**Class:** Final Year (Computer Science and Engineering)

**Year:** 2024-25 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 5**

**Exam Seat No: 22510064 – Parshwa Herwade**

**Github Link:** [**Sem-7-Assign/HPC lab at main · parshwa913/Sem-7-Assign · GitHub**](https://github.com/parshwa913/Sem-7-Assign/tree/main/HPC%20lab)

**Title of practical: Implementation of OpenMP programs.**

Implement following Programs using OpenMP with C:

1. Implementation of Matrix-Matrix Multiplication.
2. Implementation of Matrix-scalar Multiplication.
3. Implementation of Matrix-Vector Multiplication.
4. Implementation of Prefix sum.

**Problem Statement 1:**

#include <stdio.h>

#include <omp.h>

int main() {

    int r1, c1, r2, c2;

    printf("Enter rows and cols of matrix A: ");

    scanf("%d %d", &r1, &c1);

    printf("Enter rows and cols of matrix B: ");

    scanf("%d %d", &r2, &c2);

    if (c1 != r2) {

        printf("Matrix multiplication not possible.\n");

        return 0;

    }

    int A[r1][c1], B[r2][c2], C[r1][c2];

    printf("Enter elements of matrix A:\n");

    for (int i = 0; i < r1; i++)

        for (int j = 0; j < c1; j++)

            scanf("%d", &A[i][j]);

    printf("Enter elements of matrix B:\n");

    for (int i = 0; i < r2; i++)

        for (int j = 0; j < c2; j++)

            scanf("%d", &B[i][j]);

    #pragma omp parallel for

    for (int i = 0; i < r1; i++) {

        for (int j = 0; j < c2; j++) {

            C[i][j] = 0;

            for (int k = 0; k < c1; k++) {

                C[i][j] += A[i][k] \* B[k][j];

            }

        }

    }

    printf("Resultant Matrix:\n");

    for (int i = 0; i < r1; i++) {

        for (int j = 0; j < c2; j++)

            printf("%d ", C[i][j]);

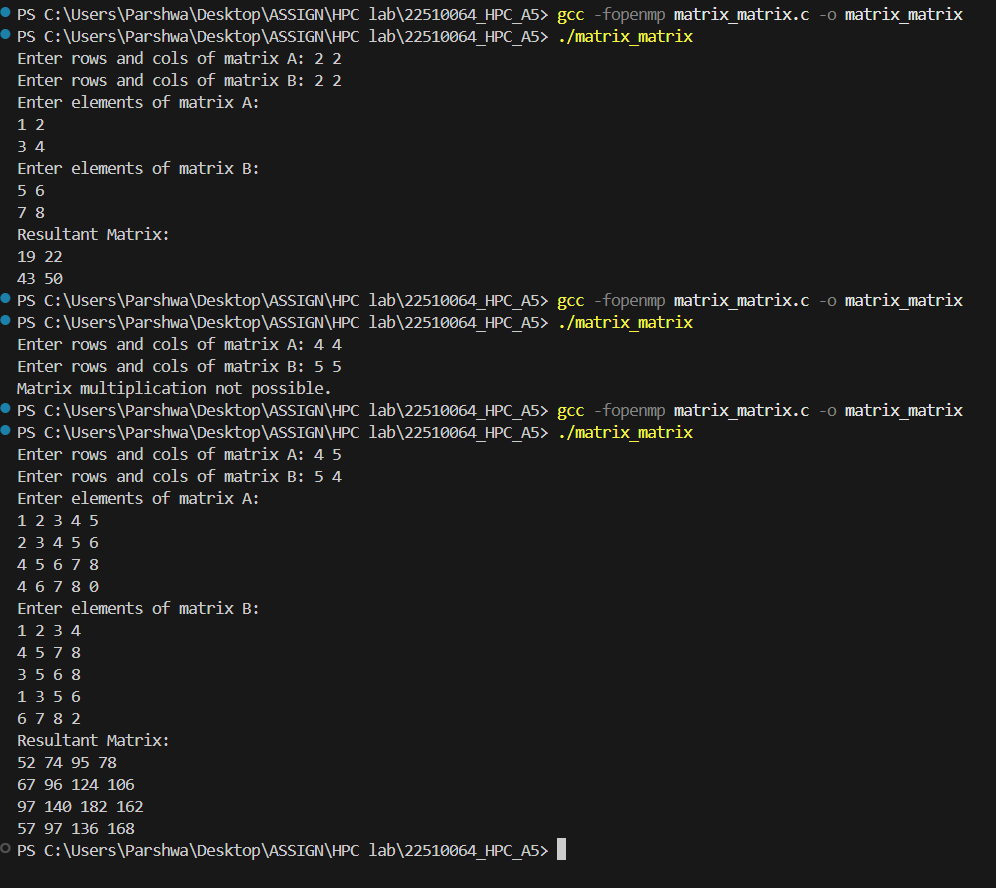
        printf("\n");

    }

    return 0;

}

**Screenshots:**

****

**Information:**

Information (Theory):

Matrix multiplication is a fundamental operation in numerical computing.

