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

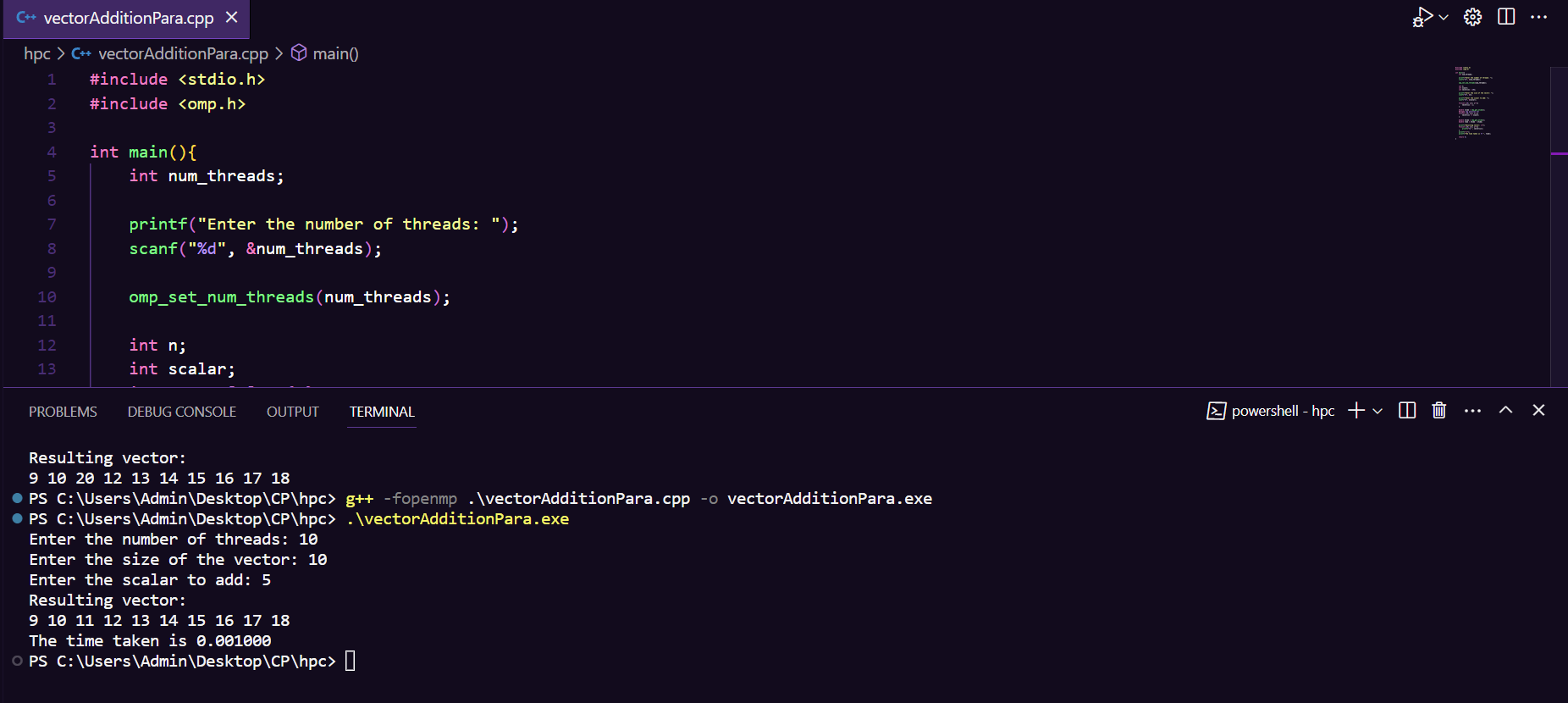
**Practical No. 2**

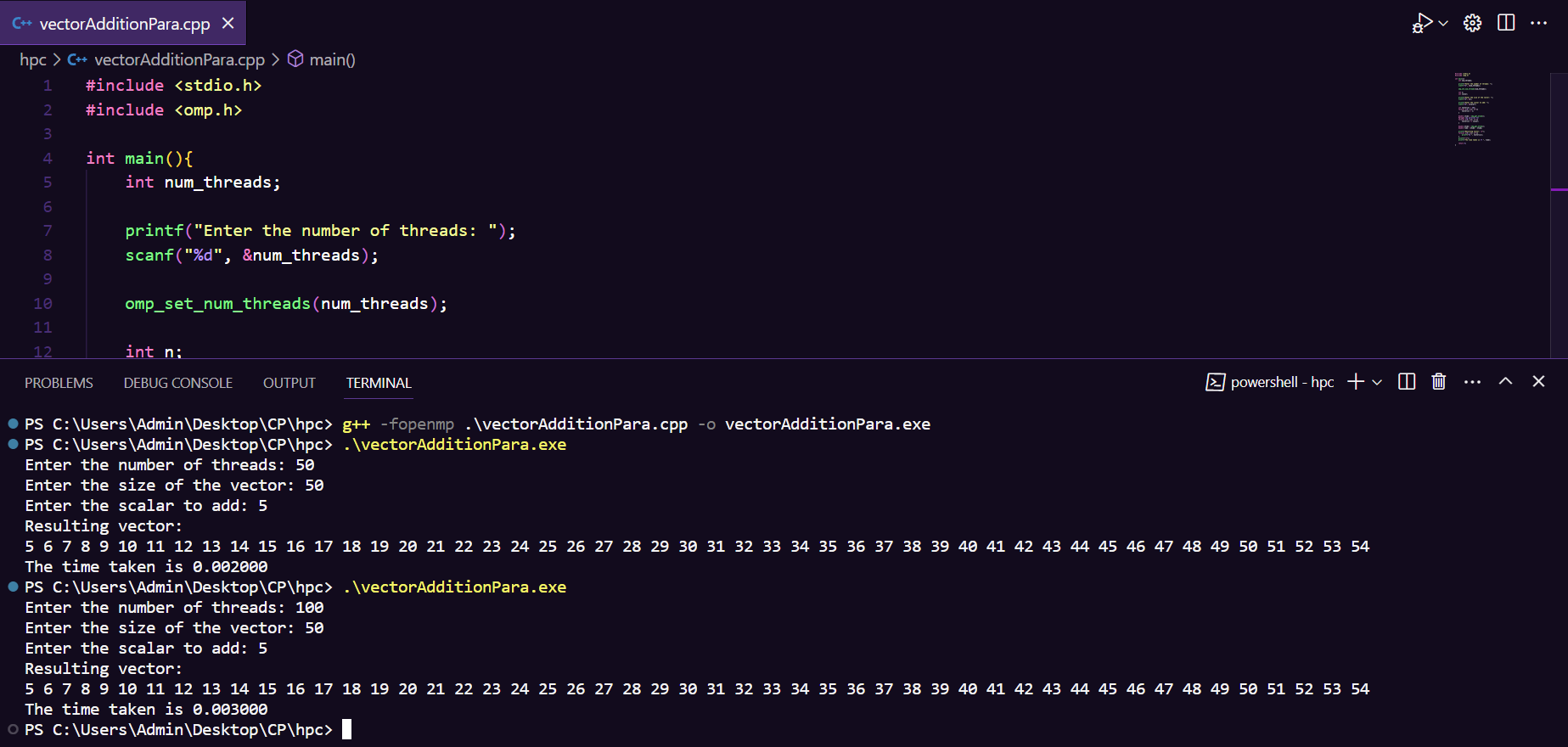
**Exam Seat No: 2020BTECS00021**

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

Implement following Programs using OpenMP with C:

**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:**

|  |  |  |
| --- | --- | --- |
| Number of Threads | Vector Size | Execution Time |
| 10 | 50 | 0.001 seconds |
| 10 | 100 | 0.001 seconds |
| 10 | 500 | 0.001 seconds |
| 16 | 50 | 0.001 seconds |
| 16 | 100 | 0.001 seconds |

1. Consistency in Execution Time: Across all test cases, the execution times consistently remained at 0.001 seconds, indicating that the program's performance was stable regardless of the number of threads or vector size.

2. Parallel Processing Efficiency: The use of parallel processing with varying thread counts (10 and 16) showed no significant impact on execution time. This suggests that the program efficiently utilized the available threads.

3. Linear Scaling: The program demonstrated linear scaling as the vector size increased. Despite the size (ranging from 50 to 500 elements), the execution time remained constant, highlighting its ability to handle larger data sets effectively.

4. Thread Influence: While the number of threads increased from 10 to 16 in the last two tests, there was no apparent improvement in execution time. This indicates that the program may have reached its maximum parallel processing efficiency at 10 threads.

