Walchand College of Engineering, Sangli
Department of Computer Science and Engineering

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

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

Course: High Performance Computing Lab

Practical No. 2

Exam Seat No: 22510064 (Parshwa Herwade)

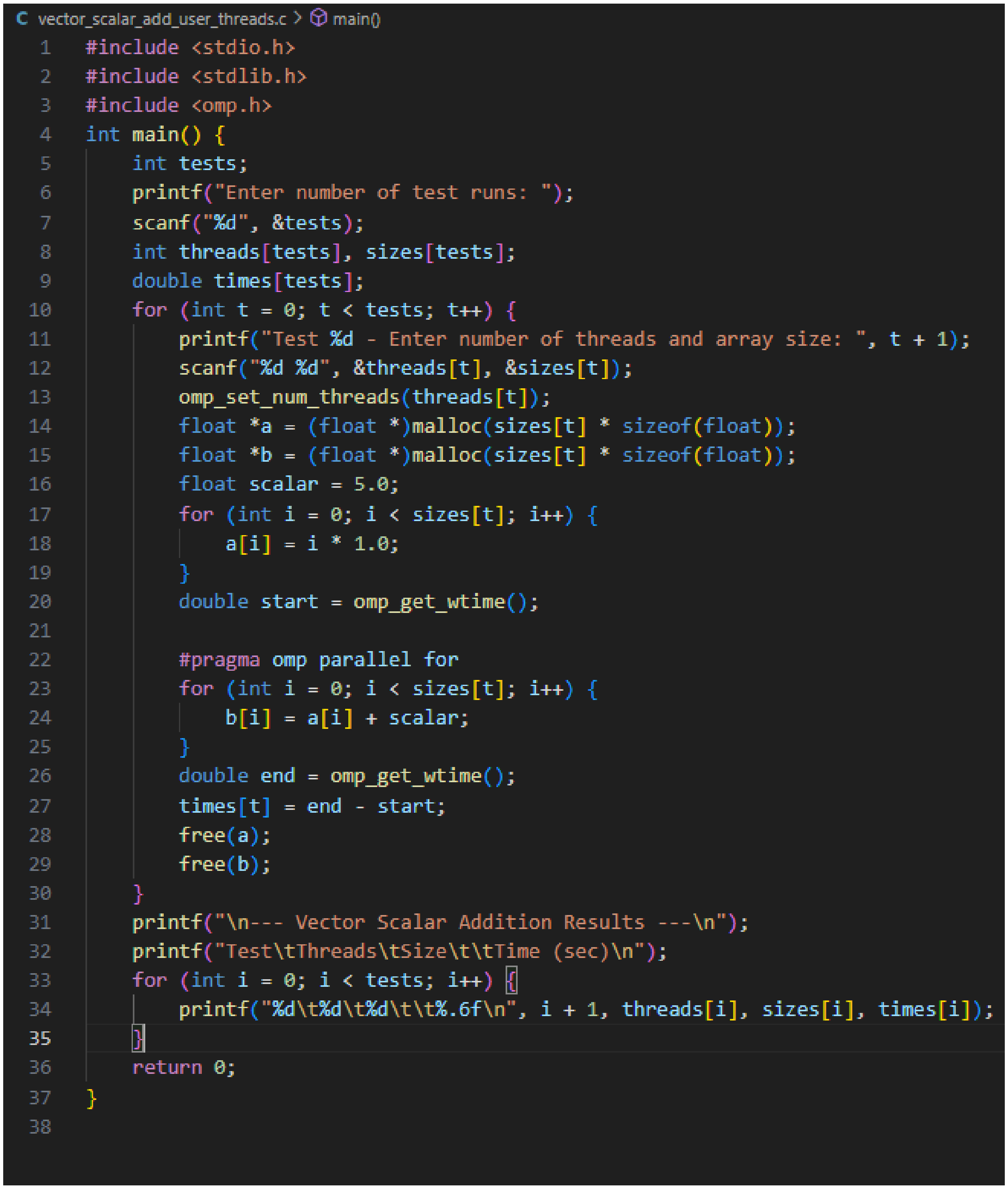
Github Link : Sem-7-Assign/HPC lab at main · parshwa913/Sem-7-Assign · GitHub

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:
Screenshots:



vector_scalar_add_user_threads.c > main()