If A is of dimension r1×c1 and B is of dimension r2×c2, multiplication is possible only when c1 = r2.

Result matrix C will be of dimension r1×c2 where:

C[i][j] = Σ (A[i][k] × B[k][j]) for k from 0 to c1–1.

OpenMP’s #pragma omp parallel for can parallelize the outer loop, allowing different rows to be computed by different threads.

Analysis:

Matrix multiplication has a time complexity of O(n³) for naive implementation.

By parallelizing the computation, each thread handles a part of the iteration space, improving performance on multi-core systems. The speedup depends on the number of threads and cache efficiency.

Algorithm:

Input dimensions of matrices A and B.

Verify multiplication feasibility (c1 = r2).

Input matrix A and matrix B.

Initialize result matrix C to zero.

Use parallel loop to compute C[i][j] as sum of A[i][k] × B[k][j].

Display result.

**Problem Statement 2:**

#include <stdio.h>

#include <omp.h>

int main() {

    int r, c, scalar;

    printf("Enter rows and cols of matrix: ");

    scanf("%d %d", &r, &c);

    printf("Enter scalar value: ");

    scanf("%d", &scalar);

    int A[r][c];

    printf("Enter elements of matrix:\n");

    for (int i = 0; i < r; i++)

        for (int j = 0; j < c; j++)

            scanf("%d", &A[i][j]);

    #pragma omp parallel for

    for (int i = 0; i < r; i++)

        for (int j = 0; j < c; j++)

            A[i][j] \*= scalar;

    printf("Resultant Matrix:\n");

    for (int i = 0; i < r; i++) {

        for (int j = 0; j < c; j++)

            printf("%d ", A[i][j]);

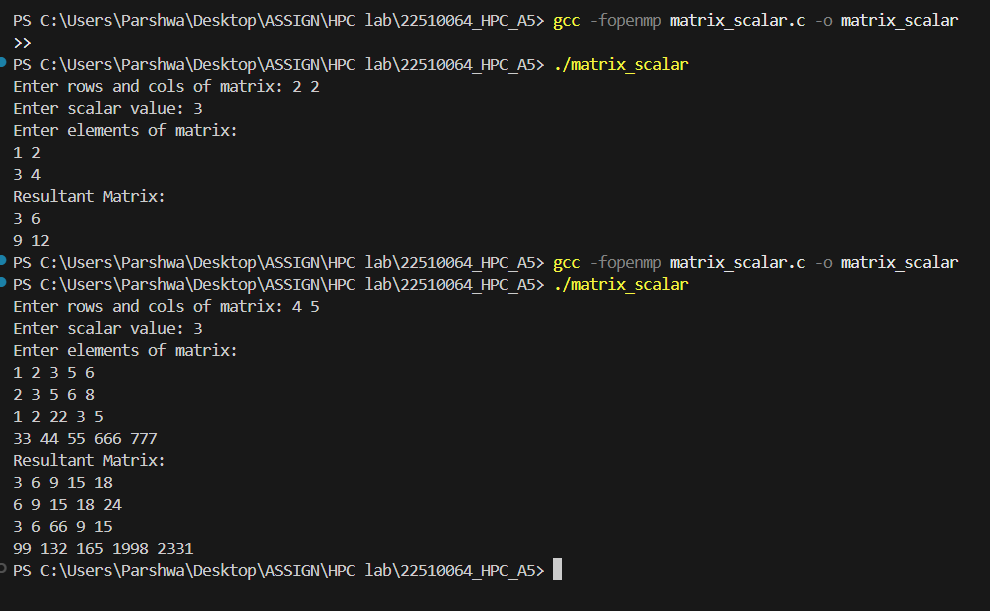
        printf("\n");

    }

    return 0;

}

**Screenshots:**

****

**Information:**

Information (Theory):  
Matrix–scalar multiplication multiplies each element of the matrix by the scalar value.  
OpenMP can parallelize the iteration over matrix elements so that multiple elements are updated simultaneously.

Analysis:  
Time complexity is O(n²) for an n×n matrix.  
With OpenMP, the work is divided among available threads, giving near-linear speedup for large matrices.

Algorithm:

1. Input dimensions of the matrix.
2. Input matrix elements.
3. Input scalar value.
4. Multiply each element by scalar using a parallel loop.
5. Display the resulting matrix.

