < i; k++) {
                    sum += (L[j][k] * U[k][i]);
                }
                L[j][i] = (A[j][i] - sum) / U[i][i]; }}}}

int main(int argc, char *argv[]) {
    int N, my_rank, num_procs, i, j;
    int A[MAX_N][MAX_N];
    float L[MAX_N][MAX_N] = {0}, U[MAX_N][MAX_N] = {0};
    float *recvL = NULL, *recvU = NULL;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &num_procs);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    if (num_procs <= 0 || num_procs > MAX_N) {
```

```
        printf("Invalid number of processes. Exiting...\n");
        MPI_Finalize();
        return 1;
    }
    if (my_rank == 0) {
        printf("Enter the size of the matrix (N): ");
        scanf("%d", &N);
        if (N <= 0 || N > MAX_N) {
            printf("Invalid matrix size. Exiting...\n");
            MPI_Finalize();
            return 1;
        }
    }
    MPI_Bcast(&N, 1, MPI_INT, 0, MPI_COMM_WORLD);
    int blockSize = N / num_procs;
    if (my_rank == 0) {
        srand(time(NULL));
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                A[i][j] = rand() % 201 - 100;
            }
        }
        recvL = (float *)malloc(N * N * sizeof(float));
        recvU = (float *)malloc(N * N * sizeof(float));
    }
    MPI_Bcast(A, MAX_N * MAX_N, MPI_INT, 0, MPI_COMM_WORLD);
    double start_time = MPI_Wtime();
    doolittle_LU_decomposition(A, L, U, N, my_rank, num_procs);
    double end_time = MPI_Wtime();
    MPI_Gather(&L[my_rank * blockSize][0], blockSize * N, MPI_FLOAT, recvL, blockSize * N, MPI_FLOAT, 0, MPI_COMM_WORLD);
    MPI_Gather(&U[my_rank * blockSize][0], blockSize * N, MPI_FLOAT, recvU, blockSize * N, MPI_FLOAT, 0, MPI_COMM_WORLD);