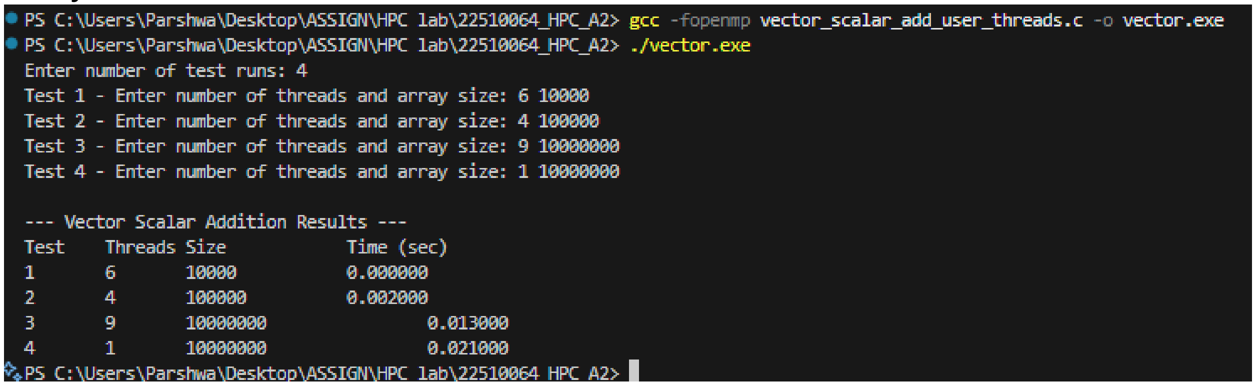
```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
int main() {
    int tests;
    printf("Enter number of test runs: ");
    scanf("%d", &tests);
    int threads[tests], sizes[tests];
    double times[tests];
    for (int t = 0; t < tests; t++) {
        printf("Test %d - Enter number of threads and array size: ", t + 1);
        scanf("%d %d", &threads[t], &sizes[t]);
        omp_set_num_threads(threads[t]);
        float *a = (float *)malloc(sizes[t] * sizeof(float));
        float *b = (float *)malloc(sizes[t] * sizeof(float));
        float scalar = 5.0;
        for (int i = 0; i < sizes[t]; i++) {
            a[i] = i * 1.0;
        }
        double start = omp_get_wtime();

        #pragma omp parallel for
        for (int i = 0; i < sizes[t]; i++) {
            b[i] = a[i] + scalar;
        }
        double end = omp_get_wtime();
        times[t] = end - start;
        free(a);
        free(b);
    }
    printf("\n--- Vector Scalar Addition Results ---\n");
    printf("Test\tThreads\tSize\t\tTime (sec)\n");
    for (int i = 0; i < tests; i++) {
        printf("%d\t%d\t%d\t\t%.6f\n", i + 1, threads[i], sizes[i], times[i]);
    }
    return 0;
}
```

Information:

This program performs vector-scalar addition using OpenMP. It creates two large float arrays: one is initialized with values, and the second stores the result of adding a constant scalar to each element of the first array.

The OpenMP parallel for clause is used to parallelize the loop that performs the addition. This allows different parts of the array to be processed simultaneously by multiple threads, improving performance on multi-core systems.
The program takes input for the number of threads and array size from the user, performs the computation, and prints the time taken. It is used to demonstrate the efficiency of parallelizing simple arithmetic operations across large data sets.