**Problem Statement 3:**

#include <stdio.h>

#include <omp.h>

int main() {

    int r, c;

    printf("Enter rows and cols of matrix: ");

    scanf("%d %d", &r, &c);

    int A[r][c], V[c], R[r];

    printf("Enter elements of matrix:\n");

    for (int i = 0; i < r; i++)

        for (int j = 0; j < c; j++)

            scanf("%d", &A[i][j]);

    printf("Enter elements of vector:\n");

    for (int i = 0; i < c; i++)

        scanf("%d", &V[i]);

    #pragma omp parallel for

    for (int i = 0; i < r; i++) {

        R[i] = 0;

        for (int j = 0; j < c; j++)

            R[i] += A[i][j] \* V[j];

    }

    printf("Resultant Vector:\n");

    for (int i = 0; i < r; i++)

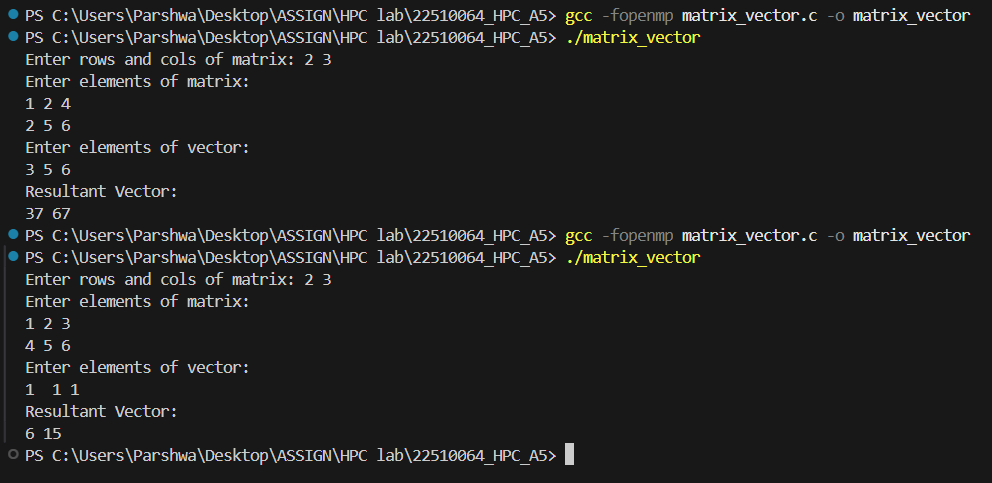
        printf("%d ", R[i]);

    printf("\n");

    return 0;

}

**Screenshots:**



**Information:**

Information (Theory):

Matrix–vector multiplication produces a vector where each element is the dot product of a row of the matrix with the input vector.

OpenMP parallelizes over rows, assigning each to a different thread.

Analysis:

The complexity is O(r×c).

Parallelization significantly improves performance for large matrices, as each row can be computed independently.

Algorithm:

Input dimensions of the matrix.

Input matrix and vector elements.

Multiply each row of the matrix with the vector using parallel loop.

Store and display the resulting vector.

**Problem Statement 4:**

#include <stdio.h>

#include <omp.h>

int main() {

    int n;

    printf("Enter number of elements: ");

    scanf("%d", &n);

    int arr[n], prefix[n];

    printf("Enter elements:\n");

    for (int i = 0; i < n; i++)

        scanf("%d", &arr[i]);

    prefix[0] = arr[0];

    for (int i = 1; i < n; i++) {

        prefix[i] = prefix[i - 1] + arr[i];

    }

    printf("Prefix Sum Array:\n");

    for (int i = 0; i < n; i++)

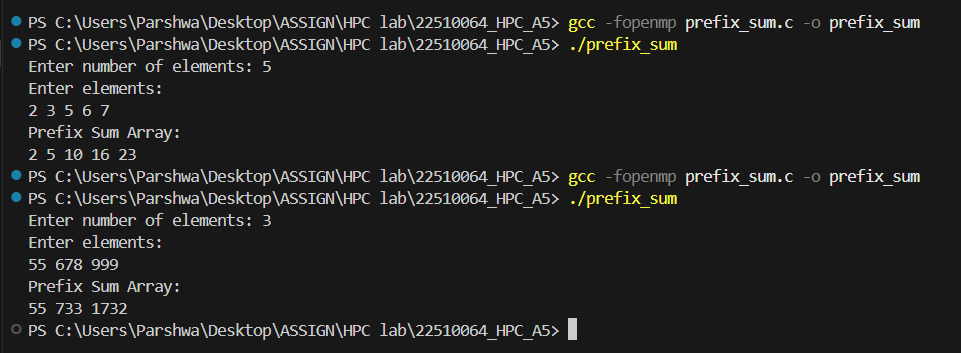
        printf("%d ", prefix[i]);

    printf("\n");

    return 0;

}

**Screenshots:**



**Information:**

Information (Theory):  
Prefix sum of an array is an array where each element is the sum of all previous elements including itself.  
Formula: prefix[i] = prefix[i–1] + arr[i].  
The sequential dependency makes naive parallelization tricky; advanced algorithms like the scan method can parallelize it, but here we demonstrate the basic computation.

Analysis:  
Sequential complexity is O(n).  
Using advanced parallel algorithms, computation can be reduced to O(log n) with enough threads.  
For small arrays, sequential is often faster due to overhead.

Algorithm:

1. Input the number of elements.
2. Input the array elements.
3. Compute prefix sums sequentially (or using parallel scan if implemented).
4. Display the prefix sum array.