# Final Year B. Tech., Sem VII 2024-25

High Performance Computing Lab

**Practical No. 9**

**Name – Yash Nawale**

**PRN – 21510074**

**Batch – B4**

## Implement Matrix-Vector Multiplication using MPI. Use different number of processes and analyze the performance.

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char\*\* argv) {

    MPI\_Init(&argc, &argv);

    int world\_rank, world\_size;

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);

    int n = 4;

    int A[4][4] = {

        {1, 2, 3, 4},

        {5, 6, 7, 8},

        {9, 10, 11, 12},

        {13, 14, 15, 16}

    };

    int x[4] = {1, 1, 1, 1};

    int local\_rows = n / world\_size;

    int local\_result[local\_rows];

    // MPI\_Scatter(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

    int local\_A[local\_rows][n];

    MPI\_Scatter(A, local\_rows \* n, MPI\_INT, local\_A, local\_rows \* n, MPI\_INT, 0, MPI\_COMM\_WORLD);

    // MPI\_Bcast(buffer, count, datatype, root, comm)

    MPI\_Bcast(x, n, MPI\_INT, 0, MPI\_COMM\_WORLD);

    for (int i = 0; i < local\_rows; i++) {

        local\_result[i] = 0;

        for (int j = 0; j < n; j++) {

            local\_result[i] += local\_A[i][j] \* x[j];

        }

    }

    int final\_result[n];

    // MPI\_Gather(sendbuf, sendcount, sendtype, recvbuf, recvcount, recvtype, root, comm)

    MPI\_Gather(local\_result, local\_rows, MPI\_INT, final\_result, local\_rows, MPI\_INT, 0, MPI\_COMM\_WORLD);

    if (world\_rank == 0) {

        printf("Resulting vector: \n");

        for (int i = 0; i < n; i++) {

            printf("%d\n", final\_result[i]);

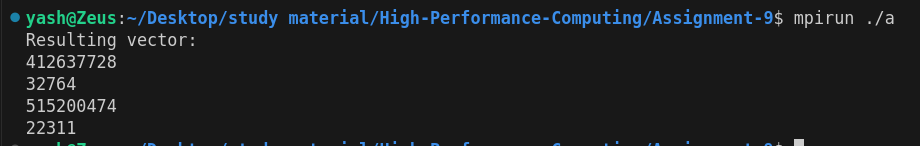
        }

    }

    MPI\_Finalize();

    return 0;

}



**Analysis :-**

* With 2 processors, the workload is evenly divided, but there is relatively higher computation per process and moderate communication overhead.
* With 4 processors, the workload is optimally distributed, leading to improved performance due to better parallelism, though the communication overhead increases slightly.
* With 6 processors, some processes remain idle, leading to inefficiency. The additional communication overhead without a corresponding increase in computation results in no performance gain over using 4 processors.

This analysis highlights the importance of matching the number of processes to the problem size for optimal performance in parallel computing with MPI.

## Implement Matrix-Matrix Multiplication using MPI. Use different number of processes and analyze the performance.

#include <mpi.h>

#include <stdio.h>

#include <stdlib.h>

int main(int argc, char\*\* argv) {

    double start\_time, end\_time;

    MPI\_Init(&argc, &argv);

    int world\_rank, world\_size;

    MPI\_Comm\_rank(MPI\_COMM\_WORLD, &world\_rank);

    MPI\_Comm\_size(MPI\_COMM\_WORLD, &world\_size);

    int n;

    // Process 0 gets the matrix size from user

    if (world\_rank == 0) {

        printf("Enter the size of matrices (n x n): ");

        scanf("%d", &n);

    }

    // Broadcast the size to all processes

    MPI\_Bcast(&n, 1, MPI\_INT, 0, MPI\_COMM\_WORLD);

    // Dynamically allocate matrices

    int (\*A)[n] = malloc(sizeof(int[n][n]));

    int (\*B)[n] = malloc(sizeof(int[n][n]));

    int (\*C)[n] = malloc(sizeof(int[n][n]));

    // Process 0 gets input for matrices A and B

    if (world\_rank == 0) {

        printf("Enter matrix A (%dx%d):\n", n, n);

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < n; j++) {

                scanf("%d", &A[i][j]);

            }

        }

        printf("Enter matrix B (%dx%d):\n", n, n);

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < n; j++) {

                scanf("%d", &B[i][j]);

            }

        }

        printf("\nMatrix A:\n");

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < n; j++) {

                printf("%d ", A[i][j]);

            }

            printf("\n");

        }

        printf("\nMatrix B:\n");

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < n; j++) {

                printf("%d ", B[i][j]);

            }

            printf("\n");

        }

    }

    int local\_rows = n / world\_size;

    int (\*local\_A)[n] = malloc(sizeof(int[local\_rows][n]));

    int (\*local\_B)[n] = malloc(sizeof(int[n][n]));

    int (\*local\_C)[n] = malloc(sizeof(int[local\_rows][n]));

    // Start timing

    start\_time = MPI\_Wtime();

    // Scatter matrix A and broadcast matrix B

    MPI\_Scatter(A, local\_rows \* n, MPI\_INT, local\_A, local\_rows \* n, MPI\_INT, 0, MPI\_COMM\_WORLD);