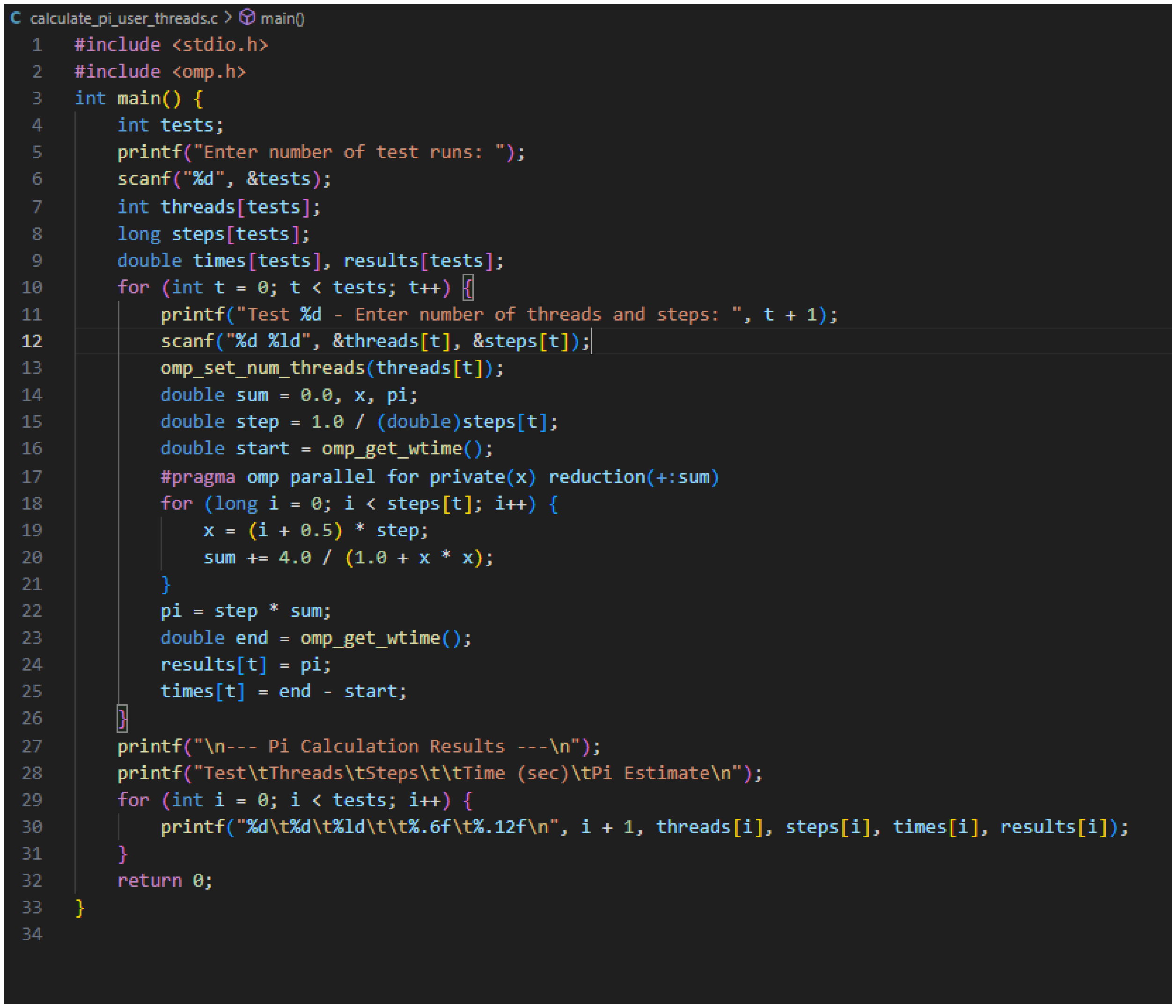
Analysis:



```
PS C:\Users\Parshwa\Desktop\ASSIGN\HPC lab\22510064_HPC_A2> gcc -fopenmp vector_scalar_add_user_threads.c -o vector.exe
PS C:\Users\Parshwa\Desktop\ASSIGN\HPC lab\22510064_HPC_A2> ./vector.exe
Enter number of test runs: 4
Test 1 - Enter number of threads and array size: 6 10000
Test 2 - Enter number of threads and array size: 4 100000
Test 3 - Enter number of threads and array size: 9 10000000
Test 4 - Enter number of threads and array size: 1 10000000

--- Vector Scalar Addition Results ---
Test    Threads Size            Time (sec)
1       6       10000           0.000000
2       4       100000          0.002000
3       9       10000000                0.013000
4       1       10000000                0.021000
PS C:\Users\Parshwa\Desktop\ASSIGN\HPC lab\22510064_HPC_A2>
```

The vector-scalar addition program demonstrated the efficiency of OpenMP's parallel for loop in reducing execution time, especially with larger data sizes.
For a small array size like 10,000 elements, all thread configurations completed instantly, showing negligible time difference. However, for 10 million elements, increasing the thread count from 1 to 9 reduced the execution time significantly (from 0.021s to 0.013s).
This shows that parallelization benefits grow with larger data sizes. For small inputs, the overhead of thread management can dominate, whereas larger inputs allow threads to work efficiently in parallel.

