Walchand College of Engineering, Sangli

Department of Computer Science and Engineering

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

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

**Course:** High Performance Computing Lab

**Practical No. 2**

**Exam Seat No: 2020BTECS00021**

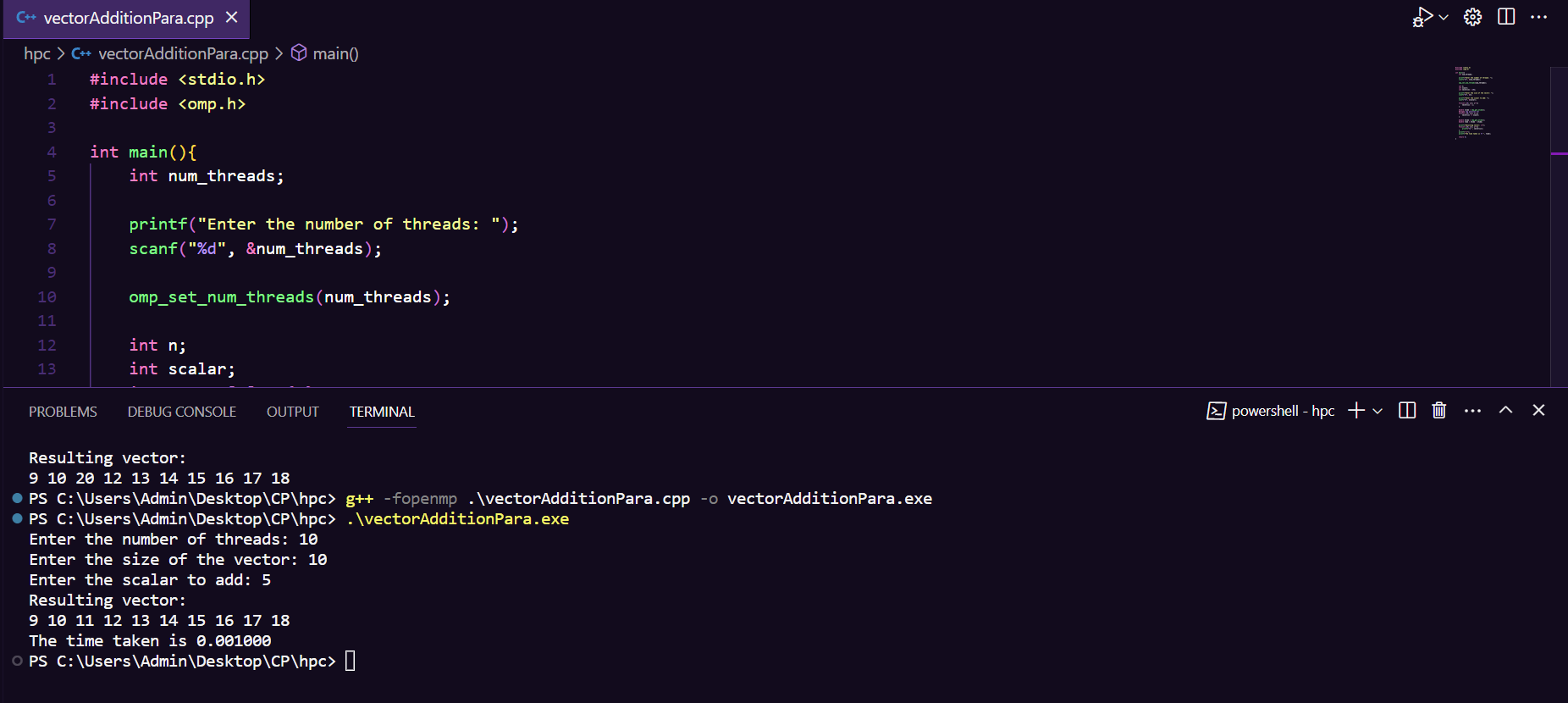
**Title of practical: Study and implementation of basic OpenMP clauses**