    if (my_rank == 0) {
        printf("\nMatrix A:\n");
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                printf("%d ", A[i][j]);
            }
```

```
            }
            printf("\n");
        }
        printf("\nMatrix L:\n");
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                printf("%.2f ", recvL[i * N + j]);
            }
            printf("\n");
        }
        printf("\nMatrix U:\n");
        for (i = 0; i < N; i++) {
            for (j = 0; j < N; j++) {
                printf("%.2f ", recvU[i * N + j]);
            }
            printf("\n");
        }
        printf("\nTime taken for Doolittle's LU decomposition: %.6f seconds\n", end_time - start_time);
        free(recvL);
        free(recvU);
    }
    MPI_Finalize();
    return 0;
}
```

Results and Outcomes

Execution Times

Serial Code

Matrix Size (N)	Execution Time (s)
4	0.000006
100	0.005901
500	0.558316

1000	4.455489
5000	766.240751

OpenMP Code

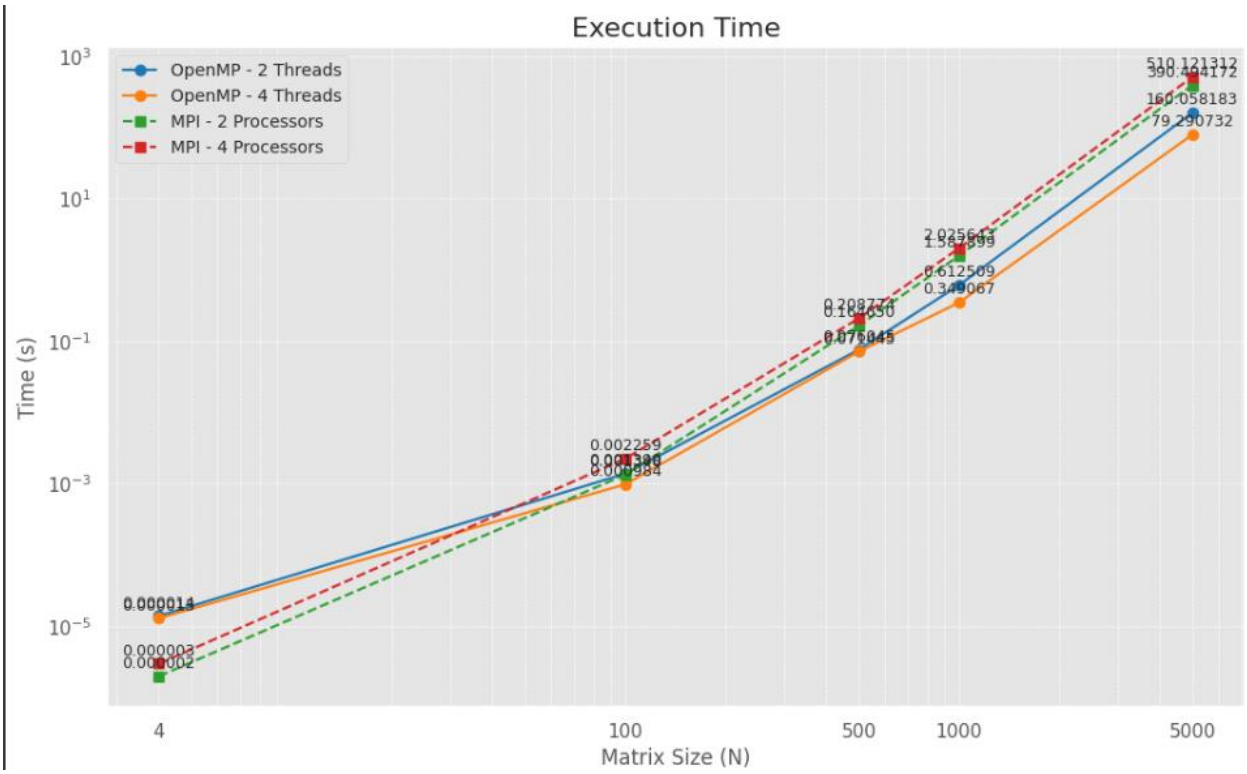
Matrix Size (N)	Threads	Execution Time (s)
4	2	0.000014
4	4	0.000013
4	8	0.000592
100	2	0.001390
100	4	0.000984
500	2	0.076045
500	4	0.071445
1000	2	0.612509
1000	4	0.349067
5000	2	160.058183
5000	4	79.290732

MPI Code

Matrix Size (N)	Processors	Execution Time (s)
4	2	0.000002
4	4	0.000003
100	2	0.001348
100	4	0.002259
500	2	0.164650
500	4	0.208774
1000	2	1.587899
1000	4	2.025643
5000	2	390.494172
5000	4	510.121312

Key Observations

- For **OpenMP**, increasing threads generally improves execution time, but diminishing returns appear for small matrices (N=4).