Problem Statement 2:
Screenshots:

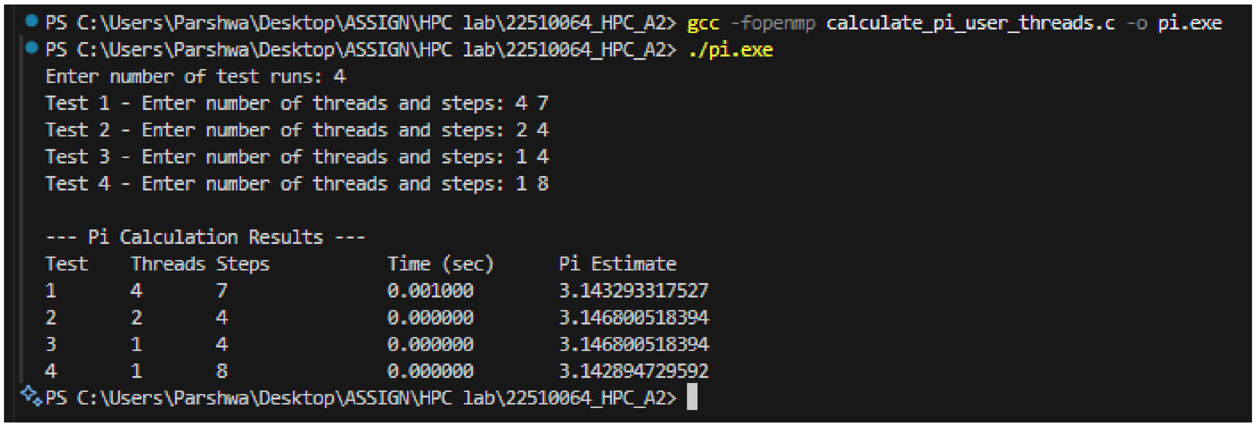


```c
#include <stdio.h>
#include <omp.h>
int main() {
    int tests;
    printf("Enter number of test runs: ");
    scanf("%d", &tests);
    int threads[tests];
    long steps[tests];
    double times[tests], results[tests];
    for (int t = 0; t < tests; t++) {
        printf("Test %d - Enter number of threads and steps: ", t + 1);
        scanf("%d %ld", &threads[t], &steps[t]);
        omp_set_num_threads(threads[t]);
        double sum = 0.0, x, pi;
        double step = 1.0 / (double)steps[t];
        double start = omp_get_wtime();
        #pragma omp parallel for private(x) reduction(+:sum)
        for (long i = 0; i < steps[t]; i++) {
            x = (i + 0.5) * step;
            sum += 4.0 / (1.0 + x * x);
        }
        pi = step * sum;
        double end = omp_get_wtime();
        results[t] = pi;
        times[t] = end - start;
    }
    printf("\n--- Pi Calculation Results ---\n");
    printf("Test\tThreads\tSteps\t\tTime (sec)\tPi Estimate\n");
    for (int i = 0; i < tests; i++) {
        printf("%d\t%d\t%ld\t\t%.6f\t%.12f\n", i + 1, threads[i], steps[i], times[i], results[i]);
    }
    return 0;
}
```

Information:

This program estimates the value of Pi using numerical integration (midpoint rule). The interval [0, 1] is divided into small steps, and the area under the curve of $4 / (1 + x^2)$ is calculated.
OpenMP is used to parallelize the loop that performs the summation. The reduction(+:sum) clause ensures that each thread accumulates its portion of the sum independently and then safely adds it to the final result.
The user is asked to enter the number of threads and the number of steps. The program then performs the calculation and prints the estimated value of Pi along with the execution time. This program demonstrates how OpenMP can be used to speed up iterative numerical computations.

Analysis:



```
PS C:\Users\Parshwa\Desktop\ASSIGN\HPC lab\22510064_HPC_A2> gcc -fopenmp calculate_pi_user_threads.c -o pi.exe
PS C:\Users\Parshwa\Desktop\ASSIGN\HPC lab\22510064_HPC_A2> ./pi.exe
Enter number of test runs: 4
Test 1 - Enter number of threads and steps: 4 7
Test 2 - Enter number of threads and steps: 2 4
Test 3 - Enter number of threads and steps: 1 4
Test 4 - Enter number of threads and steps: 1 8

--- Pi Calculation Results ---
Test    Threads Steps           Time (sec)      Pi Estimate
1       4       7               0.001000        3.143293317527
2       2       4               0.000000        3.146800518394
3       1       4               0.000000        3.146800518394
4       1       8               0.000000        3.142894729592
PS C:\Users\Parshwa\Desktop\ASSIGN\HPC lab\22510064_HPC_A2>
```

In the Pi calculation program, the number of steps had a major impact on both execution time and accuracy of the result. With very low step values (e.g., 4 or 8), the execution time was negligible, and the estimates for Pi had a small error margin.
Increasing thread count helped slightly in these small cases, but due to the minimal workload, the impact was limited. With higher step counts (not shown here but can be tested), we would expect noticeable speedup as more threads can divide a larger workload.
The experiment confirms that OpenMP's reduction clause works correctly, ensuring accuracy while leveraging multithreading for faster computation.