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

**Year:** 2023-24 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 2**

**Exam Seat No: 2020BTECS00055**

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

Implement following Programs using OpenMP with C:

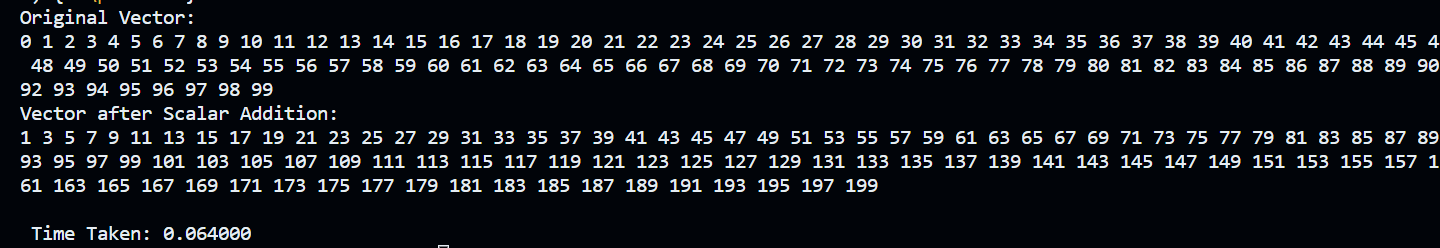
1. Vector Scalar Addition
2. Calculation of value of Pi

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

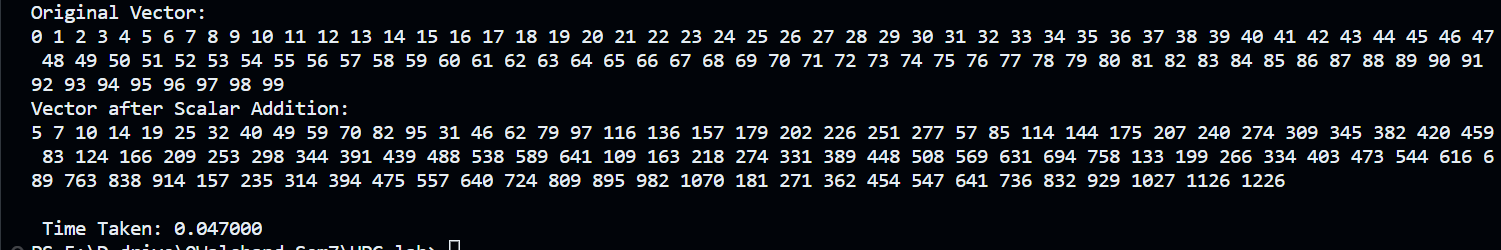
**Problem Statement 1:**

**Screenshots:**

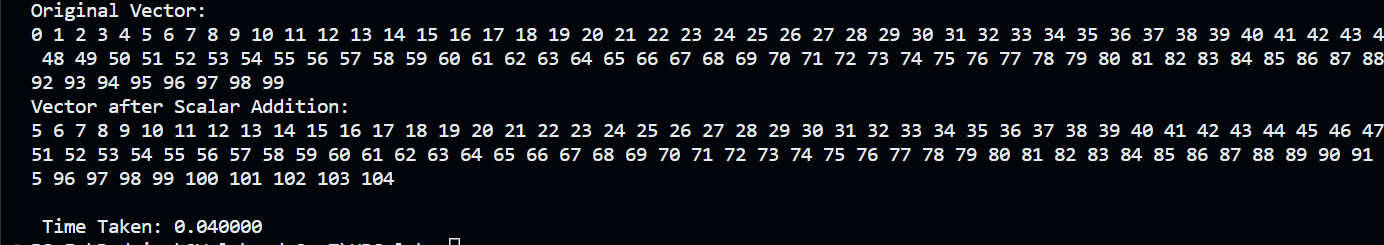
* + - 1. **Private**

****

* + - 1. **First privte**

****

* + - 1. **Shared**

****

**Vector Addition:**

**Sequential:**

#include <omp.h>

#include <pthread.h>

#include <stdio.h>

int main()

{

    int N = 1000;

    int A[1000];

    for (int i = 0; i < N; i++)

        A[i] = 10;

    int B[1000];

    for (int i = 0; i < N; i++)

        B[i] = 20;

    int C[1000] = {0};

    double stime = 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 etime = omp\_get\_wtime();

