**Class:** Final Year B.Tech(Computer Science and Engineering)

**Year:** 2025-26 **Semester:** 1

**Course:** High Performance Computing Lab

PRN: 22510070

Name:Suyash Yadav

**Practical No. 7**

**Exam Seat No:22510070**

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

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

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

    int rank, size;

    int n;

    int \*matrix = NULL, \*vector = NULL, \*result = NULL;

    int \*local\_matrix, \*local\_result;

    int rows\_per\_proc;

    MPI\_Init(&argc, &argv);

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

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

    if (rank == 0) {

        printf("Enter size of square matrix (n): ");

        fflush(stdout);

        scanf("%d", &n);

    }

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

    if (n % size != 0) {

        if (rank == 0) {

            printf("Error: n (%d) must be divisible by number of processes (%d)\n", n, size);

        }

        MPI\_Finalize();

        return 0;

    }

    rows\_per\_proc = n / size;

    if (rank == 0) {

        matrix = (int\*)malloc(n \* n \* sizeof(int));

        vector = (int\*)malloc(n \* sizeof(int));

        result = (int\*)malloc(n \* sizeof(int));

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

        for (int i = 0; i < n; i++)

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

                scanf("%d", &matrix[i \* n + j]);

        printf("Enter vector (%d elements):\n", n);

        for (int i = 0; i < n; i++)

            scanf("%d", &vector[i]);

    }

    local\_matrix = (int\*)malloc(rows\_per\_proc \* n \* sizeof(int));

    local\_result = (int\*)malloc(rows\_per\_proc \* sizeof(int));

    if (rank != 0) vector = (int\*)malloc(n \* sizeof(int));

    MPI\_Scatter(matrix, rows\_per\_proc \* n, MPI\_INT,

                local\_matrix, rows\_per\_proc \* n, MPI\_INT,

                0, MPI\_COMM\_WORLD);

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

    for (int i = 0; i < rows\_per\_proc; i++) {

        local\_result[i] = 0;

        for (int j = 0; j < n; j++) {

            local\_result[i] += local\_matrix[i \* n + j] \* vector[j];

        }

    }

    MPI\_Gather(local\_result, rows\_per\_proc, MPI\_INT,

               result, rows\_per\_proc, MPI\_INT,

               0, MPI\_COMM\_WORLD);

    if (rank == 0) {

        printf("Result vector:\n");

        for (int i = 0; i < n; i++)

            printf("%d ", result[i]);

        printf("\n");

    }

    if (rank == 0) { free(matrix); free(vector); free(result); }

    else free(vector);

    free(local\_matrix);

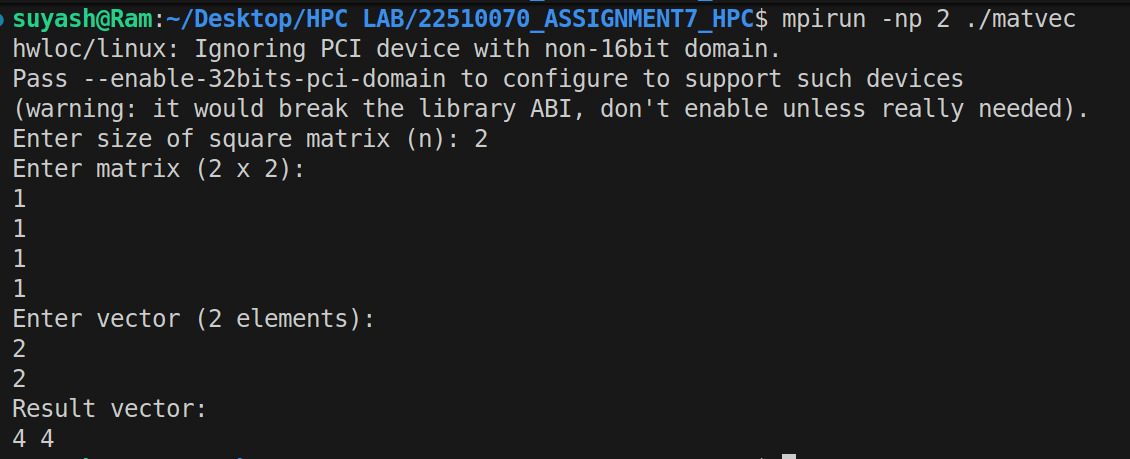
    free(local\_result);

