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

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

**Course: High Performance Computing Lab**

**Practical No. 2**

**PRN : 23520004**

**Name : Harshad Ajit Dange**

**Batch : B4**

**Title of practical: Study and implementation of basic OpenMP clauses**

Implement following Programs using OpenMP with C:

1. Vector Scalar Addition
2. Calculation of value of Pi

Analyse the performance of your programs for different number of threads and Data size.

**Problem Statement 1: Vector Scalar Addition**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define SIZE 10000000

int main() {

    int \*a = (int\*) malloc(SIZE \* sizeof(int));

    int \*b = (int\*) malloc(SIZE \* sizeof(int));

    int scalar = 10;

    for (int i = 0; i < SIZE; i++)

        a[i] = i;

    for (int threads = 1; threads <= 12; threads++) {

        omp\_set\_num\_threads(threads);

        double start = omp\_get\_wtime();

        #pragma omp parallel for

        for (int i = 0; i < SIZE; i++)

            b[i] = a[i] + scalar;

        double end = omp\_get\_wtime();

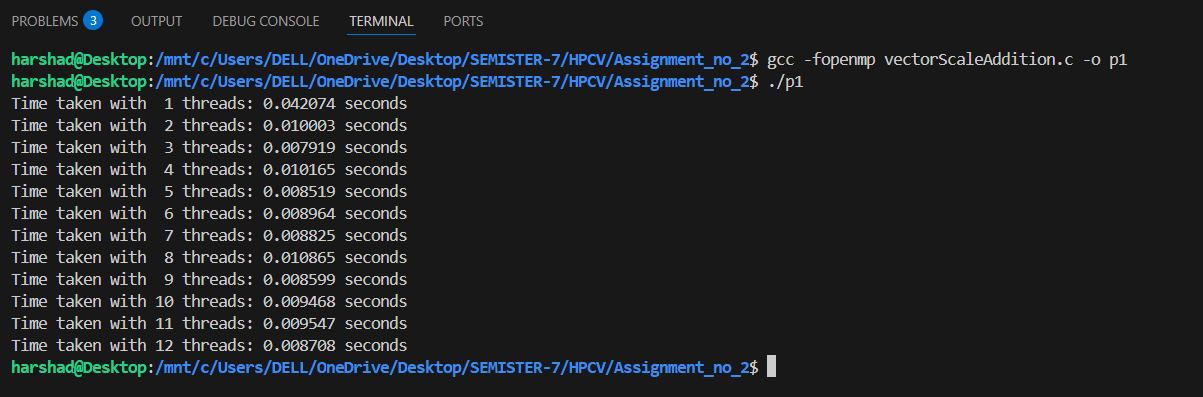
        printf("Time taken with %2d threads: %f seconds\n", threads, end - start);

    }

    return 0;

}

**Screenshots:**

****

**Information:**

* This program performs element-wise addition of a scalar value to a vector of size 10,000,000.
* It uses dynamic allocation to handle large arrays.
* ses the #pragma omp parallel for directive to parallelize the for-loop.
* Performance is measured for different thread counts ranging from 1 to 12.
* OpenMP is used to control the number of threads via omp\_set\_num\_threads.

**Analysis:**

* The computational task is *embarrassingly parallel*, making it highly scalable.
* As the number of threads increases, execution time generally decreases.
* Speedup depends on CPU core availability — after the number of physical/logical cores is exceeded, benefits plateau or degrade.
* The program efficiently demonstrates the OpenMP parallel for clause and its ability to split workload across threads.

**Problem Statement 2:**

#include <stdio.h>

#include <omp.h>

int main() {

    long long num\_steps = 100000000;

    double step = 1.0 / (double)num\_steps;

    for (int threads = 1; threads <= 12; threads++) {

        double pi = 0.0;

        omp\_set\_num\_threads(threads);

        double start = omp\_get\_wtime();

        #pragma omp parallel

        {

            double x, sum = 0.0;

            int id = omp\_get\_thread\_num();

            int nthreads = omp\_get\_num\_threads();

            for (long long i = id; i < num\_steps; i += nthreads) {

                x = (i + 0.5) \* step;

                sum += 4.0 / (1.0 + x \* x);

            }

            pi += sum \* step;

        }

        double end = omp\_get\_wtime();

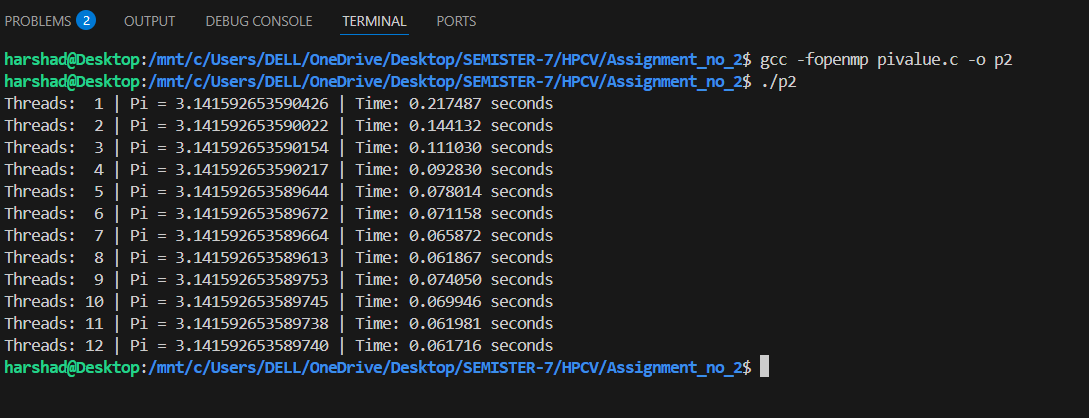
        printf("Threads: %2d | Pi = %.15f | Time: %f seconds\n", threads, pi, end - start);

    }

    return 0;

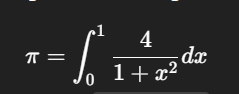
}

**Screenshots:**

****

**Information:**

* Approximates the value of π using numerical integration:



* Uses the rectangle (midpoint) rule with 100,000,000 steps.
* Each thread computes a partial sum of the integral.
* omp\_get\_thread\_num and omp\_get\_num\_threads used for manual work distribution.
* The final value of π is calculated by summing contributions from all threads.

**Analysis:**

* The workload is again highly parallelizable, but manual handling of sum accumulation introduces risks:
* In the current code, each thread updates pi independently, which could lead to incorrect final results due to race conditions.
* Time decreases with increased thread count, but:
* Adding too many threads may result in overhead.
* Ideal performance gain is subject to Amdahl’s Law.

**Github Link:**