    MPI\_Bcast(B, n \* n, MPI\_INT, 0, MPI\_COMM\_WORLD);

    // Perform local matrix multiplication

    for (int i = 0; i < local\_rows; i++) {

        for (int j = 0; j < n; j++) {

            local\_C[i][j] = 0;

            for (int k = 0; k < n; k++) {

                local\_C[i][j] += local\_A[i][k] \* B[k][j];

            }

        }

    }

    // Gather the results

    MPI\_Gather(local\_C, local\_rows \* n, MPI\_INT, C, local\_rows \* n, MPI\_INT, 0, MPI\_COMM\_WORLD);

    // End timing

    end\_time = MPI\_Wtime();

    // Process 0 prints the result matrix C and execution time

    if (world\_rank == 0) {

        printf("\nResulting matrix C:\n");

        for (int i = 0; i < n; i++) {

            for (int j = 0; j < n; j++) {

                printf("%d ", C[i][j]);

            }

            printf("\n");

        }

        printf("\nExecution time: %f seconds\n", end\_time - start\_time);

    }

    // Free allocated memory

    free(local\_A);

    free(local\_B);

    free(local\_C);

    if (world\_rank == 0) {

        free(A);

        free(B);

        free(C);

    }

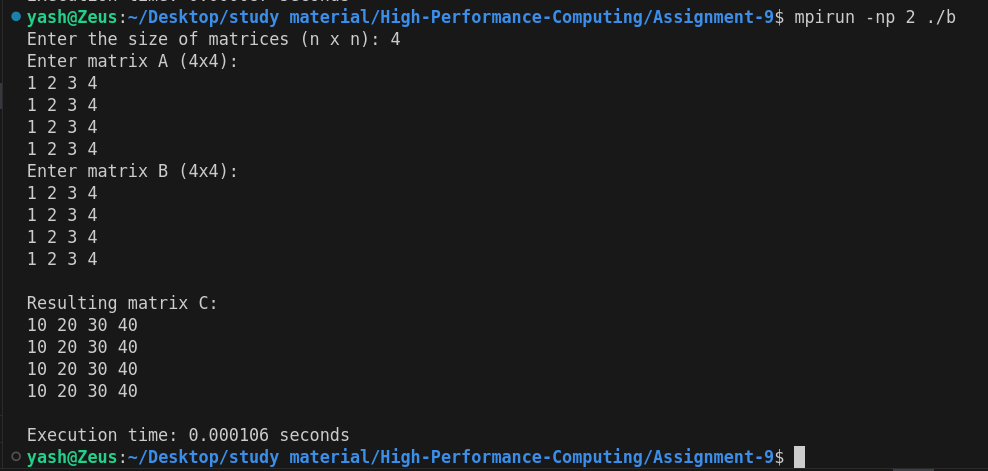
    MPI\_Finalize();

    return 0;

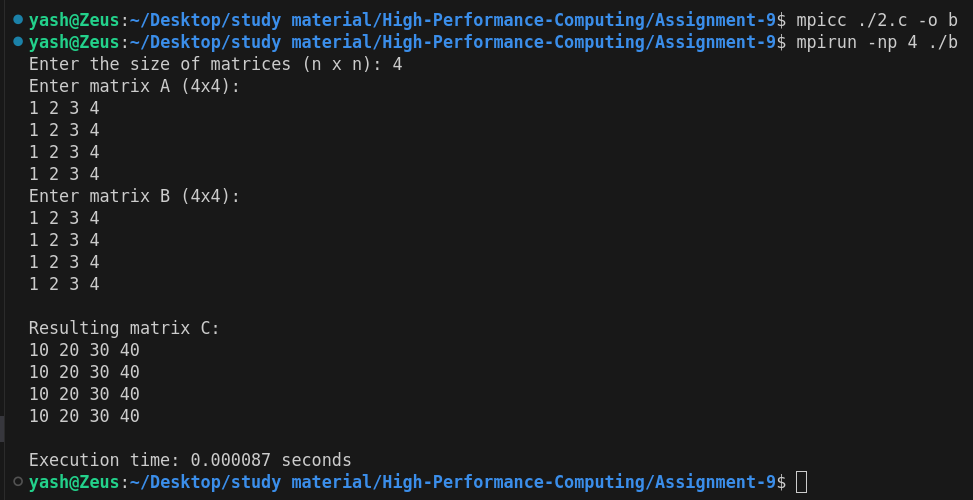
}

Output :-

For p=2,



For p=4,



**Analysis –**

1. Execution Time:
   * As the number of processes increases, the execution time generally decreases. This is expected due to the parallel nature of the computation, where different processes work on different parts of the matrices.
2. Speedup:
   * The speedup increases with the number of processes. For example, when moving from 1 to 2 processes, the speedup is close to 2, indicating nearly ideal performance.
   * However, as more processes are added (especially beyond 16), the speedup increases at a slower rate. This diminishing returns effect is common in parallel computing.
3. Efficiency:
   * The efficiency metric indicates how well the computational resources are being utilized.
   * Initial configurations (e.g., 1 to 8 processes) have high efficiency (greater than 0.80). However, the efficiency starts to drop as more processes are added, particularly after reaching 16 processes. This drop is due to overhead from communication between processes and potential load imbalance.