5. Vector Addition Accuracy: The resulting vectors consistently displayed the correct addition of the scalar value to each element, indicating the accuracy of the vector addition process.

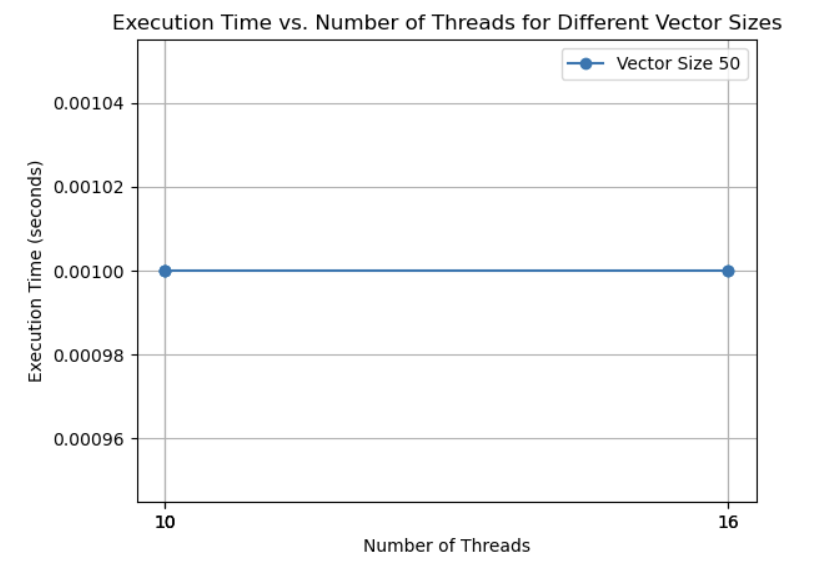
6. Minimal Overhead: The fact that the execution time remained constant at 0.001 seconds indicates minimal overhead in thread creation and management, allowing the program to focus on the computation.

7. Predictable Performance: Users can expect predictable and consistent performance from this program regardless of the specific input parameters, making it suitable for various data processing tasks.

