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: 2020BTECS00042

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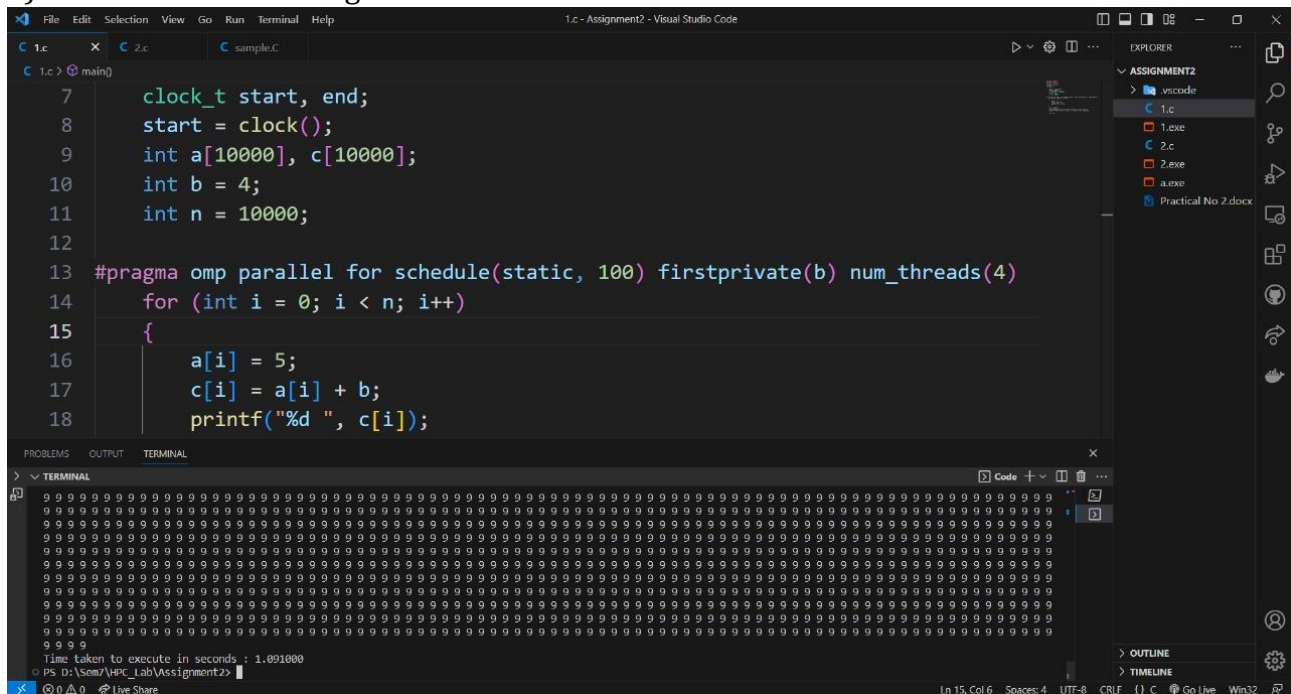   Analyse the performance of your programs for different number of threads and Data size.