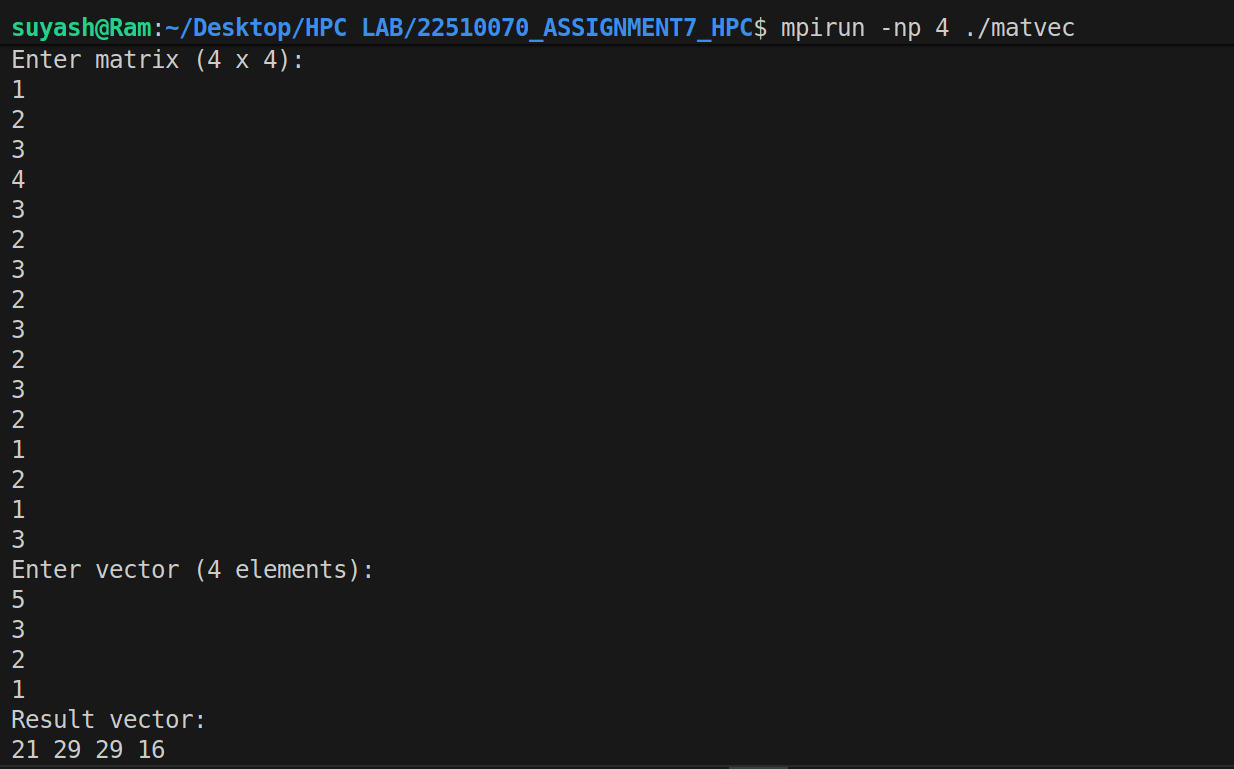
    MPI\_Finalize();

    return 0;

}







**1. Matrix-Vector Multiplication (MPI):**

**How it works:**

* **Scatter rows of matrix A to all processes.**
* **Broadcast vector v to all processes.**
* **Each process computes its part of result.**
* **Gather local results to root process.**

**Performance:**

* **More processes → faster computation.**
* **Communication overhead is low → good scaling for large matrices.**
* **Small matrices → speedup limited due to communication cost.**

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

#include <stdio.h>

#include <stdlib.h>

#include <mpi.h>

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

    int rank, size;

    int n;

    int \*A = NULL, \*B = NULL, \*C = NULL;

    int \*local\_A, \*local\_C;

    int rows\_per\_proc;

