Assignment No. 2

PRN: 2020BTECS00025
Name: Sahil Santosh Otari
Course: High Performance Computing Lab
Title of practical: Study and implementation of Basic Open MP clauses.

1. Vector Scalar Addition
   Sequential:

```
#include <omp.h>
#include <pthread.h>
#include <stdio.h>

int main() {
    int N = 200;
    int A[200];
    for (int i = 0; i < N; i++)
        A[i] = 10;

    int B[200];
    for (int i = 0; i < N; i++)
        B[i] = 20;

    int C[200] = {0};
    double startTime = omp_get_wtime();
    for (int i = 0; i < N; i++) {
        C[i] = A[i] + B[i];
                        printf("Index:   %d   Thread:   %d\n",   i,
omp_get_thread_num());
    }

    for (int i = 0; i < N; i++) {
        printf("%d ", C[i]);
    }
```

```
    double endTime = omp_get_wtime();
    double time = endTime - startTime;
    printf("Time taken is %f\n", time);
    return 0;
}
```

Output:

```
Index: 182 Thread: 0
Index: 183 Thread: 0
Index: 184 Thread: 0
Index: 185 Thread: 0
Index: 186 Thread: 0
Index: 187 Thread: 0
Index: 188 Thread: 0
Index: 189 Thread: 0
Index: 190 Thread: 0
Index: 191 Thread: 0
Index: 192 Thread: 0
Index: 193 Thread: 0
Index: 194 Thread: 0
Index: 195 Thread: 0
Index: 196 Thread: 0
Index: 197 Thread: 0
Index: 198 Thread: 0
Index: 199 Thread: 0
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
Time taken is 0.047000
PS D:\HPC\Assignment2>
```

We have performs element-wise addition of two arrays `A` and `B`, storing the result in array `C`. It also measures and prints the time taken for the computation. The program utilizes OpenMP for parallelism, enabling multiple threads to execute the addition concurrently.

Information:

The code utilizes OpenMP to create a parallel region where multiple threads can execute the addition of array elements concurrently.

It defines three arrays: `A` and `B` of size 200, both initialized with constant values, and `C` of the same size to store the result.

A timer (`omp_get_wtime()`) is used to measure the execution time of the parallel computation.
After the addition of elements, the code prints the index of the element being processed and the thread number executing that addition.
Finally, it prints the elements of array `C` and the time taken for the computation.

Source Code (Parallel):

```
#include <omp.h>
#include <pthread.h>
#include <stdio.h>

int main() {
    int N = 200;
    int A[200];
    for (int i = 0; i < N; i++)
        A[i] = 10;

    int B[200];
    for (int i = 0; i < N; i++)
        B[i] = 20;

    int C[200] = {0};
    double startTime = omp_get_wtime();
    #pragma omp parallel for reduction(+ : C)
    for (int i = 0; i < N; i++) {
        C[i] = A[i] + B[i];
                printf("Index is %d Thread is %d\n", i,
omp_get_thread_num());
    }

    for (int i = 0; i < N; i++) {
        printf("%d ", C[i]);
    }
```

```
    double endTime = omp_get_wtime();
    double time = endTime - startTime;
    printf("\nTime taken is %f\n", time);
    return 0;
}
```

Output:

```
Index is 36 Thread is 1
Index is 37 Thread is 1
Index is 38 Thread is 1
Index is 39 Thread is 1
Index is 40 Thread is 1
Index is 41 Thread is 1
Index is 42 Thread is 1
Index is 43 Thread is 1
Index is 44 Thread is 1
Index is 45 Thread is 1
Index is 46 Thread is 1
Index is 47 Thread is 1
Index is 48 Thread is 1
Index is 49 Thread is 1
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30 30
Time taken is 0.041000
PS D:\HPC\Assignment2>
```

We perform element-wise addition of two arrays `A` and `B`, storing the result in array `C`. It also measures and prints the time taken for the computation. The program utilizes OpenMP for parallelism, enabling multiple threads to execute the addition concurrently.

Information:

The code is similar to the previous version but with the addition of the `reduction` clause in the OpenMP parallel for loop.

The `reduction` clause is used to specify a reduction operation (in this case, addition) that is applied to a shared variable (`C` in this case) within the parallel loop.

The `reduction(+ : C)` clause ensures that each thread maintains a private copy of `C`, performs the addition operation on its private copy, and then combines the

results of all threads using the specified reduction operation (addition) to update the shared variable `C`.