- **MPI** performance doesn't always scale efficiently with processors due to communication overhead for smaller matrices.
- **Large matrices (N=5000)** benefit significantly from parallelism, but MPI overhead is noticeable compared to OpenMP.



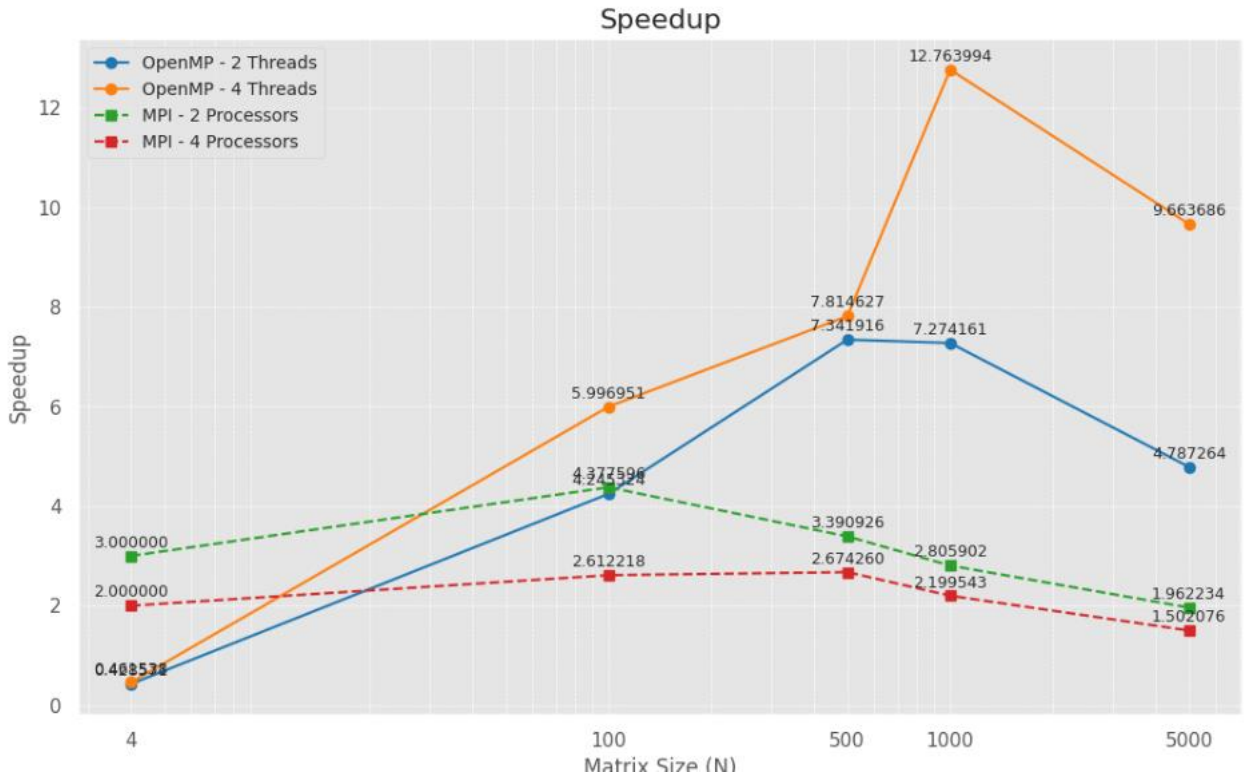
Performance Metrics

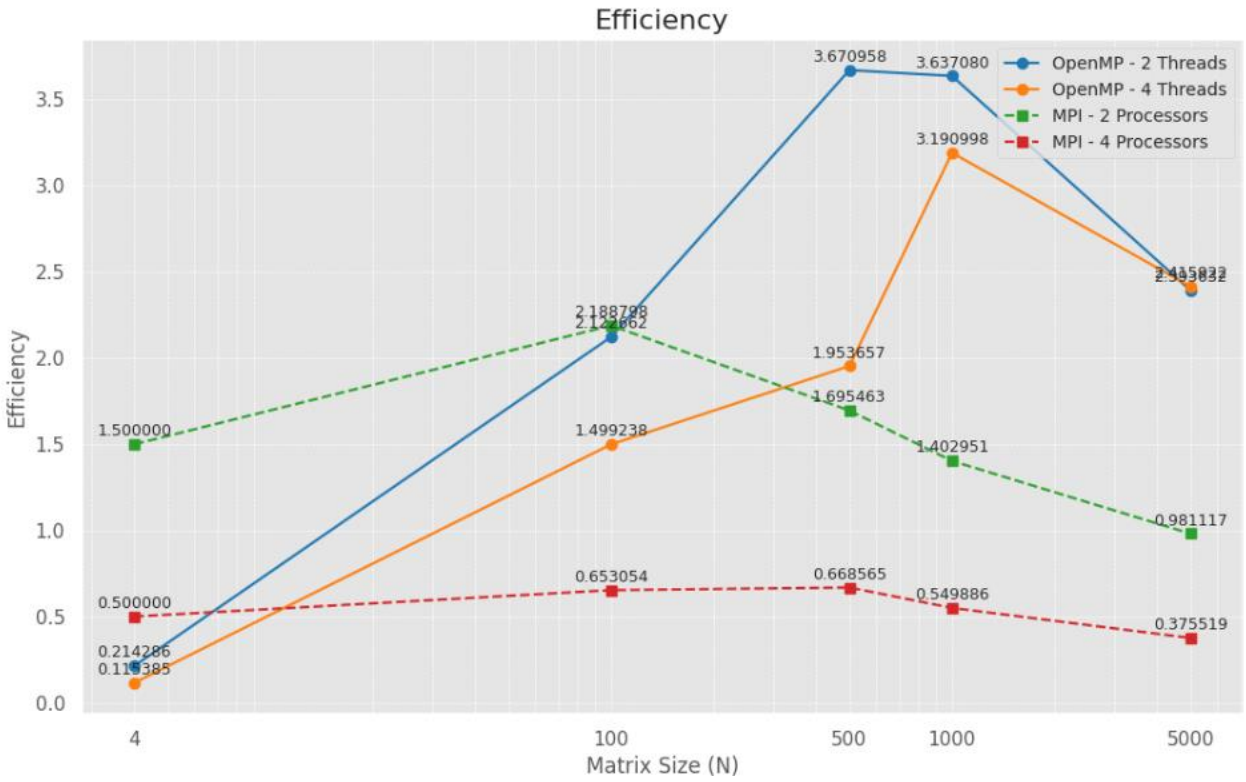
OpenMP Metrics

Matrix Size (N)	Threads	Execution Time (s)	Speedup	Efficiency
4	2	0.000014	0.428571	0.214286
4	4	0.000013	0.461538	0.115385
100	2	0.001390	4.245324	2.122662
100	4	0.000984	5.996951	1.499238
500	2	0.076045	7.341916	3.670958
500	4	0.071445	7.814627	1.953657
1000	2	0.612509	7.274161	3.637080
1000	4	0.349067	12.763994	3.190998
5000	2	160.058183	4.787264	2.393632
5000	4	79.290732	9.663686	2.415922

MPI Metrics

Matrix Size (N)	Processors	Execution Time (s)	Speedup	Efficiency
4	2	0.000002	3.000000	1.500000
4	4	0.000003	2.000000	0.500000
100	2	0.001348	4.377596	2.188798
100	4	0.002259	2.612218	0.653054
500	2	0.164650	3.390926	1.695463
500	4	0.208774	2.674260	0.668565
1000	2	1.587899	2.805902	1.402951
1000	4	2.025643	2.199543	0.549886
5000	2	390.494172	1.962234	0.981117
5000	4	510.121312	1.502076	0.375519





Conclusion

The project successfully implemented LU Decomposition using Doolittle method in C and effectively compared three different approaches. OpenMP showcased shared-memory parallelism and displayed the same speed as MPI, which demonstrated similar performance owing to distributed-memory parallelism. This comparison provides insights into selecting the appropriate parallelization technique based on the problem's nature and system resources.