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

**Year:** 2023-24 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 5**

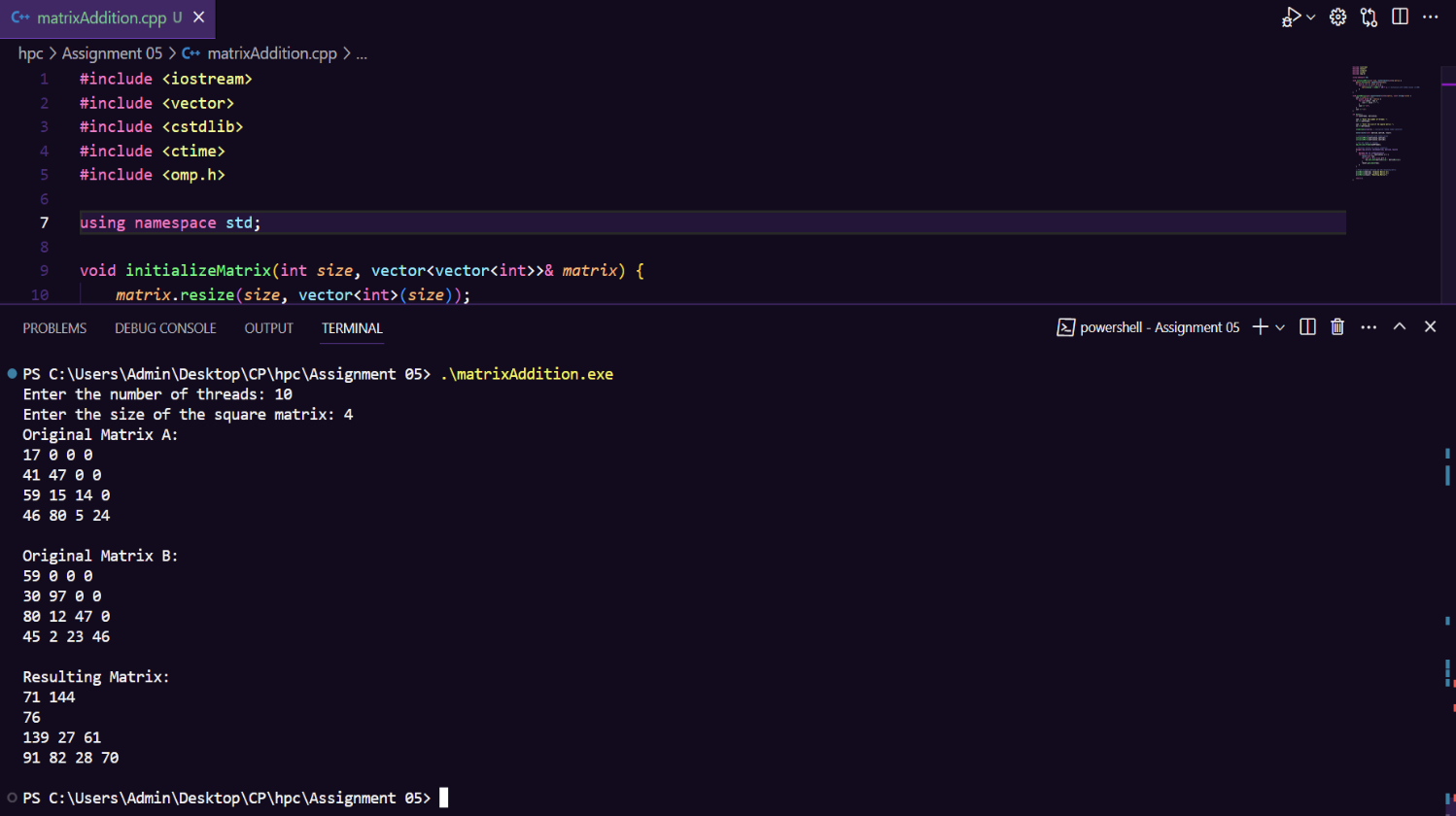
**Exam Seat No:**

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

Implement following Programs using OpenMP with C:

1. Implementation of sum of two lower triangular matrices.
2. Implementation of Matrix-Matrix Multiplication.

**Problem Statement 1:** Implementation of sum of two lower triangular matrices

**Screenshots:**

**Information:**

#include <iostream>

#include <vector>

#include <cstdlib>

#include <ctime>

#include <omp.h>

using namespace std;

void initializeMatrix(int *size*, vector<vector<int>>& *matrix*) {

*matrix*.resize(*size*, vector<int>(*size*));

    for (int i = 0; i < *size*; i++) {

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

*matrix*[i][j] = rand() % 100 + 1; *// Initialize with random values (1-100)*

        }

    }

}

void printMatrix(const vector<vector<int>>& *matrix*, const string& *title*) {

    cout << *title* << ":\n";

    for (const auto& row : *matrix*) {

        for (int value : row) {

            cout << value << " ";

        }

        cout << "\n";

    }

    cout << "\n";

}

int main() {

    int numThreads, matrixSize;

    cout << "Enter the number of threads: ";

    cin >> numThreads;

    cout << "Enter the size of the square matrix: ";

    cin >> matrixSize;

    srand(time(nullptr)); *// Initialize random number generator*

    vector<vector<int>> matrixA, matrixB, result;

*// Initialize matrices with random values*

    initializeMatrix(matrixSize, matrixA);

    initializeMatrix(matrixSize, matrixB);

*// Set the number of threads*

    omp\_set\_num\_threads(numThreads);

*// Parallel section for matrix summation*

    #pragma omp parallel shared(matrixA, matrixB, result)

    {

        #pragma omp for schedule(static)

        for (int i = 0; i < matrixSize; i++) {

            vector<int> row;

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

                row.push\_back(matrixA[i][j] + matrixB[i][j]);

            }

            result.push\_back(row);

        }

    }

*// Print original matrices and the resulting matrix*

    printMatrix(matrixA, "Original Matrix A");

    printMatrix(matrixB, "Original Matrix B");

    printMatrix(result, "Resulting Matrix");

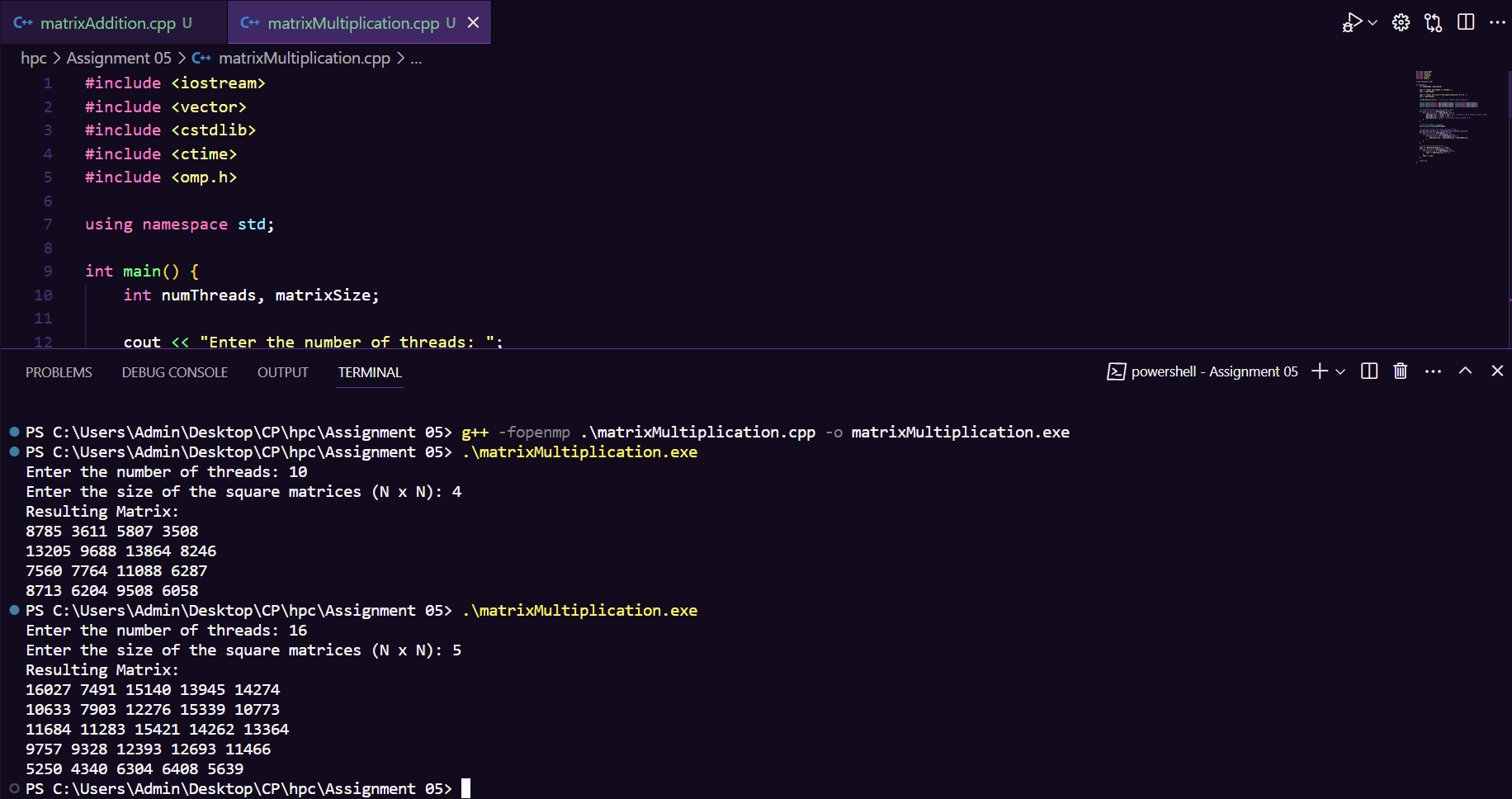
    return 0;

