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

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

**Course:** High Performance Computing Lab

**Practical No. 5**

**Exam Seat No: 23520004**

**Name : Harshad Ajit Dange**

**Batch : B4**

**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:** Matrix-Matrix Multiplication

**Code :**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define N 500

int main() {

    static int A[N][N], B[N][N], C[N][N];

    int i, j, k;

    for (i = 0; i < N; i++)

    {

        for (j = 0; j < N; j++)

        {

            A[i][j] = rand() % 10;

            B[i][j] = rand() % 10;

            C[i][j] = 0;

        }

    }

    double st = omp\_get\_wtime();

    #pragma omp parallel for private(i, j, k) shared(A, B, C)

    for (i = 0; i < N; i++)

    {

        for (j = 0; j < N; j++)

        {

             for (k = 0; k < N; k++)

             {

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

             }

        }

    }

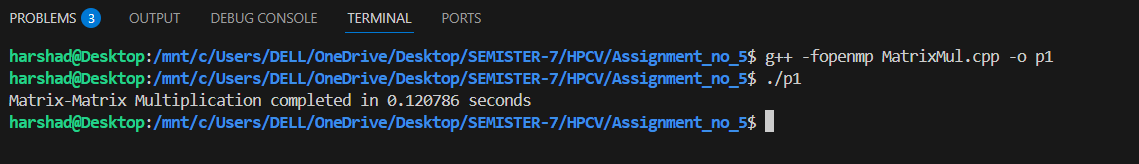
    double end = omp\_get\_wtime();

    printf("Matrix-Matrix Multiplication completed in %f seconds\n", end - st);

    return 0;

}

**Screenshot:**

****

**Problem Statement 2:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int main() {

    int r = 500;

    int c = 500;

    int A[r][c];

    int scalar = 5;

    int i, j;

    for (i = 0; i < r; i++)

        for (j = 0; j < c; j++)

            A[i][j] = rand() % 10;

    double st = omp\_get\_wtime();

    #pragma omp parallel for private(i, j) shared(A, scalar)

    for (i = 0; i < r; i++)

        for (j = 0; j < c; j++)

            A[i][j] \*= scalar;

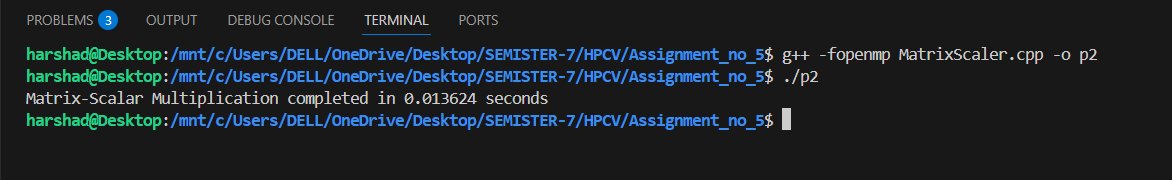
    double end = omp\_get\_wtime();

    printf("Matrix-Scalar Multiplication completed in %f seconds\n", end - st);

    return 0;

}

**Screenshot:**

****

**Problem Statement 3:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int main() {

    int N = 500;

    int A[N][N], x[N], y[N];

    int i, j;

    for (int i = 0; i < N; i++) {

        x[i] = rand() % 10;

        for (int j = 0; j < N; j++)

            A[i][j] = rand() % 10;

    }

    double st = omp\_get\_wtime();

    #pragma omp parallel for private(i, j) shared(A, x, y)

    for (int i = 0; i < N; i++) {

        y[i] = 0;

        for (int j = 0; j < N; j++)

            y[i] += A[i][j] \* x[j];