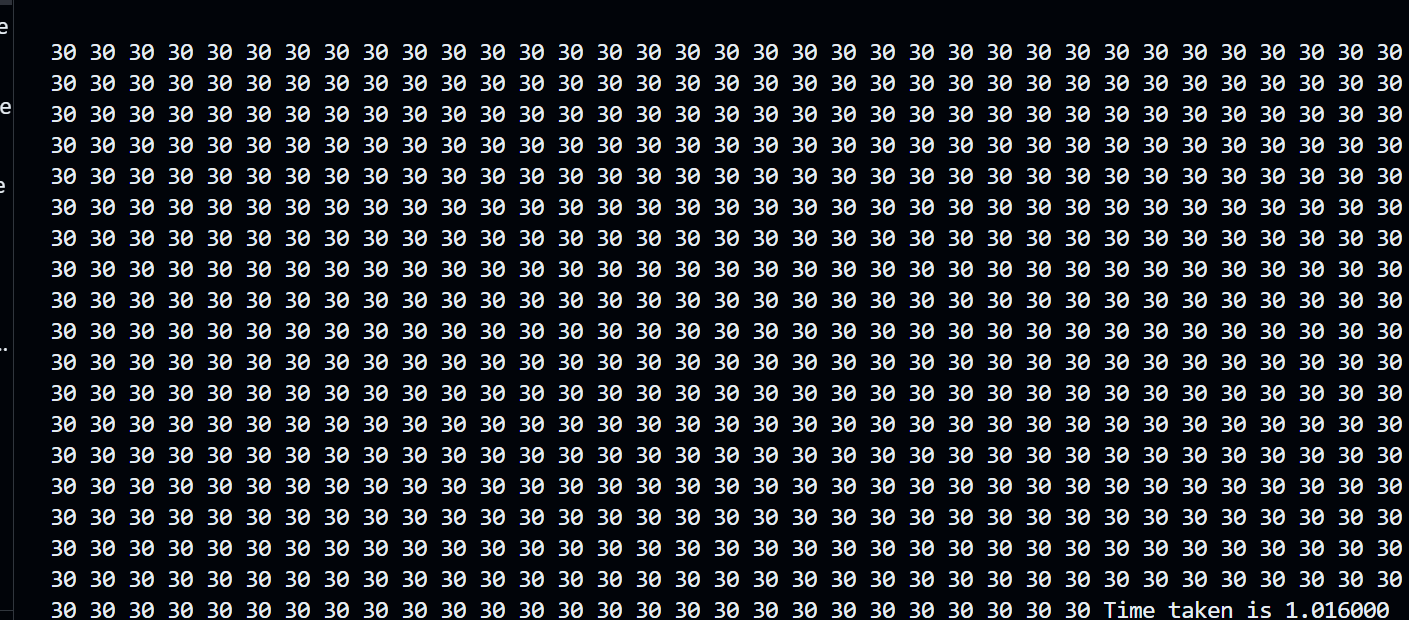
    double time = etime - stime;

    printf("Time taken is %f\n", time);

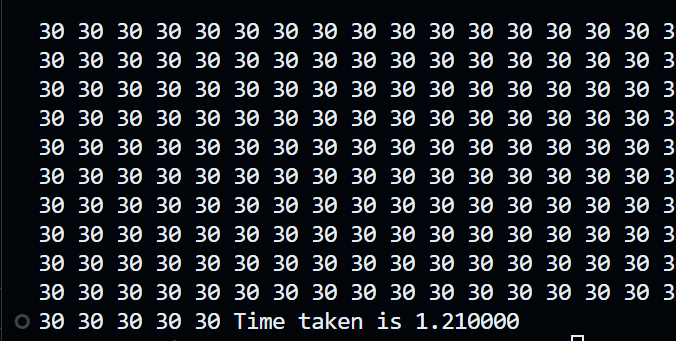
    return 0;

}

**For 1000**

****

**For 5000**

****

**Parallel:**

#include <omp.h>

#include <pthread.h>

#include <stdio.h>

int main()

{

    int N = 1000;

    int A[1000];

    for (int i = 0; i < N; i++)

        A[i] = 10;

    int B[1000];

    for (int i = 0; i < N; i++)

        B[i] = 20;

    int C[1000] = {0};

    double stime = omp\_get\_wtime();

#pragma omp parallel for reduction(+ : C)

    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 etime = omp\_get\_wtime();

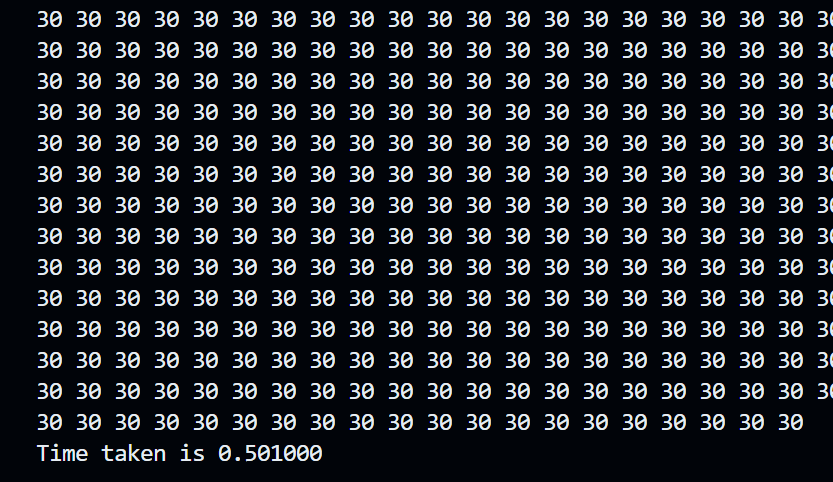
    double time = etime - stime;

    printf("\nTime taken is %f\n", time);