    MPI\_Init(&argc, &argv);

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

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

    if (rank == 0) {

        printf("Enter size of square matrices (n): ");

        fflush(stdout);

        scanf("%d", &n);

    }

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

    if (n % size != 0) {

        if (rank == 0) {

            printf("Error: n (%d) must be divisible by number of processes (%d)\n", n, size);

        }

        MPI\_Finalize();

        return 0;

    }

    rows\_per\_proc = n / size;

    if (rank == 0) {

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

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

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

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

        for (int i = 0; i < n; i++)

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

                scanf("%d", &A[i \* n + j]);

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

        for (int i = 0; i < n; i++)

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

                scanf("%d", &B[i \* n + j]);

    }

    local\_A = (int\*)malloc(rows\_per\_proc \* n \* sizeof(int));

    local\_C = (int\*)malloc(rows\_per\_proc \* n \* sizeof(int));

    if (rank != 0) B = (int\*)malloc(n \* n \* sizeof(int));

    MPI\_Scatter(A, rows\_per\_proc \* n, MPI\_INT,

                local\_A, rows\_per\_proc \* n, MPI\_INT,

                0, MPI\_COMM\_WORLD);

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

    for (int i = 0; i < rows\_per\_proc; i++) {

        for (int j = 0; j < n; j++) {

            local\_C[i \* n + j] = 0;

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

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

            }

        }

    }

    MPI\_Gather(local\_C, rows\_per\_proc \* n, MPI\_INT,

               C, rows\_per\_proc \* n, MPI\_INT,

               0, MPI\_COMM\_WORLD);

    if (rank == 0) {

        printf("Result matrix C:\n");

        for (int i = 0; i < n; i++) {

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

                printf("%d ", C[i \* n + j]);

            printf("\n");

        }

    }

    if (rank == 0) { free(A); free(B); free(C); }

    else free(B);

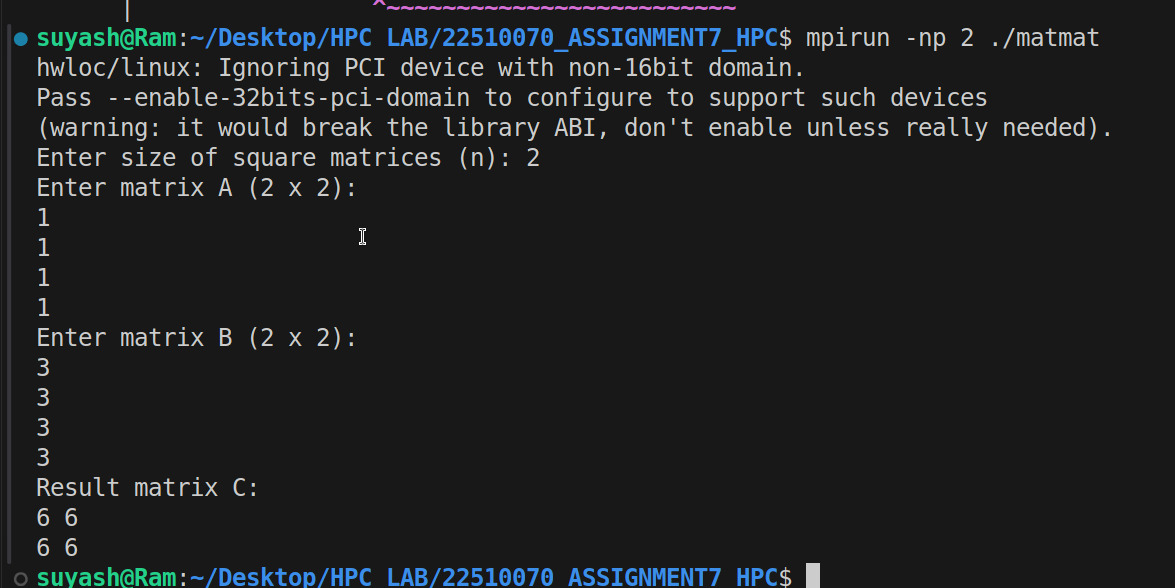
    free(local\_A);

