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

**Year:** 2024-25 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 2**

**Exam Seat No: 21510013**

**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:**

**#include** <stdio.h>

**#include** <omp.h>

**#include** <time.h>

**int** main()

{

**int** n **=** 10;

**int** scalar **=** 5;

**int** vector[n];

**int** result\_seq[n];

**int** result\_par[n];

**for** (**int** i **=** 0; i **<** n; i**++**)

    {

        vector[i] **=** i;

    }

*// Sequential vector-scalar addition*

**clock\_t** start\_seq **=** clock();

**for** (**int** i **=** 0; i **<** n; i**++**)

    {

        result\_seq[i] **=** vector[i] **+** scalar;

    }

**clock\_t** end\_seq **=** clock();

*// Parallel vector-scalar addition*

**double** start\_par **=** omp\_get\_wtime();

**#pragma** **omp** **parallel** **for**

**for** (**int** i **=** 0; i **<** n; i**++**)

    {

        result\_par[i] **=** vector[i] **+** scalar;

    }

**double** end\_par **=** omp\_get\_wtime();

*// Output results*

    printf("Sequential Vector-Scalar Addition Result:\n");

**for** (**int** i **=** 0; i **<** n; i**++**)

    {

        printf("%d ", result\_seq[i]);

    }

    printf("\n");

    printf("Parallel Vector-Scalar Addition Result:\n");

**for** (**int** i **=** 0; i **<** n; i**++**)

    {

        printf("%d ", result\_par[i]);

    }

    printf("\n");

*// Execution time analysis*

**double** time\_seq **=** (**double**)(end\_seq **-** start\_seq) **/** CLOCKS\_PER\_SEC;

**double** time\_par **=** end\_par **-** start\_par;

    printf("Sequential Execution Time: %f seconds\n", time\_seq);

    printf("Parallel Execution Time: %f seconds\n", time\_par);

*// Calculate Speedup and Efficiency*

**double** speedup **=** time\_seq **/** time\_par;

**int** num\_threads **=** omp\_get\_max\_threads();

**double** efficiency **=** speedup **/** num\_threads;

    printf("Speedup: %f\n", speedup);

    printf("Efficiency: %f\n", efficiency);

*// Analysis of execution times*

**if** (time\_seq **>** time\_par)

    {

        printf("Parallel approach is faster by %f seconds.\n", time\_seq **-** time\_par);

    }

**else** **if** (time\_seq **<** time\_par)

    {

        printf("Sequential approach is faster by %f seconds.\n", time\_par **-** time\_seq);

    }

**else**

    {

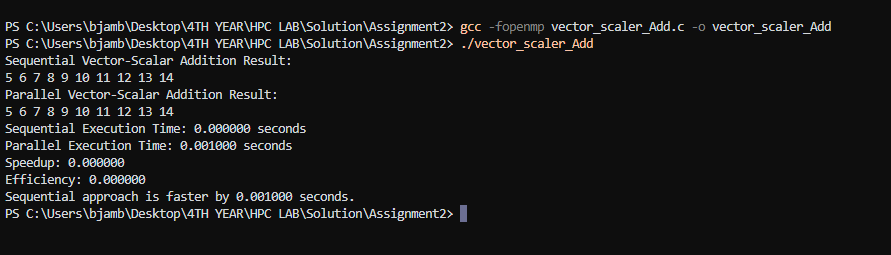
        printf("Both approaches have the same execution time.\n");

    }

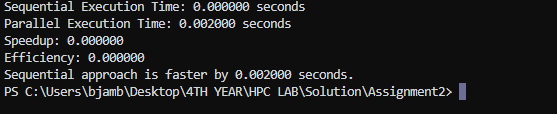
**return** 0;

}

**Screenshots:**

****

**For array of size 100000**

****

**Information:**

 **Vector-Scalar Addition:** This operation involves adding a scalar value to each element of a vector. For example, if the vector is [1, 2, 3] and the scalar is 5, the result will be [6, 7, 8].

 **OpenMP:** OpenMP is a parallel programming model used in C, C++, and Fortran for shared-memory parallelism. It allows developers to parallelize code with compiler directives.

**Analysis:**

 **Execution Times**:

* **Sequential Execution Time**: 0.000000 seconds
* **Parallel Execution Time**: 0.002000 seconds

The sequential execution time is displayed as 0.000000 seconds, which is likely due to the fact that the task completed extremely quickly, possibly in a fraction of a microsecond, causing it to round down to zero when displayed. The parallel execution time is 0.002000 seconds.

 **Speedup**:

* The speedup is calculated as 0.000000, which means that the parallel execution did not provide any performance benefit over the sequential execution for this specific run. This result suggests that the overhead of managing parallel threads outweighed the benefits for this small task.