Problem Statement 1: Vector Scalar Addition
Screenshots:

```c
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
#include <time.h>
int main()
{
    clock_t start, end;
    start = clock();
    int a[10000], c[10000];
    int b = 4;
    int n = 10000;
#pragma omp parallel for schedule(static, 100) firstprivate(b) num_threads(4)
    for (int i = 0; i < n; i++)
    {
        a[i] = 5;
        c[i] = a[i] + b;
        printf("%d ", c[i]);
    }
    end = clock();
    double duration = ((double)end - start) / CLOCKS_PER_SEC;
    printf("\nTime taken to execute in seconds : %f", duration);
    return 0;
}
```
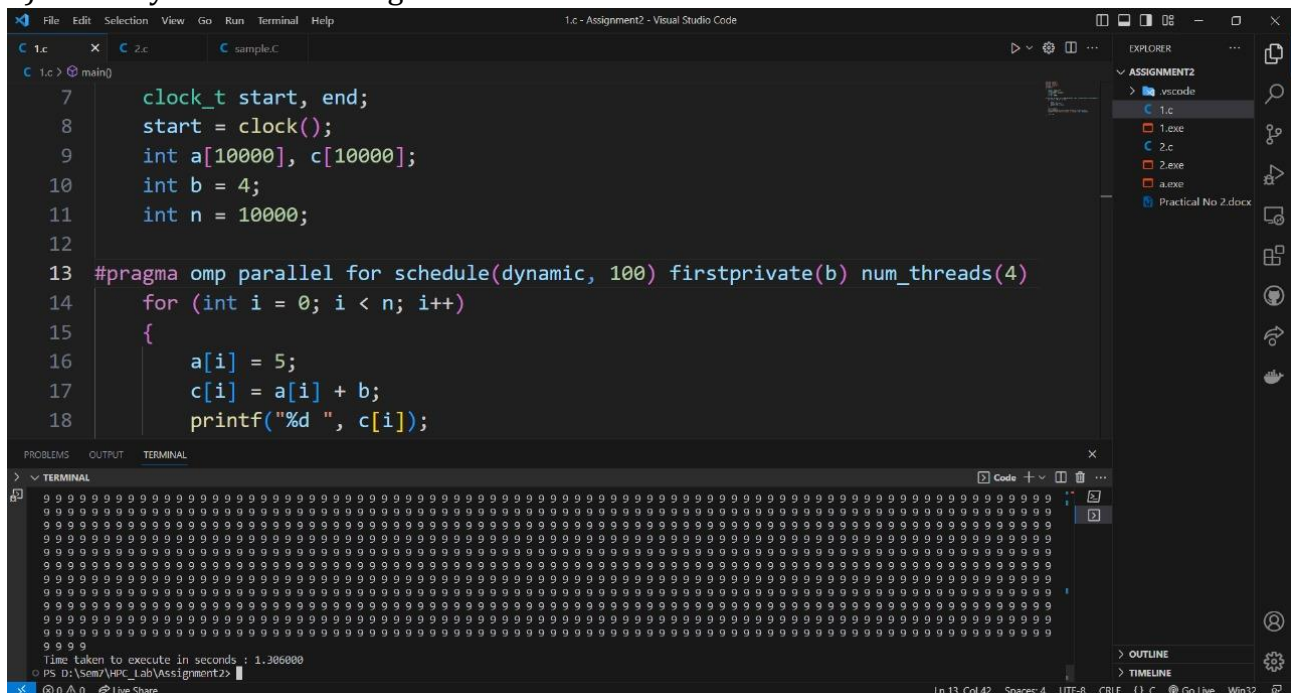
Final Year: High Performance Computing Lab 2023-24 Sem I

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

Output:
1) When static scheduling is used
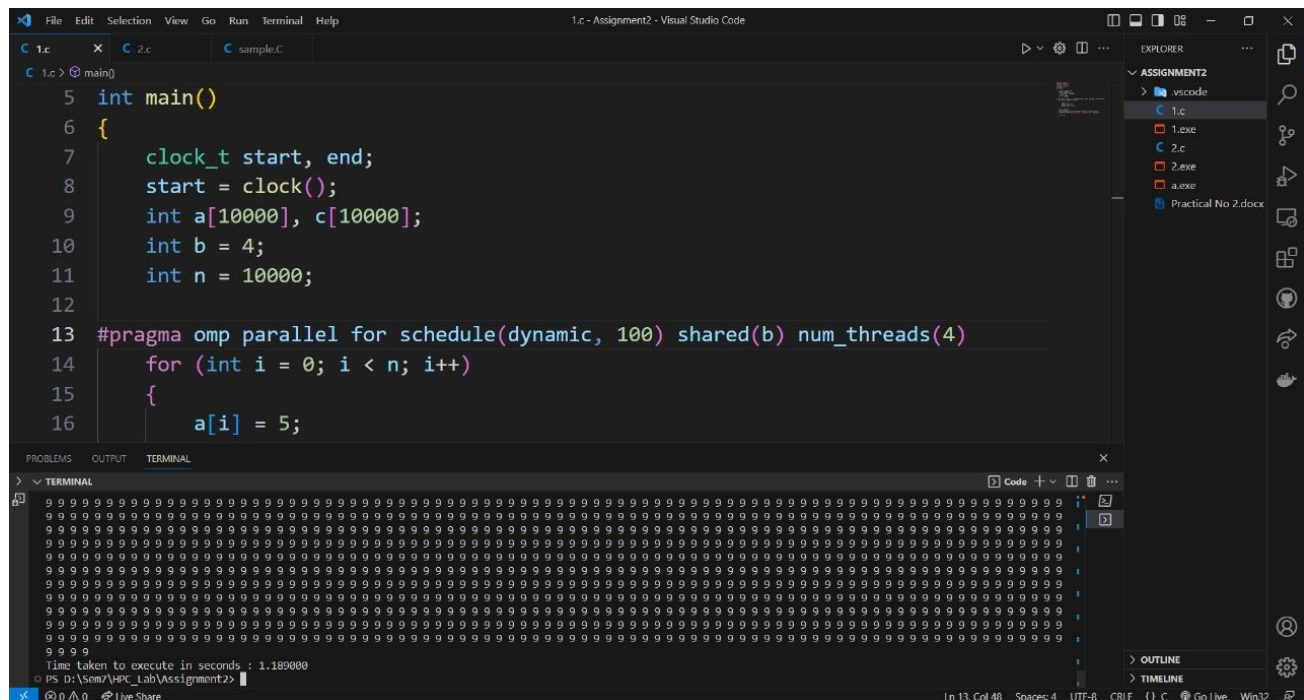
2) When dynamic scheduling is used:

3) When variable is shared among threads using shared clause:

Final Year: High Performance Computing Lab 2023-24 Sem I

Information:

OpenMP provides several clauses that control the data sharing and data visibility among threads. Among these, the "private," "firstprivate," and "shared" clauses are commonly used to manage data in parallel regions.

1) **Private:** The private clause is used to declare variables as private to each thread within a parallel region. Each thread gets its own private copy of the variable, and any modifications to it within the parallel region do not affect the original variable outside the region. Once the parallel region is exited, the private variables are typically undefined.

2) **Firstprivate:** The firstprivate clause combines the behavior of both private and shared clauses. It creates a private copy of the variable for each thread, initializes it with the value from outside the parallel region, and then allows modifications within the region. After the parallel region, the original variable retains its value, while the private copies hold the modified values.

3) **Shared:** The shared clause (sometimes implied) indicates that a variable should be shared among all threads within a parallel region. This means all threads can access and modify the same memory location, potentially leading to race conditions if not properly synchronized.

In OpenMP, the schedule clause is used to control how loop iterations are divided and assigned to threads in a parallel loop construct. The two commonly used scheduling strategies are "static" and "dynamic."

1) **Static schedule:** The schedule(static, chunk_size) clause divides the loop iterations into equal-sized chunks, where each chunk consists of a specified number of loop iterations. These chunks are then assigned to threads in a round-robin manner.

Final Year: High Performance Computing Lab 2023-24 Sem I

Static scheduling can be advantageous when the workload of each iteration is relatively uniform.

2) **Dynamic schedule:** The schedule(dynamic, chunk_size) clause divides the loop iterations into chunks of the specified size and assigns these chunks to threads on a first-come, first-served basis. When a thread finishes its assigned chunk, it can request another chunk of iterations, leading to a more dynamic distribution of work. Dynamic scheduling can be useful when the workload of each iteration is unpredictable or when iterations have varying execution times.

Analysis:

1) Since our task for each thread is uniform, static scheduling gives the better result than dynamic scheduling.

2) When each thread doesn't have its private copy of the variable, accesses the variable from shared memory. As we know at a time only one thread can access the same memory location. So here access becomes shared though we try to parallelize. Hence, in case of shared, some extra time is required.

Number of Threads	Data Size	Sequential Time	Parallel Time
4	1000	0.00000	0.00100
4	10000	0.00000	0.00200
4	100000	0.00200	0.00200
8	100000	0.00200	0.00300
4	1000000	0.00180	0.00140
4	10000000	0.24200	0.08500
8	10000000	0.24200	0.13700
4	200000000	2.88300	1.82100