Analysis:

1. The `reduction` clause improves code performance by eliminating data races and efficiently aggregating the results of parallel computations.
2. The code structure remains the same as the previous version, with a loop that distributes the work among multiple threads using OpenMP.
3. Each thread independently computes the element-wise addition of arrays `A` and `B` and stores the result in its private copy of array `C`.
4. After the loop, the reduction operation sums up the private copies of `C` from all threads and updates the shared `C` array with the final result.
5. The code measures and prints the execution time, which can be helpful for performance analysis and optimization.
6. The use of the `reduction` clause simplifies the code by automatically handling the aggregation of results in a thread-safe manner.
7. Like the previous version, the code doesn't explicitly specify the number of threads, so the number of threads used depends on the system's default settings or any environment variable that may have been set to control the number of threads.

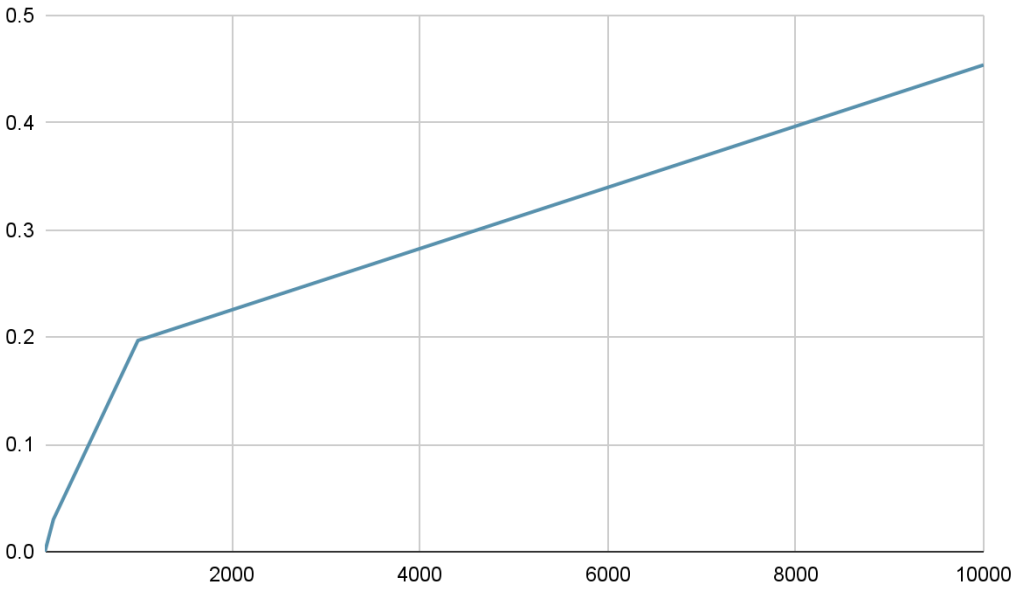
Table :

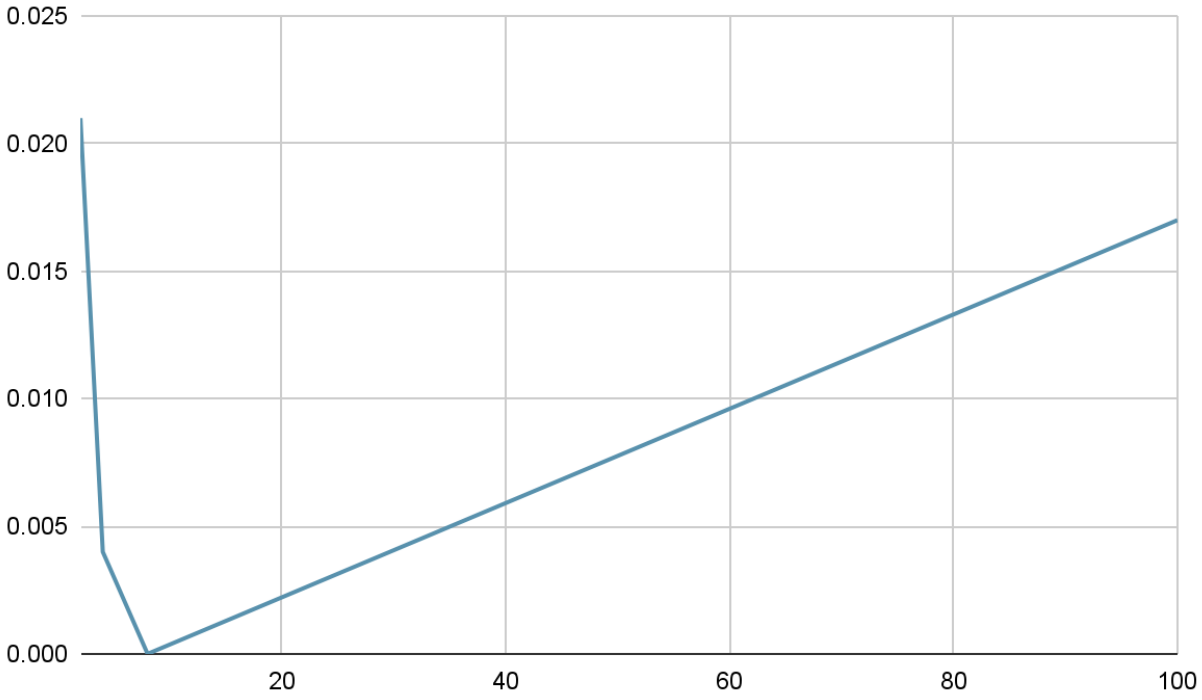
Threads: 4

Input Size	10	100	1000	10000
Time	0.000000	0.030000	0.197000	0.454000

Thread No.	2	4	8	100
Time	0.021000	0.004000	0.000000	0.017000

Graph:





2. Calculation of value of Pi (Source Code):

Sequential:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
```

```
#define N 100000

int main() {
    int i, num_inside = 0;
    double x, y;

    // Set the seed for random number generation
    srand(52525);
    double startTime = omp_get_wtime();
            #pragma    omp    parallel    for    private(x,    y)
reduction(+:num_inside)
    for (i = 0; i < N; i++) {
        // Generate random points within the unit square
        x = (double)rand() / RAND_MAX;
        y = (double)rand() / RAND_MAX;

            // Check if the point is inside the quarter-circle
(radius <= 1)
        if (x * x + y * y <= 1.0) {
            num_inside++;
        }
    }

    // Estimate Pi using the Monte Carlo method
    double pi = 4.0 * num_inside / N;

    printf("Estimated Pi Value: %f\n", pi);

    double endTime = omp_get_wtime();
    double time = endTime - startTime;
    printf("\nTime taken is %f\n", time);
    return 0;
```

```
}
```

Output:

```
PS D:\HPC\Assignment2> gcc -o PiValueSeq -fopenmp PiValueSeq.cpp
PS D:\HPC\Assignment2> .\PiValueSeq
Estimated Pi Value: 3.116840

Time taken is 0.001000
PS D:\HPC\Assignment2>
```

Source Code (Parallel)

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

#define N 100000

int main() {
    int i, num_inside = 0;
    double x, y;

    // Set the seed for random number generation
    srand(52525);
    double startTime = omp_get_wtime();
    #pragma omp parallel private(x, y)
    {
        int local_inside = 0;

        #pragma omp for
        for (i = 0; i < N; i++) {
            // Generate random points within the unit square
            x = (double)rand() / RAND_MAX;
            y = (double)rand() / RAND_MAX;
```

```
            // Check if the point is inside the quarter-circle
(radius <= 1)
            if (x * x + y * y <= 1.0) {
                local_inside++;
            }
        }

        // Accumulate local_inside_circle values from all threads
        #pragma omp atomic
        num_inside += local_inside;
    }

    // Estimate Pi using the Monte Carlo method
    double pi = 4.0 * num_inside / N;

    printf("Estimated Pi: %f\n", pi);
    double endTime = omp_get_wtime();
    double time = endTime - startTime;
    printf("\nTime taken is %f\n", time);
    return 0;
}
```

Output:

```
PS D:\HPC\Assignment2> gcc -o PiValuePar -fopenmp PiValuePar.cpp
PS D:\HPC\Assignment2> .\PiValuePar
Estimated Pi: 3.116840

Time taken is 0.000000
PS D:\HPC\Assignment2>
```

Analysis:

1. The Monte Carlo method is a statistical approach to estimating Pi by randomly sampling points. As the number of points generated (`N`) increases, the accuracy of the estimation improves.