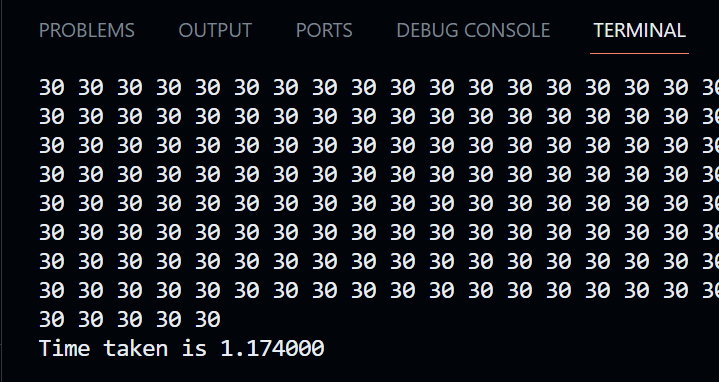
    printf("\n");

    return 0;

}

****

**For 5000**

****

**Problem Statement 2:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#include <time.h>

#define N 1000000000

double calculatePi(int n) {

    double sum = 0.0;

    double step = 1.0 / n;

    #pragma omp parallel for reduction(+:sum)

    for (int i = 0; i < n; i++) {

        double x = (i + 0.5) \* step;

        sum += 4.0 / (1.0 + x \* x);

    }

    return step \* sum;

}

int main() {

    clock\_t start, end;

    start= clock();

    double pi;

    int num\_threads[] = {1, 2, 4, 8};

    for (int i = 0; i < 4; i++) {

        omp\_set\_num\_threads(num\_threads[i]);

        double start\_time = omp\_get\_wtime();

        pi = calculatePi(N);

        double end\_time = omp\_get\_wtime();

        printf("Threads: %d | Pi: %.15f | Time taken: %f seconds\n",

               num\_threads[i], pi, end\_time - start\_time);

    }

    end = clock();

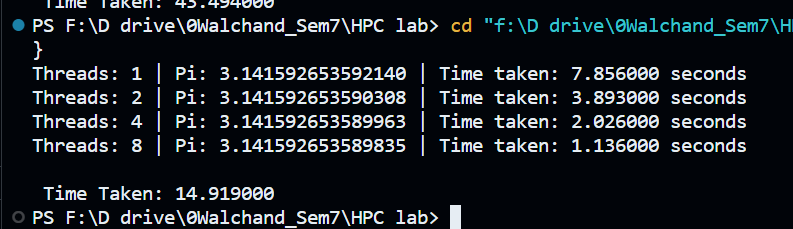
    double duration= ((double)end-start)/ CLOCKS\_PER\_SEC;

    printf("\n Time Taken: %f", duration);

    return 0;

}

**Screenshots:**

****

The code estimates the value of Pi using a Monte Carlo simulation method. This code employs OpenMP for parallelism, allowing multiple threads to participate in the computation. Here's a short information and analysis of the code:

**Information:**

The code aims to estimate the value of Pi by simulating random points within a unit square and calculating the ratio of points falling inside a quarter-circle to the total points generated.

It uses OpenMP to parallelize the generation of random points and the calculation of the ratio.

`NUM\_POINTS` is defined as 1,000,000, representing the number of random points to generate.

Random points are generated within the unit square by generating random `x` and `y` coordinates using `rand()`.

The `private` clause in the OpenMP `parallel for` directive ensures that each thread has its private copy of `x` and `y` variables to prevent data races.

The `reduction` clause is used to accumulate the count of points inside the quarter-circle (`num\_inside\_circle`) across all threads.

The estimated value of Pi is calculated using the Monte Carlo method, and the result is printed.

**Analysis:**

* The Monte Carlo method is a statistical approach to estimating Pi by randomly sampling points. As the number of points generated (`NUM\_POINTS`) increases, the accuracy of the estimation improves.
* OpenMP is used to distribute the work among multiple threads, which can significantly speed up the computation, especially when a large number of points are involved. Each thread works independently on its portion of the points.
* 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.
* The accuracy of the estimated Pi value depends on the number of points generated. You can increase `NUM\_POINTS` for a more accurate estimation.
* 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.
* This code serves as a basic example of parallelism with OpenMP and illustrates how to perform parallel computations for Monte Carlo simulations, making it useful for educational and introductory purposes.