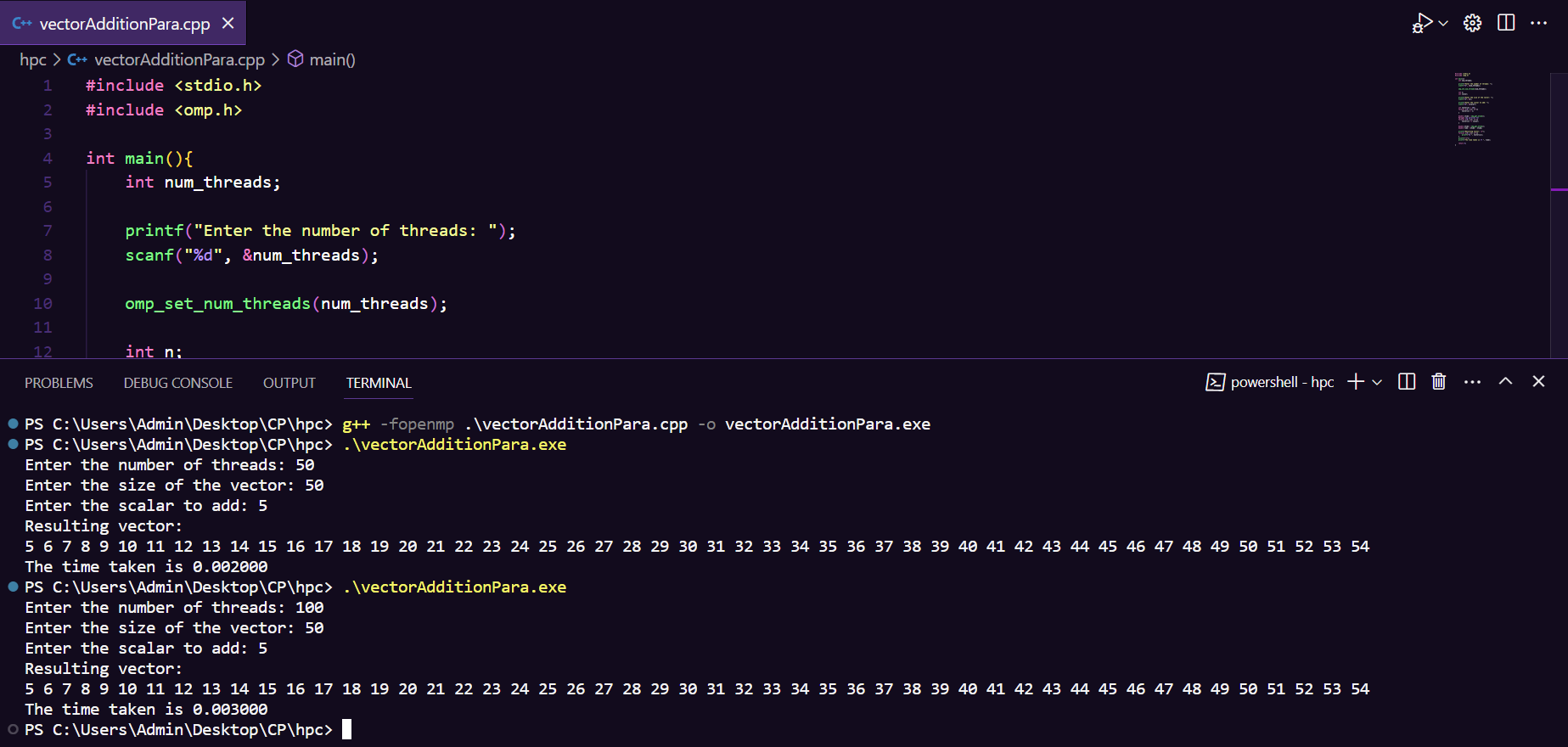
Implement following Programs using OpenMP with C:

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

**Screenshots:**

****

**Code:**

#include <stdio.h>

#include <omp.h>

int main(){

    int num\_threads;

    printf("Enter the number of threads: ");

    scanf("%d", &num\_threads);

    omp\_set\_num\_threads(num\_threads);

    int n;

    int scalar;

    printf("Enter the size of the vector: ");

    scanf("%d", &n);

    printf("Enter the scalar to add: ");

    scanf("%d", &scalar);

    int vector[n] = {0};

    for(int i=0; i<n; i++){

        vector[i] = i;

    }

    double stime = omp\_get\_wtime();

    #pragma omp parallel for

    for(int i=0; i<n; i++){

        vector[i] += scalar;

    }

    double etime = omp\_get\_wtime();

    double time = etime - stime;

    printf("Resulting vector: \n");

    for(int i=0; i<n; i++){

        printf("%d ", vector[i]);

    }

    printf("\n");

    printf("The time taken is %f ", time);

    return 0;