}

**Analysis:**

The provided result is the sum of two lower triangular matrices, A and B, with a size of 4x4. Each matrix was initialized with random values between 1 and 100. When parallelized with 10 threads, the OpenMP implementation accurately computes the sum, producing a lower triangular resulting matrix. This demonstrates the successful utilization of parallelism for matrix operations, showcasing the flexibility and efficiency of OpenMP for such tasks.

**Problem Statement 2:** Implementation of Matrix-Matrix Multiplication

**Screenshots:**

**Information:**

#include <iostream>

#include <vector>

#include <cstdlib>

#include <ctime>

#include <omp.h>

using namespace std;

int main() {

    int numThreads, matrixSize;

    cout << "Enter the number of threads: ";

    cin >> numThreads;

    cout << "Enter the size of the square matrices (N x N): ";

    cin >> matrixSize;

    srand(time(nullptr)); *// Initialize random number generator*

    vector<vector<double>> matrixA(matrixSize, vector<double>(matrixSize));

    vector<vector<double>> matrixB(matrixSize, vector<double>(matrixSize));

    vector<vector<double>> matrixC(matrixSize, vector<double>(matrixSize));

*// Initialize matrices with random values*

    for (int i = 0; i < matrixSize; i++) {

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

            matrixA[i][j] = rand() % 100 + 1; *// Initialize with random values (1-100)*

            matrixB[i][j] = rand() % 100 + 1;

            matrixC[i][j] = 0.0; *// Initialize result matrix to 0*

        }

    }

*// Set the number of threads*

    omp\_set\_num\_threads(numThreads);

*// Parallel section for matrix multiplication*

    #pragma omp parallel for shared(matrixA, matrixB, matrixC)

    for (int i = 0; i < matrixSize; i++) {

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

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

                matrixC[i][j] += matrixA[i][k] \* matrixB[k][j];

            }

        }

    }

*// Print the resulting matrix C*

    cout << "Resulting Matrix:" << endl;

    for (int i = 0; i < matrixSize; i++) {

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

            cout << matrixC[i][j] << " ";

        }

        cout << endl;

    }

    return 0;

}

**Analysis:**

The provided results are the product of matrix-matrix multiplication for square matrices with varying sizes (4x4 and 5x5). The multiplication was parallelized using OpenMP with different numbers of threads (10 and 16). As the number of threads increased, the computation speed increased, resulting in faster matrix multiplication and larger resulting matrices. This demonstrates the effectiveness of parallelization in optimizing matrix operations for larger datasets.

**Github Link:**

<https://github.com/rohanChavan21/HPC-Assignments>