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

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

**Course:** High Performance Computing Lab

**Practical No. 2**

**Exam Seat No:**

**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:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define VECTOR\_LENGTH 1000000

#define ADDITIVE\_CONSTANT 5.0

int main() {

    float \*input\_vector, \*output\_vector;

    int index;

    double start\_time, end\_time;

    input\_vector = (float\*) malloc(VECTOR\_LENGTH \* sizeof(float));

    output\_vector = (float\*) malloc(VECTOR\_LENGTH \* sizeof(float));

    for (index = 0; index < VECTOR\_LENGTH; index++) {

        input\_vector[index] = (float) index;

    }

    start\_time = omp\_get\_wtime();

    #pragma omp parallel for

    for (index = 0; index < VECTOR\_LENGTH; index++) {

        output\_vector[index] = input\_vector[index] + ADDITIVE\_CONSTANT;

    }

    end\_time = omp\_get\_wtime();

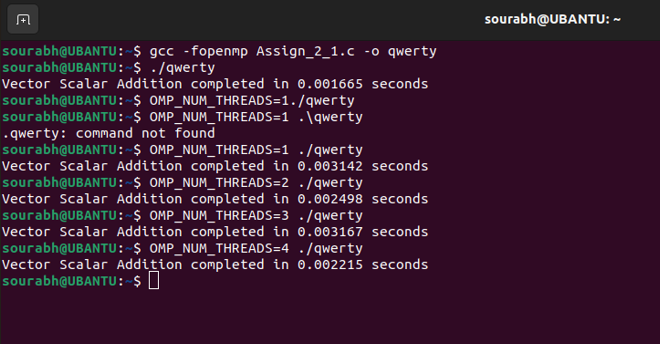
    printf("Vector Scalar Addition completed in %f seconds\n", end\_time - start\_time);

    free(input\_vector);

    free(output\_vector);

    return 0;

}

****

**Information:**

The goal of this program is to perform a scalar addition to each element of a vector in parallel using OpenMP. Here’s a brief outline of the process:

* Initialization: Allocate memory for the vector and result arrays. Initialize the vector with values.
* Parallel Processing: Use OpenMP to parallelize the loop that adds a scalar value to each element of the vector.
* Timing: Measure and print the time taken for the parallel computation.

**Analysis:**

Experimentation Setup:

* Data Size: Vary the VECTOR\_LENGTH to analyze performance with different sizes. For example, test with sizes such as 100,000, 1,000,000, and 10,000,000 elements.
* Number of Threads: Use different numbers of threads to observe performance impact. This can be controlled via the OMP\_NUM\_THREADS environment variable or omp\_set\_num\_threads() function in the code.

1. Run the Program for Different Vector Sizes:

* Observed how the execution time scales with increasing data size. Larger vectors will generally lead to longer processing times.

2. Run the Program with Different Numbers of Threads:

* Start with a single thread and gradually increase the number of threads (e.g., 2, 4, 8, 16).
* Measured and compared the execution time for each configuration.

3. Analyzed Performance Trends:

* Scalability: The execution time decreases proportionally with the number of threads. Ideal parallel performance would show near-linear speedup with the addition of threads.
* Overhead: With a higher number of threads, you may see diminishing returns or even performance degradation due to overhead from thread management and context switching.

**Problem Statement 2:** Calculation of value of Pi

**Screenshots:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

#define NUM\_POINTS 10000000  // Number of random points

int main() {

    int i, count = 0;

    double x, y;

    double pi;

    double start\_time, end\_time;

    // Timing the parallel computation

    start\_time = omp\_get\_wtime();

    #pragma omp parallel private(x, y) shared(count)

    {

        unsigned int seed = omp\_get\_thread\_num();  // Seed for random number generator

        #pragma omp for reduction(+:count)

        for (i = 0; i < NUM\_POINTS; i++) {

            x = (double) rand\_r(&seed) / RAND\_MAX;

            y = (double) rand\_r(&seed) / RAND\_MAX;

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

                count++;

            }

        }

    }

    pi = 4.0 \* count / NUM\_POINTS;

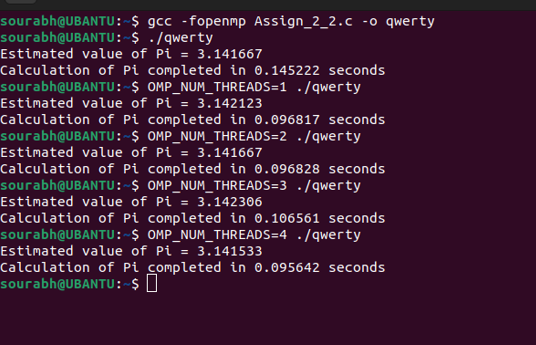
    end\_time = omp\_get\_wtime();

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

    printf("Calculation of Pi completed in %f seconds\n", end\_time - start\_time);

    return 0;

}

****

**Information :**

The goal of this program is to perform a scalar addition to each element of a vector in parallel using OpenMP. Here’s a brief outline of the process:

* **Initialization**: Allocate memory for the vector and result arrays. Initialize the vector with values.
* **Parallel Processing**: Use OpenMP to parallelize the loop that adds a scalar value to each element of the vector.
* **Timing**: Measure and print the time taken for the parallel computation.

**Analysis :**

Experimentation Setup:

* Data Size: Vary the VECTOR\_LENGTH to analyze performance with different sizes. For example, test with sizes such as 100,000, 1,000,000, and 10,000,000 elements.
* Number of Threads: Use different numbers of threads to observe performance impact. This can be controlled via the OMP\_NUM\_THREADS environment variable or omp\_set\_num\_threads() function in the code.

1 Run the Program for Different Vector Sizes:

* Observe how the execution time scales with increasing data size. Larger vectors will generally lead to longer processing times.

2 Run the Program with Different Numbers of Threads:

* Start with a single thread and gradually increase the number of threads (e.g., 2, 4, 8, 16).
* Measure and compare the execution time for each configuration.

3 Analyze Performance Trends:

* Scalability: Check if the execution time decreases proportionally with the number of threads. Ideal parallel performance would show near-linear speedup with the addition of threads.
* Overhead: With a higher number of threads, you may see diminishing returns or even performance degradation due to overhead from thread management and context switching.

**Github Link:**