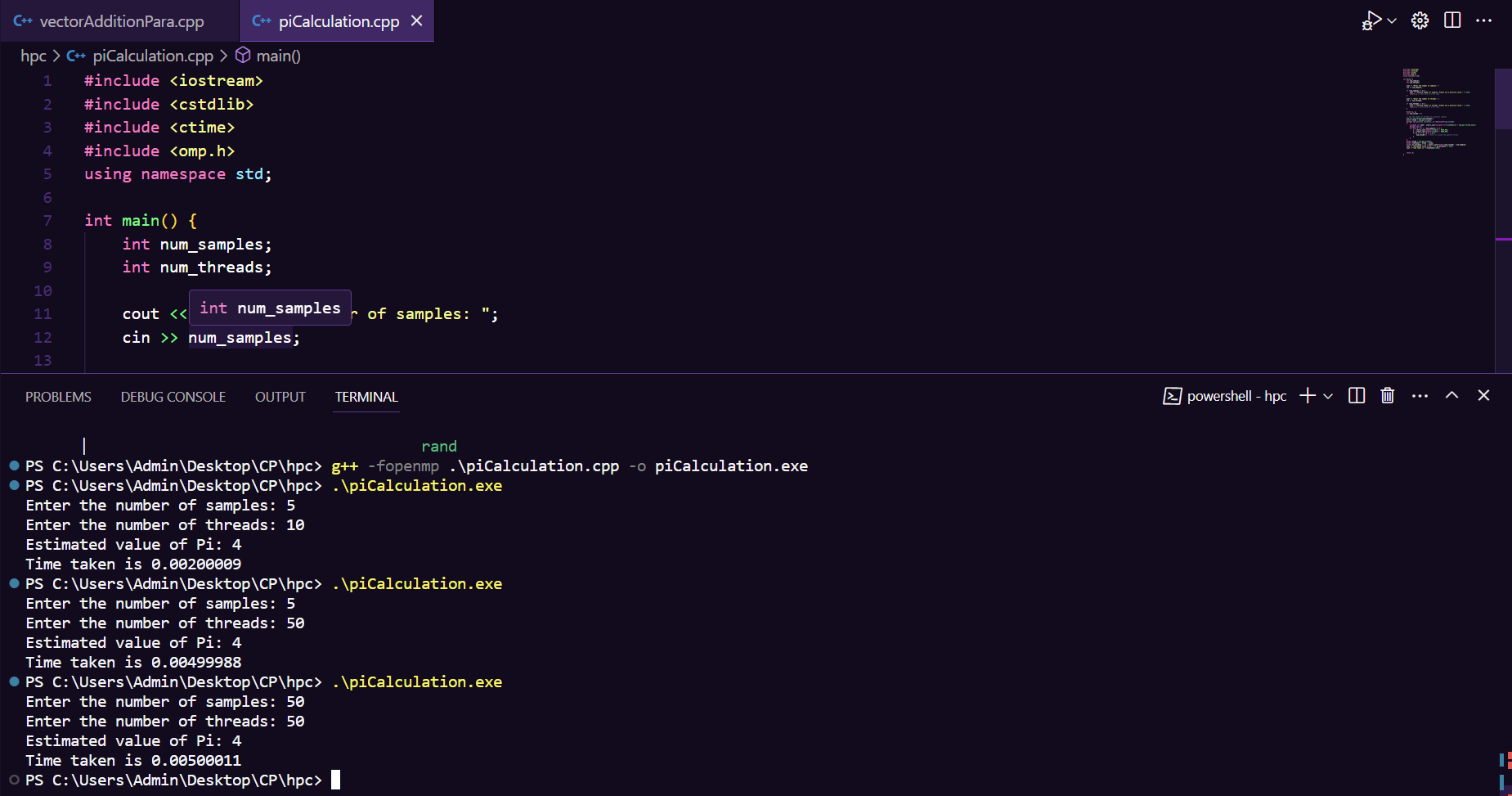
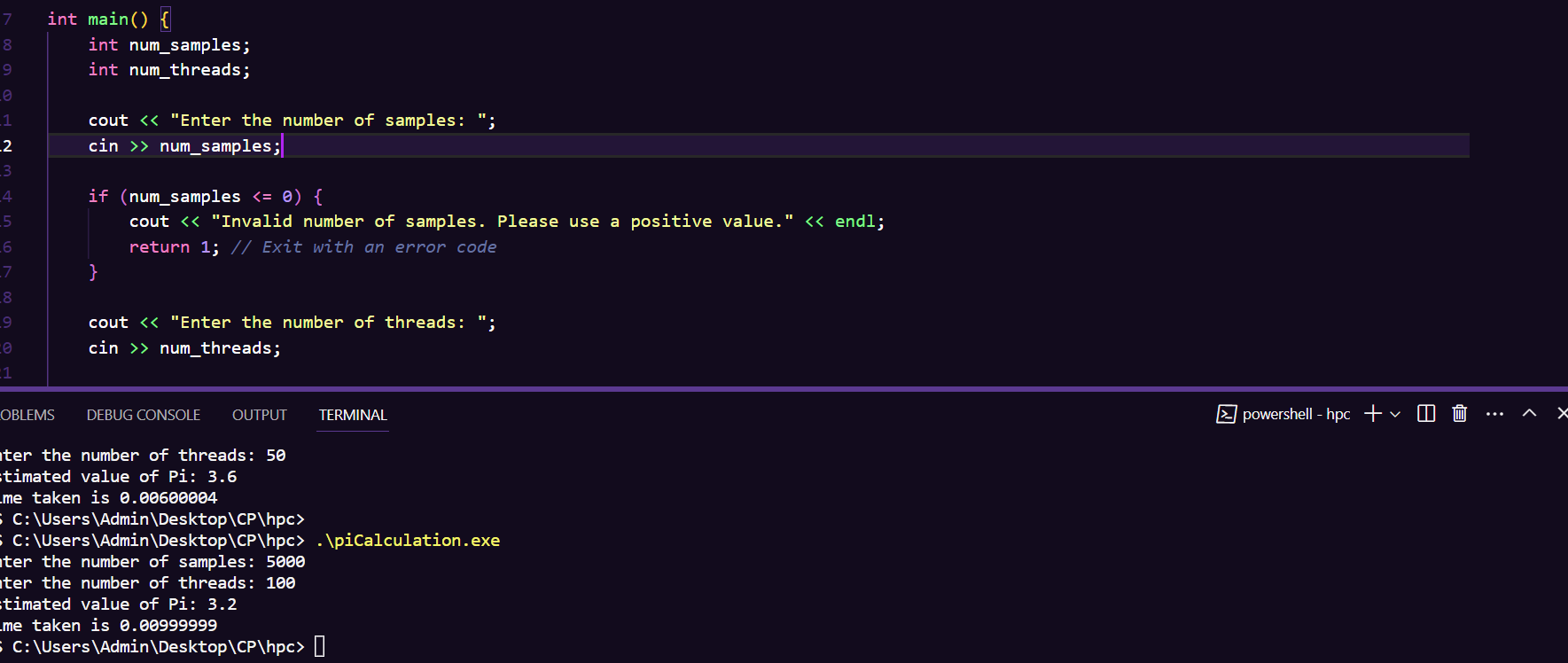
}

