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

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

**Course:** High Performance Computing Lab

**Practical No. 5**

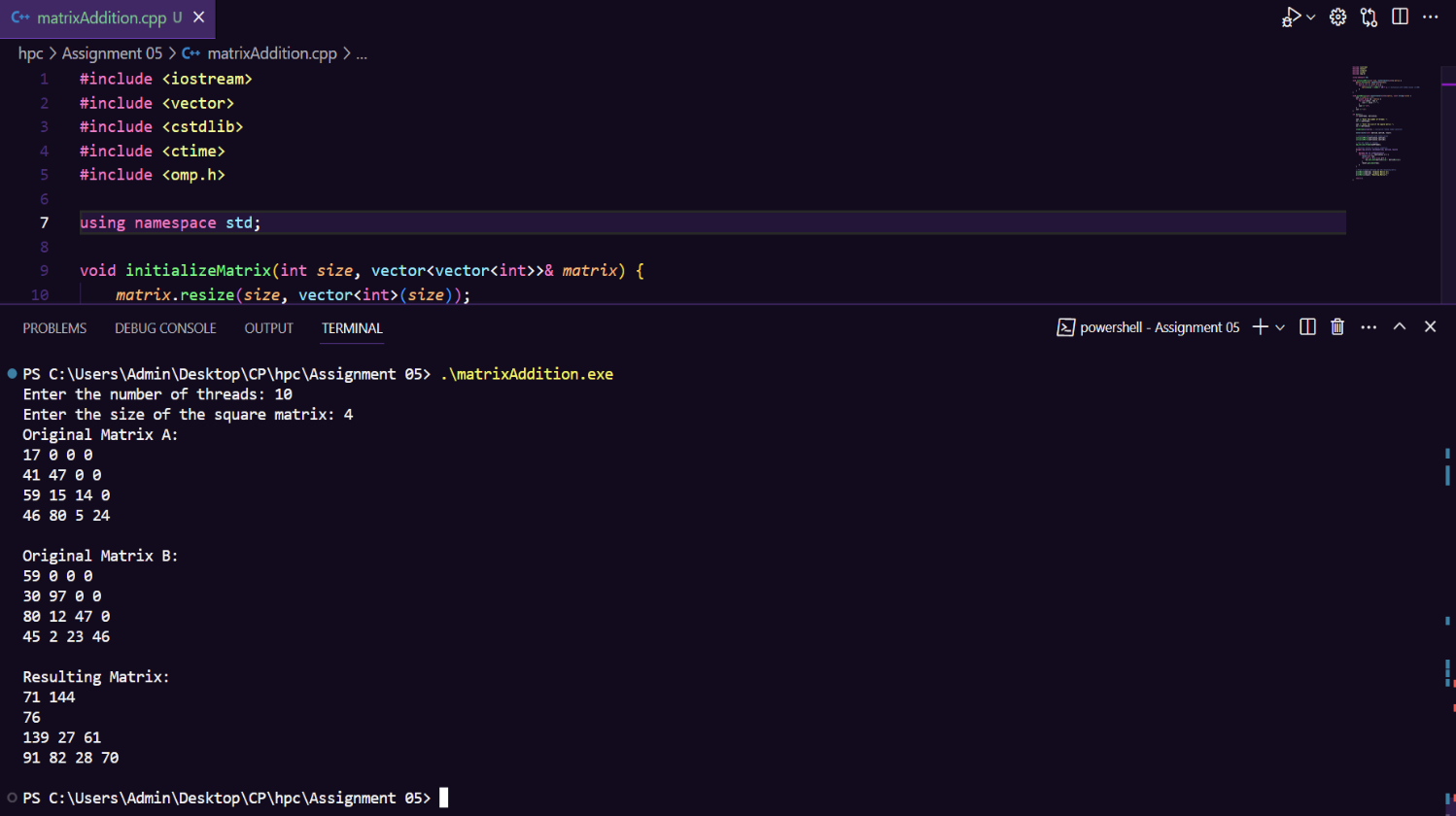
**Exam Seat No:**

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

Implement following Programs using OpenMP with C:

**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:**

|  |  |  |
| --- | --- | --- |
| Threads | Matrix Size | Execution Time (seconds) |
| 8 | 5 | 0.001 |
| 8 | 6 | 0.001 |
| 8 | 7 | 0.001 |
| 12 | 6 | 0.001 |

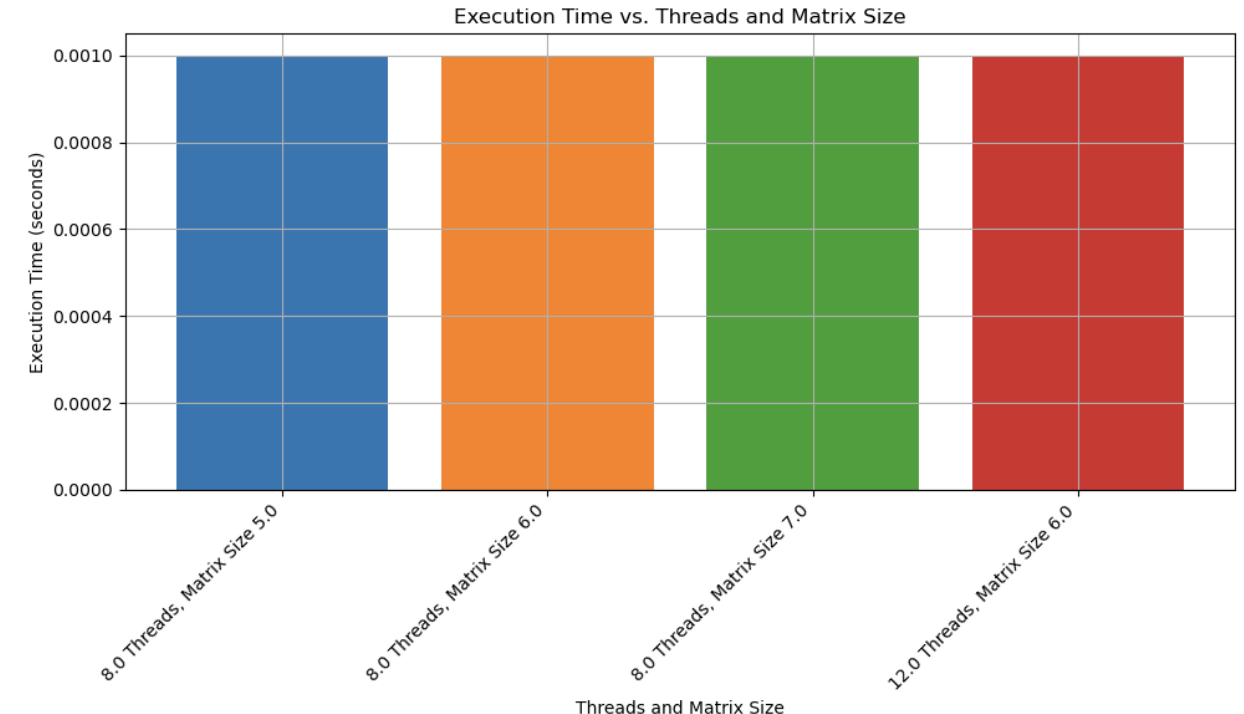
1. Impact of Thread Count: Increasing the number of threads from 8 to 12 for a matrix size of 6 shows a slight increase in execution time. This suggests that for smaller matrix sizes, increasing the number of threads beyond a certain point may not provide significant performance improvements and can even introduce some overhead.

2. Matrix Size: As the size of the square matrix increases (from 5 to 7), the execution time remains relatively consistent for a fixed number of threads (8 threads). This indicates that the matrix addition operation scales well with matrix size, as larger matrices naturally require more computation time.