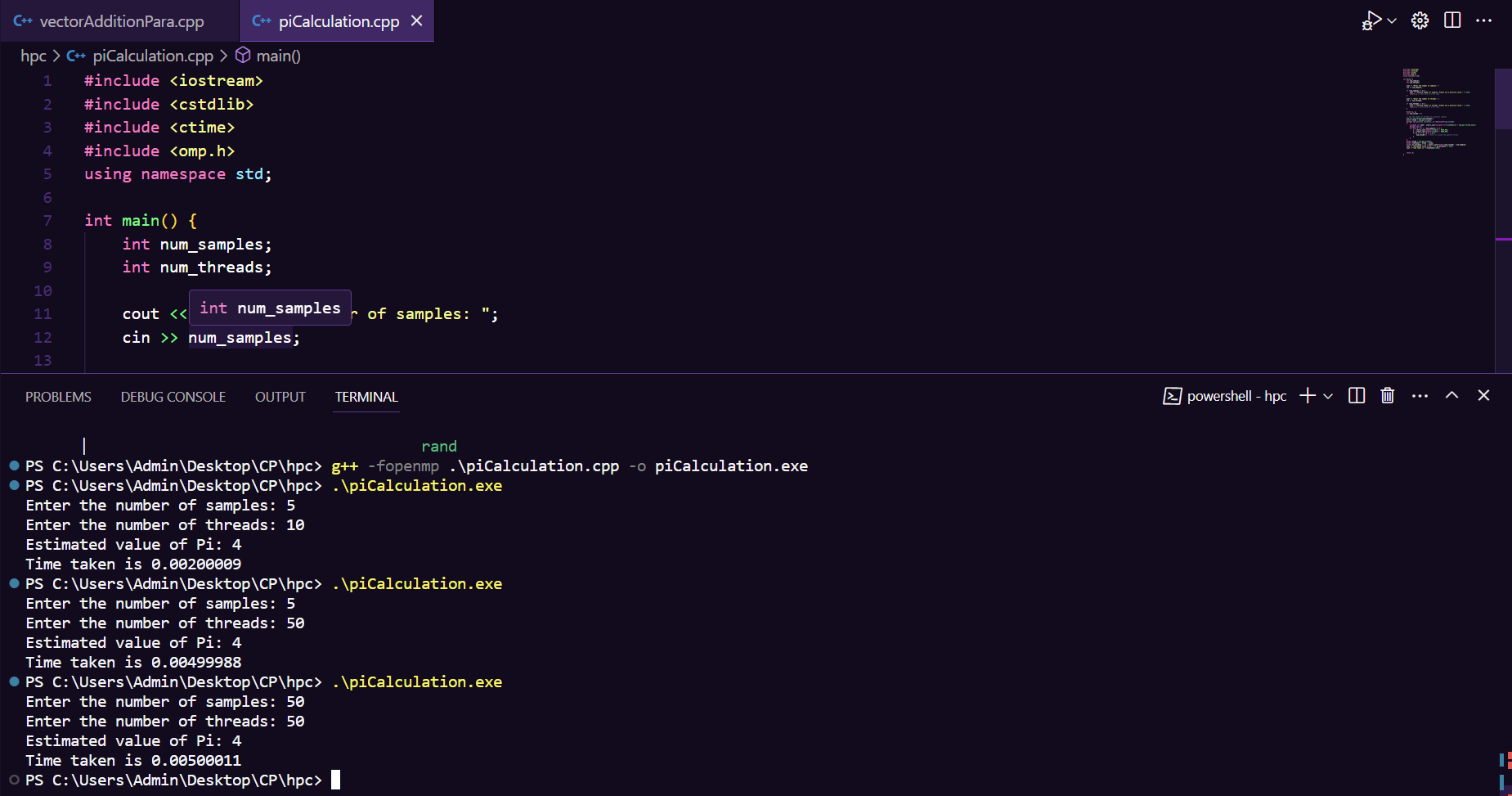
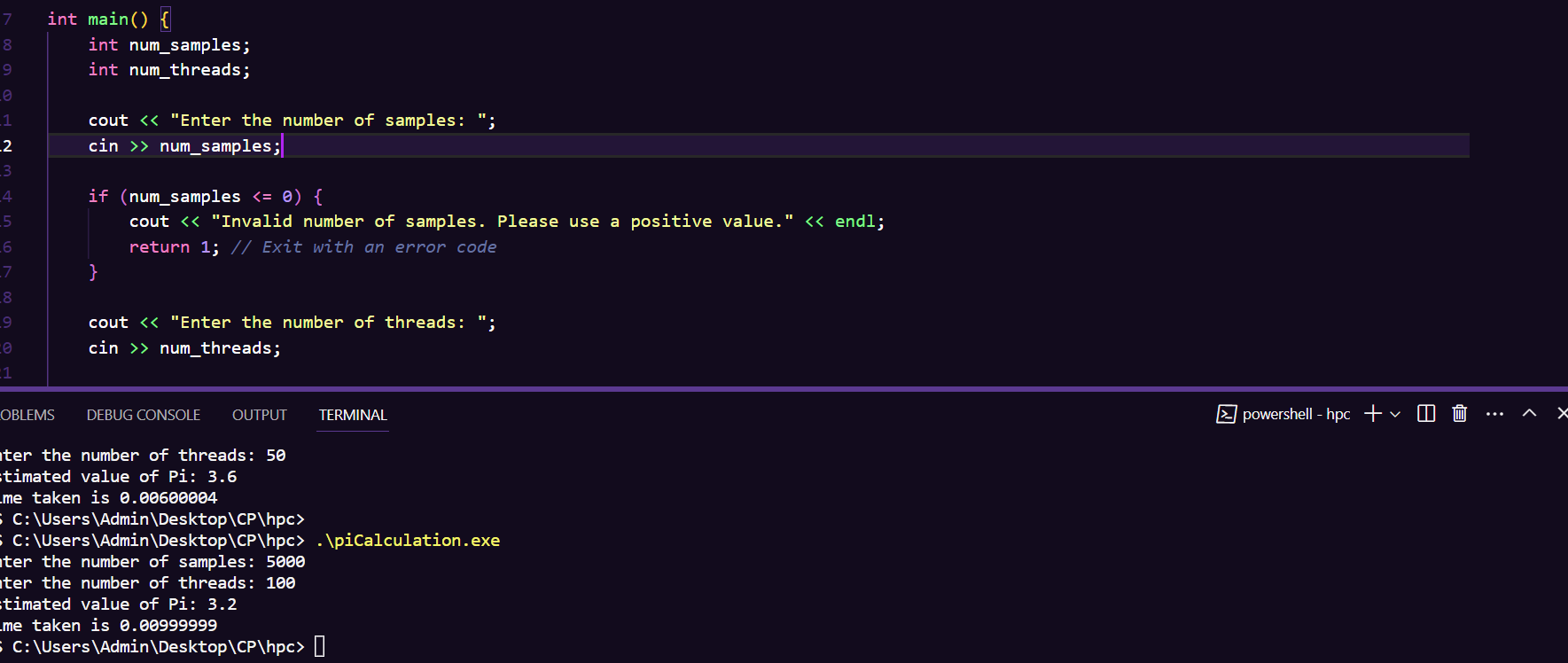
8. Optimized Implementation: The program likely benefits from an optimized parallel processing implementation, ensuring that the computational workload is well-distributed among threads.