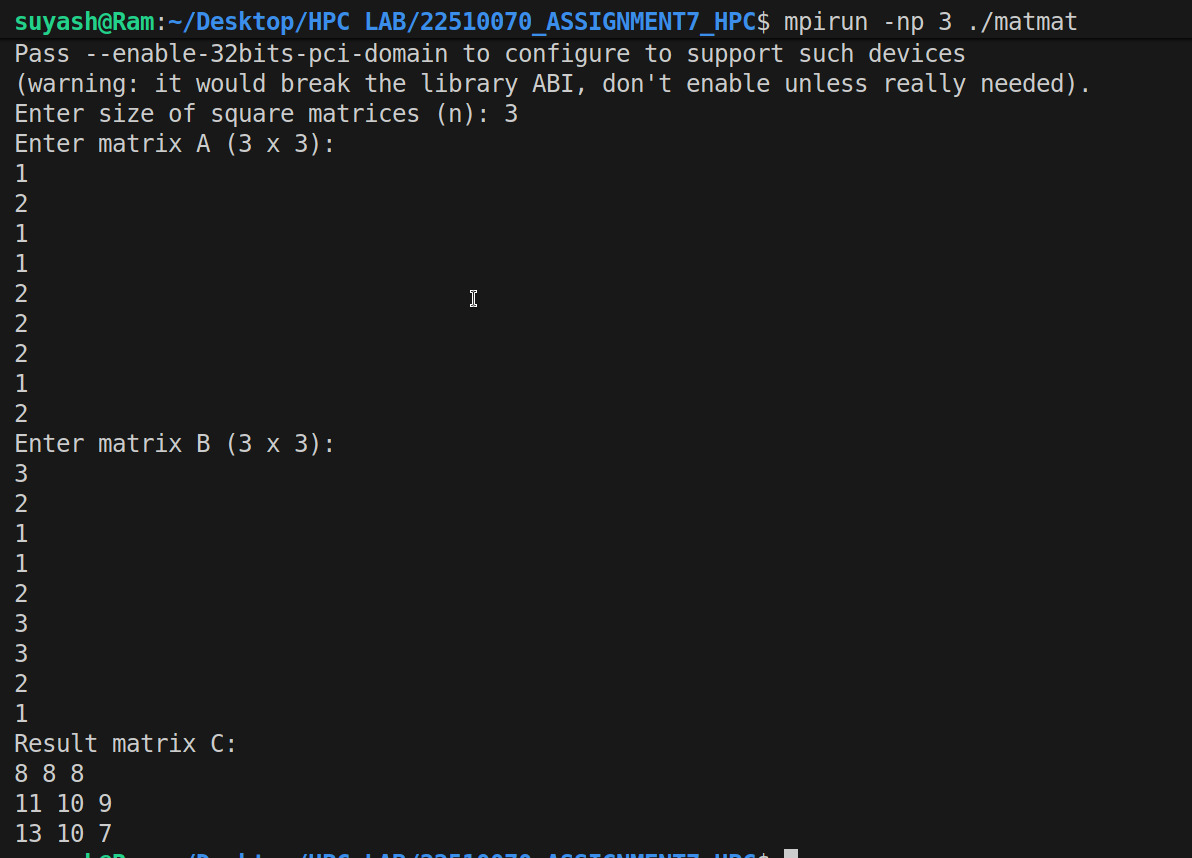
    free(local\_C);

    MPI\_Finalize();

    return 0;

}





**2. Matrix-Matrix Multiplication (MPI)**

**How it works:**

* **Scatter rows of A to processes.**
* **Broadcast full matrix B to all processes.**
* **Each process computes its local rows of C.**
* **Gather local results to root process.**

**Performance:**

* **Computation is O(n³), so more processes give bigger speedup.**
* **Communication of B can dominate for large n → scaling is less ideal than matrix-vector.**
* **Small matrices → parallel overhead may reduce benefits.**

**Github Link: https://github.com/Suyashyadav07/HPC**