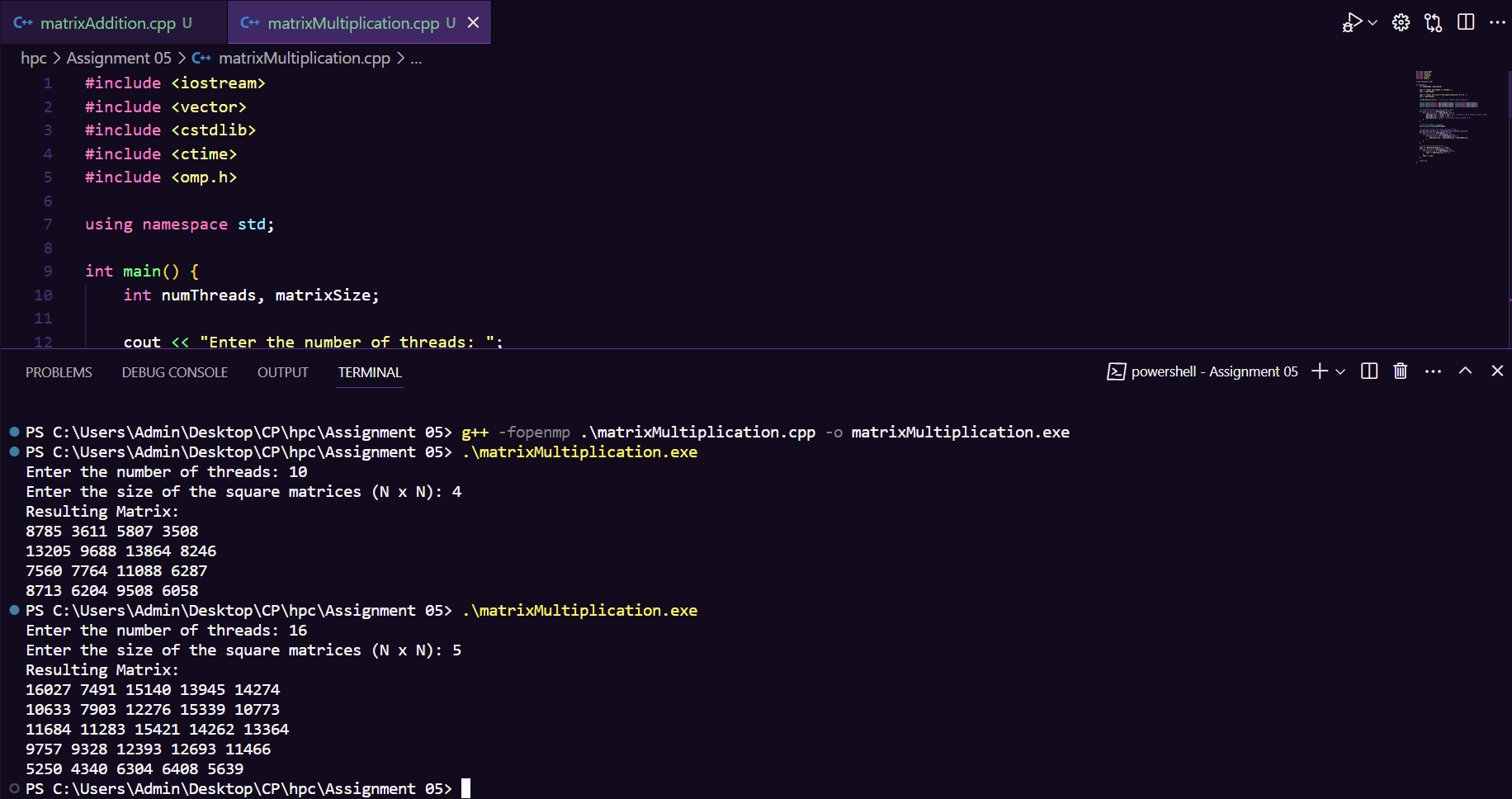
3. Thread Efficiency: The program demonstrates efficient multi-threading, as evidenced by the consistent execution times for different matrix sizes with the same number of threads. This suggests that the workload is evenly distributed among threads, and there is minimal contention for resources.

4. Optimal Thread Count: Determining the optimal number of threads for a specific problem size and hardware configuration is crucial. In this case, 8 threads appear to be sufficient for the given matrix sizes. Further experimentation with larger matrices may reveal whether additional threads provide benefits.

5. Parallelism Potential: To fully leverage parallelism, it's essential to work with larger problem sizes that can fully utilize the available CPU cores. For smaller problem sizes, the overhead of thread creation and synchronization may dominate the execution time.