Problem Statement 2:
Screenshots:

```c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <omp.h>
#include <time.h>
int main()
{
    clock_t start, end;
    start = clock();
    int total_points = 10000000;   // Total number of points
    int points_inside_circle = 0; // Counter for points inside the quarter circle
    // Seed the random number generator
    srand(time(NULL));
```
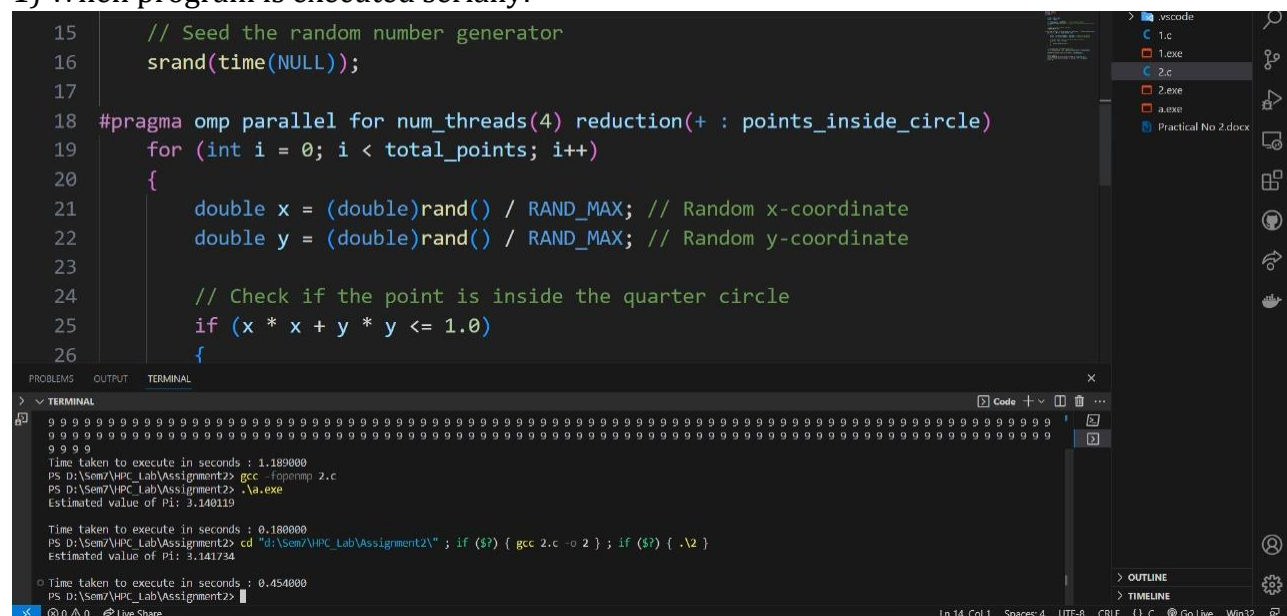
Final Year: High Performance Computing Lab 2023-24 Sem I

```
#pragma omp parallel for num_threads(4) reduction(+ : points_inside_circle)
    for (int i = 0; i < total_points; i++)
    {
        double x = (double)rand() / RAND_MAX; // Random x-coordinate
        double y = (double)rand() / RAND_MAX; // Random y-coordinate

        // Check if the point is inside the quarter circle
        if (x * x + y * y <= 1.0)
        {
            points_inside_circle++;
        }
    }
    // Calculate the estimated value of Pi
    double estimated_pi = 4.0 * points_inside_circle / total_points;
    printf("Estimated value of Pi: %lf\n", estimated_pi);
    end = clock();
    double duration = ((double)end - start) / CLOCKS_PER_SEC;
    printf("\nTime taken to execute in seconds : %f", duration);
    return 0;
}
```

Output:

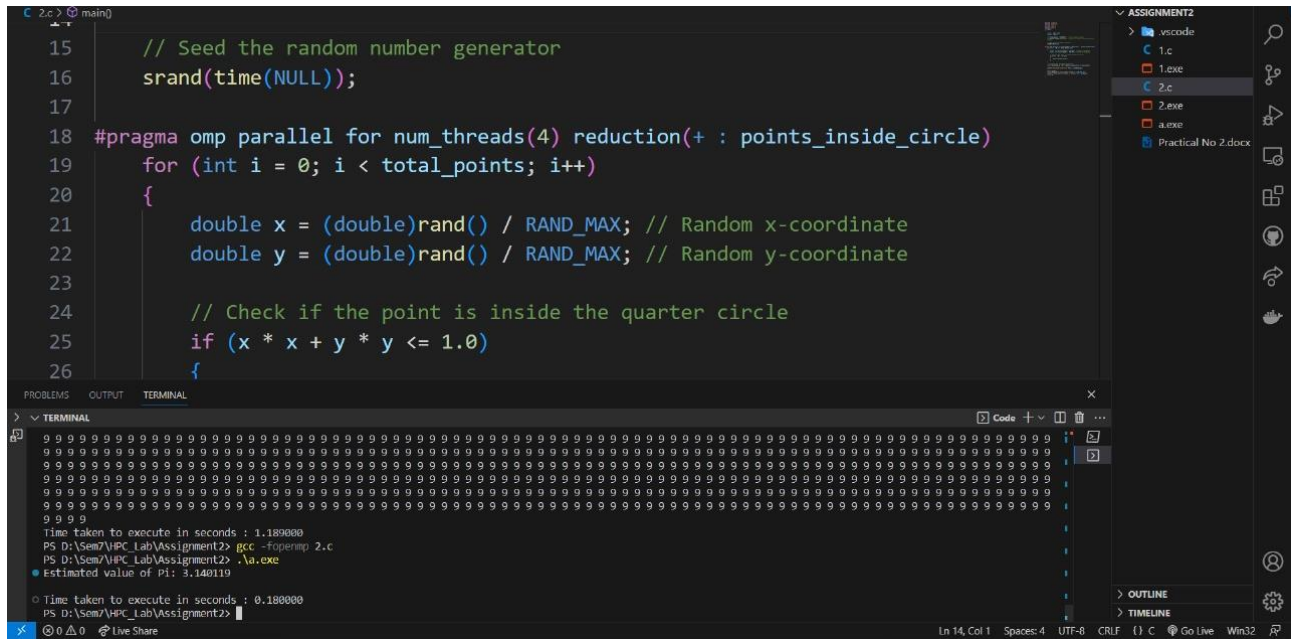1) When program is executed serially:

```
15        // Seed the random number generator
16        srand(time(NULL));
17
18    #pragma omp parallel for num_threads(4) reduction(+ : points_inside_circle)
19        for (int i = 0; i < total_points; i++)
20        {
21            double x = (double)rand() / RAND_MAX; // Random x-coordinate
22            double y = (double)rand() / RAND_MAX; // Random y-coordinate
23
24            // Check if the point is inside the quarter circle
25            if (x * x + y * y <= 1.0)
26            {
```

```
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9 9
9 9 9 9
Time taken to execute in seconds : 1.189000
PS D:\Sem7\HPC_Lab\Assignment2> gcc -fopenmp 2.c
PS D:\Sem7\HPC_Lab\Assignment2> .\a.exe
Estimated value of Pi: 3.140119

Time taken to execute in seconds : 0.180000
PS D:\Sem7\HPC_Lab\Assignment2> cd "d:\Sem7\HPC_Lab\Assignment2\" ; if ($?) { gcc 2.c -o 2 } ; if ($?) { .\2 }
Estimated value of Pi: 3.141734

Time taken to execute in seconds : 0.454000
PS D:\Sem7\HPC_Lab\Assignment2>
```

2) When program is executed parallelly:

Final Year: High Performance Computing Lab 2023-24 Sem I

```c
15    // Seed the random number generator
16    srand(time(NULL));
17
18 #pragma omp parallel for num_threads(4) reduction(+ : points_inside_circle)
19    for (int i = 0; i < total_points; i++)
20    {
21        double x = (double)rand() / RAND_MAX; // Random x-coordinate
22        double y = (double)rand() / RAND_MAX; // Random y-coordinate
23
24        // Check if the point is inside the quarter circle
25        if (x * x + y * y <= 1.0)
26        {
```

Information:

In OpenMP, the **parallel for** construct is used to parallelize the execution of a loop across multiple threads. This construct combines the parallel construct (which creates a parallel region) with the for construct (which specifies a loop to be parallelized).

The **reduction** clause in OpenMP is used to perform a reduction operation on a variable across multiple threads in a parallel region. Reductions are commonly used to accumulate results from multiple threads into a single result, such as summing up values or finding the maximum value. The reduction clause simplifies the process of maintaining thread-local copies of the variable and combining the results at the end of the parallel region.

Analysis:

We get the better speedup with the parallel program of the calculation of pi value. Here we are using parallel for clause and reduction to get our tasks run with parallel threads.
We can see that, over the sequential, it gives better result and as we increase the number of threads from 2 to 4, speedup increases. But as we further increase the number of threads, we cannot get better speedup as waiting time of threads increase due to unavailability of the physical threads.

Number of Threads	Data Size (Iterations)	Sequential Time	Parallel Time
4	100000000	4.62900	1.58900
8	100000000	4.62900	1.61200
12	100000000	4.62900	1.56200
16	100000	4.62900	1.56000

Github Link: https://github.com/sahilthakur007/HPC-Assignments.git

Final Year: High Performance Computing Lab 2023-24 Sem I