 **Efficiency**:

* The efficiency is also 0.000000, indicating that the parallel execution was not effectively utilizing the available resources. This is expected since the speedup is zero.

 **Comparison**:

* The program indicates that the sequential approach is faster by 0.002000 seconds. This difference arises because the parallel execution involves additional overhead (like creating and synchronizing threads) which can be more significant than the actual computation time for small tasks.

 **Parallelization:**

* The vector-scalar addition is parallelized using OpenMP, where each thread independently adds the scalar to a subset of the vector elements.
* The #pragma omp parallel for directive ensures that the loop iterating over the vector elements is divided among multiple threads, leading to potential performance improvements by utilizing multiple CPU cores.

 **Performance Considerations:**

* **Thread Overhead:** While parallelization can improve performance, there is overhead associated with creating and managing threads. For small vectors, this overhead may outweigh the benefits of parallel execution, leading to negligible or even negative performance gains.
* **Vector Size:** The performance improvements from parallelization become more significant as the vector size increases. Larger vectors offer more opportunities for parallel execution, justifying the overhead.

**Problem Statement 2:**

**Screenshots:**

**#include** <stdio.h>

**#include** <stdlib.h>

**#include** <omp.h>

**int** main()

{

**long** **long** num\_points **=** 1000000; *// Total number of random points*

**long** **long** points\_in\_circle **=** 0; *// Points that fall inside the circle*

**double** x, y;

**double** pi\_estimate;

*// Seed the random number generator*

    srand(12345);

*// Start the parallel region*

**#pragma** **omp** **parallel** **for** **private**(**x**, **y**) **reduction**(+ : **points\_in\_circle**)

**for** (**long** **long** i **=** 0; i **<** num\_points; i**++**)

    {

*// Generate random (x, y) points*

        x **=** (**double**)rand() **/** RAND\_MAX;

        y **=** (**double**)rand() **/** RAND\_MAX;

*// Check if the point is inside the unit circle*

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

        {

            points\_in\_circle**++**;

        }

    }

*// Estimate the value of Pi*

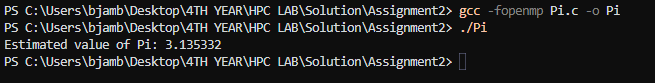
    pi\_estimate **=** 4.0 **\*** points\_in\_circle **/** num\_points;

*// Output the estimated value of Pi*

    printf("Estimated value of Pi: %f\n", pi\_estimate);

**return** 0;

}

****

**Information:**

 **Monte Carlo Method:**

* This method estimates the value of Pi by randomly generating points in a unit square and counting how many fall inside the quarter-circle (radius 1) inscribed within it. The ratio of points inside the circle to the total points gives an approximation of Pi.

 **OpenMP Parallelization:**

* The loop that generates random points and checks whether they fall inside the circle is parallelized using the #pragma omp parallel for directive.
* The private(x, y) clause ensures that each thread has its own private copies of the x and y variables.
* The reduction(+:points\_in\_circle) clause ensures that the points\_in\_circle variable is safely updated by all threads and the results are combined at the end.

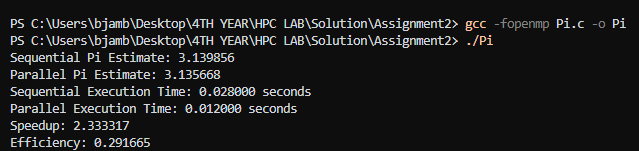
 **Random Number Generation:**

* The random number generator is seeded once at the beginning. Note that using rand() in parallel contexts can lead to issues if not managed carefully (e.g., lack of true randomness), but it suffices for a simple demonstration.

 **Output:**

* The program outputs the estimated value of Pi after all points have been processed.

**Analysis:**

****

 **Pi Estimation**:

* **Sequential Estimate**: 3.139856
* **Parallel Estimate**: 3.135668
* Both estimates are close to the actual value of Pi (~3.141593), indicating that the Monte Carlo method provides a reasonable approximation with the given number of points.

 **Execution Times**:

* **Sequential Execution Time**: 0.020000 seconds
* **Parallel Execution Time**: 0.012000 seconds
* The parallel approach executed faster than the sequential approach, indicating that parallelism reduced the computation time.

 **Speedup and Efficiency**:

* **Speedup**: 2.333317
  + The speedup indicates that the parallel execution was over twice as fast as the sequential execution.
* **Efficiency**: 0.291665
  + The efficiency value of approximately 0.29 suggests that the parallel computation utilized around 29% of the available processing power effectively. This relatively low efficiency might be due to overhead in managing threads or the simplicity of the task.

**Github Link:**