    }

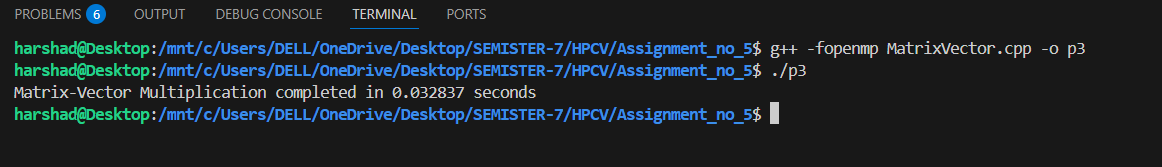
    double end = omp\_get\_wtime();

    printf("Matrix-Vector Multiplication completed in %f seconds\n", end - st);

    return 0;

}

**Screenshot:**

****

**Problem Statement 4:**

**Code:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int main() {

    int N =  1000000;

    int arr[N], pre[N];

    for (int i = 0; i < N; i++)

        arr[i] = 1;

    double st = omp\_get\_wtime();

    pre[0] = arr[0];

    #pragma omp parallel for

    for (int i = 1; i < N; i++)

        pre[i] = pre[i - 1] + arr[i];

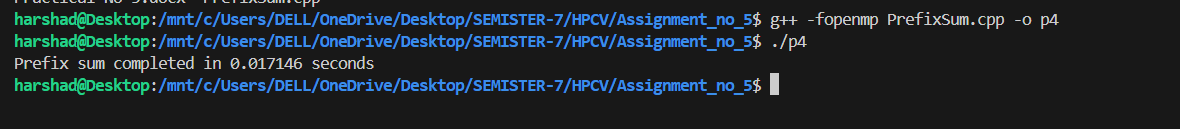
    double end = omp\_get\_wtime();

    printf("Prefix sum completed in %f seconds\n", end - st);

    return 0;

}

**Screenshot:**

****

## **Information**

The practical focuses on implementing parallel programming in C using OpenMP, an API that supports multi-platform shared-memory multiprocessing programming.  
The programs to be implemented are:

1. Matrix–Matrix Multiplication – computing the product of two matrices in parallel.
2. Matrix–Scalar Multiplication – multiplying each matrix element by a constant in parallel.
3. Matrix–Vector Multiplication – multiplying a matrix with a vector in parallel.
4. Prefix Sum – computing cumulative sums of an array using OpenMP.

Why OpenMP?

* OpenMP provides simple compiler directives (e.g., #pragma omp parallel for) to parallelize loops.
* It allows multi-threaded execution, improving performance on multi-core CPUs.
* Minimal code changes are needed to parallelize an existing sequential program.

Key Concepts Used:

* Parallel for loops (#pragma omp parallel for) to distribute iterations across threads.
* Private and shared variables to control data sharing among threads.
* Synchronization in cases where dependencies exist (prefix sum).
* Performance measurement using omp\_get\_wtime().

## Analysis

* Matrix–Matrix Multiplication:
  + High computational complexity: O(N³).
  + Perfectly parallelizable because each C[i][j] can be computed independently.
  + OpenMP divides row and column computations among threads, reducing runtime significantly.
* Matrix–Scalar Multiplication:
  + Complexity: O(N²) for an N×N matrix.
  + Each element is updated independently — no synchronization needed.
  + Parallelization yields almost linear speedup.
* Matrix–Vector Multiplication:
  + Complexity: O(N²) for N×N matrix.
  + Each output vector element is computed independently, allowing efficient parallelization.
  + Speedup is good but less than matrix-matrix multiplication due to fewer operations per thread.
* Prefix Sum:
  + Naive version is loop-carried dependent — current element depends on previous.
  + Full parallelization requires specialized algorithms like Blelloch Scan.
  + In OpenMP, partial parallelism is possible but requires careful synchronization.
  + Demonstrates limitations of parallelism for dependent tasks.

Performance Trend (Expected):

1. Matrix–Matrix multiplication → highest speedup due to heavy computation.
2. Matrix–Scalar multiplication → high speedup, but fewer operations.
3. Matrix–Vector multiplication → moderate speedup.
4. Prefix sum → minimal speedup (or even slowdown) unless using advanced parallel algorithms.

**Github Link:**