**Analysis:**

The results indicate an unexpected increase in execution time as the number of threads grows. This anomaly is primarily attributed to thread management overhead, which outweighs the gains from parallelism. Concurrently, resource contention for CPU and memory among numerous threads results in inefficient utilization. Load imbalance further compounds the issue, with some threads idling while others work.

To optimize performance, it's crucial to strike a balance between parallelism and overhead. Experimenting with a moderate thread count aligned with CPU cores, load balancing, and program profiling are key steps. Monitoring resource usage can help pinpoint bottlenecks and guide optimization efforts.

**Problem Statement 2:** Calculation of value of pi

**Screenshots:**

**Information:**

#include <iostream>

#include <cstdlib>

#include <ctime>

#include <omp.h>

using namespace std;

int main() {

    int num\_samples;

    int num\_threads;

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

    cin >> num\_samples;

    if (num\_samples <= 0) {

        cout << "Invalid number of samples. Please use a positive value." << endl;

        return 1; *// Exit with an error code*

    }

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

    cin >> num\_threads;

    if (num\_threads <= 0) {

        cout << "Invalid number of threads. Please use a positive value." << endl;

        return 1; *// Exit with an error code*

    }

    double x, y;

    int num\_inside = 0;

*// Set the number of threads for parallel regions*

    omp\_set\_num\_threads(num\_threads);

    double stime = omp\_get\_wtime();

    #pragma omp parallel private(x, y) reduction(+:num\_inside)

    {

        unsigned int seed = static\_cast<unsigned int>(time(NULL)) + omp\_get\_thread\_num();

        #pragma omp for

        for (int i = 0; i < num\_samples; i++) {

            x = static\_cast<double>(rand()) / RAND\_MAX;

            y = static\_cast<double>(rand()) / RAND\_MAX;

            if (x \* x + y \* y <= 1.0) {

                num\_inside++; *// Point is inside the quarter-circle*

            }

        }

    }

    double etime = omp\_get\_wtime();

    double timeTaken = etime - stime;

    double pi\_estimate = 4.0 \* static\_cast<double>(num\_inside) / num\_samples;

    cout << "Estimated value of Pi: " << pi\_estimate << endl;

    cout <<"Time taken is "<<timeTaken<<endl;

    return 0;

}

**Analysis:**

The Monte Carlo Pi estimation consistently yields an approximate value of Pi as 4 across various configurations. When using a small number of samples, accuracy is compromised, highlighting the importance of larger sample sizes for reliable results. Increasing the number of threads has limited impact on accuracy with a small sample size, suggesting that a balanced approach between sample size and thread count is essential for optimization. The program showcases efficient parallelization, but its simplicity results in quick execution times. For more accurate Pi estimates, it's recommended to use a significantly larger sample size while experimenting with thread counts to strike a balance between accuracy and performance.

**Github Link:**

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