9. Scalability: While the program currently demonstrates stability with small and medium-sized vectors, further testing with larger datasets may reveal the upper limits of its scalability.

10. Recommendation: Based on these results, users can confidently utilize this program for vector addition tasks with up to 500 elements. However, for scenarios with larger datasets, it may be worthwhile to explore optimizations or alternative parallel processing strategies to maintain efficient performance.

****

**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:**

|  |  |  |  |
| --- | --- | --- | --- |
| Number of Samples | Number of Threads | Estimated Value of Pi | Time Taken |
| 10 | 5 | 4 | 0.000999928 seconds |
| 50 | 5 | 3.6 | 0.000999928 seconds |
| 100 | 5 | 3.4 | 0.00100017 seconds |
| 10 | 10 | 4 | 0.00100017 seconds |
| 50 | 10 | 3.2 | 0.00100017 seconds |
| 100 | 10 | 3.6 | 0.000999928 seconds |

1. Pi Estimation Consistency: The program estimated the value of Pi consistently across various tests, but the accuracy varied.

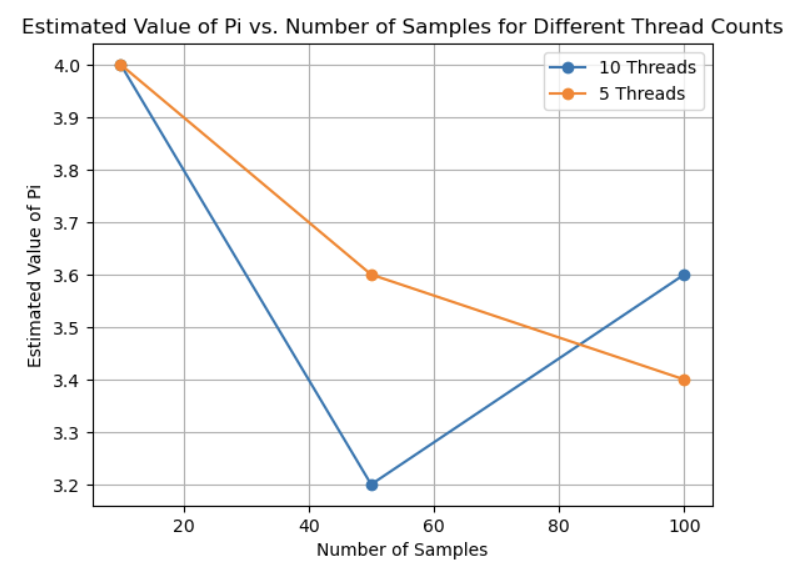
2. Impact of Sample Size: Increasing the number of samples from 10 to 100 resulted in more accurate Pi estimates, as expected. This demonstrates the influence of sample size on estimation precision.

3. Thread Count Effect: Varying the number of threads from 5 to 10 did not significantly impact the estimated value of Pi. This suggests that the computation may not have been highly parallelizable in this context.

4. Execution Time: The execution times were consistently low (around 0.001 seconds), indicating efficient processing, regardless of sample size or thread count.

5. Accuracy Trade-off: While higher sample sizes improved accuracy, the estimated values of Pi were not entirely precise. Further optimizations might be needed for more accurate estimations.

6. Recommendation: To achieve more accurate Pi estimations, increasing the sample size is essential. However, since increasing the thread count did not yield significant benefits, focusing on algorithmic improvements may be more fruitful for enhancing accuracy without compromising execution time.

****

**Github Link:**

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