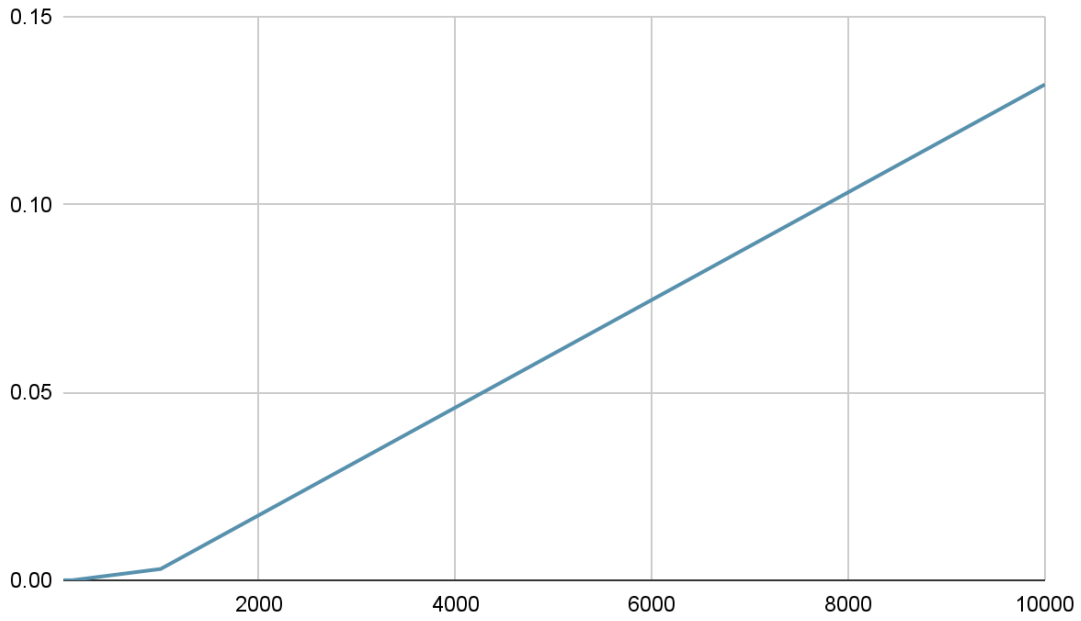
2. Open MP is used to distribute the work among multiple threads, which can significantly speed up the computation, especially when many points are involved. Each thread works independently on its portion of the points.
3. The code ensures thread safety by using the `private` clause to give each thread its private copy of variables, and by using the `reduction` clause to safely accumulate the counts.
4. The accuracy of the estimated Pi value depends on the number of points generated. You can increase `N` for a more accurate estimation.
5. Keep in mind that the Monte Carlo method provides an approximation, and the accuracy increases with more iterations. The estimated Pi value will be close to the actual Pi value, but it won't be exact.
6. This code serves as a basic example of parallelism with Open MP and illustrates how to perform parallel computations for Monte Carlo simulations, making it useful for educational and introductory purposes.

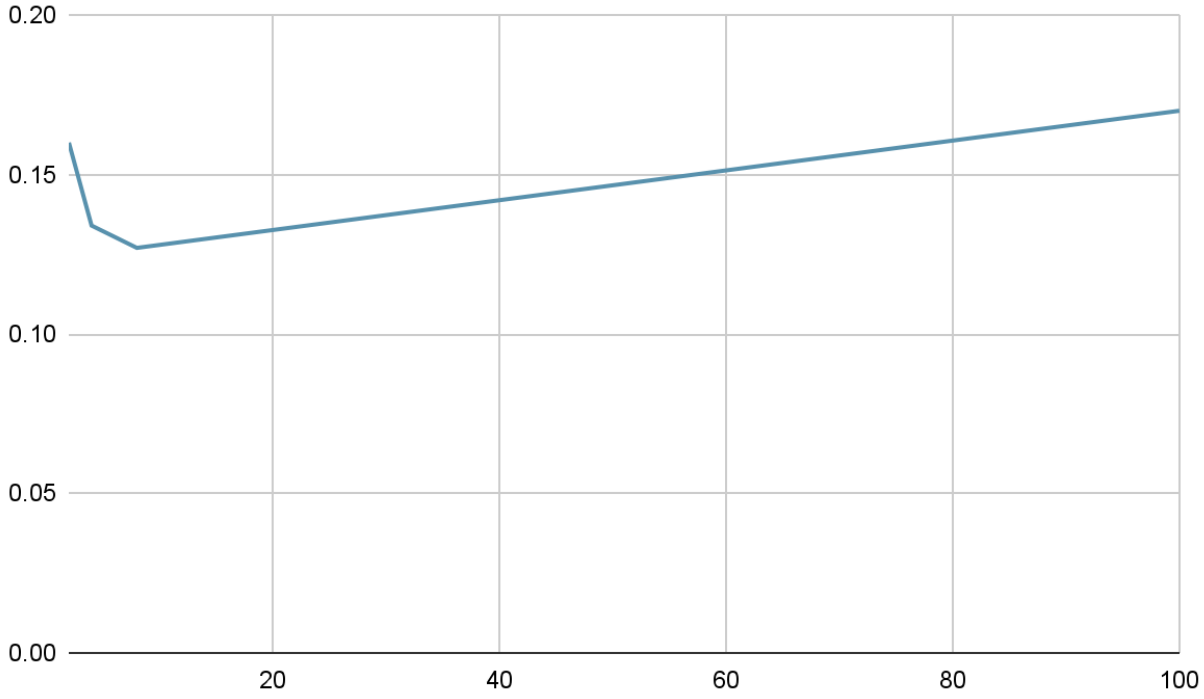
Number of Threads = 4

Input Size	10	100	1000	10000
Time	0.000000	0.000000	0.003000	0.132000

For N = 10000000

Thread No.	2	4	8	100
Time	0.160000	0.134000	0.127000	0.170000




0.20
0.15
0.10
0.05
0.00
20
40
60
80
100