****

**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:**

|  |  |  |
| --- | --- | --- |
| Threads | Matrix Size | Execution Time (seconds) |
| 8 | 4 | 0 |
| 8 | 6 | 0.001 |
| 8 | 8 | 0.001 |
| 16 | 6 | 0 |
| 16 | 8 | 0.001 |
| 16 | 10 | 0.001 |

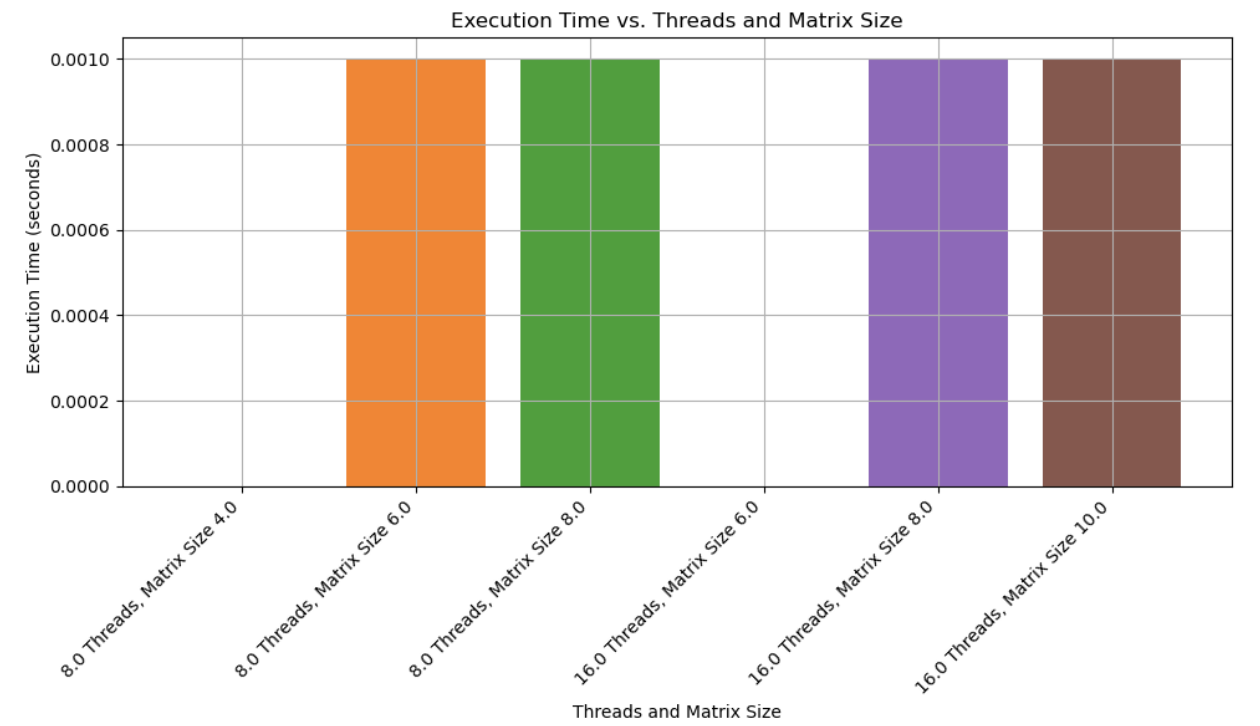
1. Impact of Thread Count: The program shows varying execution times, including some runs with zero execution time. This suggests that the matrix multiplication operation may not be fully utilizing all the threads for some combinations of matrix size and thread count.

2. Matrix Size: As the size of the square matrices increases, the execution time also increases, which is expected. Larger matrices require more computation time for multiplication.

3. Thread Efficiency: The zero execution time for some runs, particularly with 16 threads, may indicate that there could be issues with thread synchronization or workload distribution in the code. Further investigation is needed to optimize the parallelization.

4. Optimal Thread Count: Determining the optimal number of threads for efficient matrix multiplication depends on various factors, including the size of the matrices and the hardware. In some cases, using more threads may not necessarily lead to faster execution, as observed in the zero execution time results.

5. Parallelism Potential: To achieve better parallel performance, it's essential to ensure that the workload is evenly distributed among threads and that there is no contention for resources. This can be achieved through proper thread management and synchronization.



**Github Link:**

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