**Course: High Performance Computing Lab**

**Practical No 2**

**PRN: 22510078**

**Name: Piyush Krishna Jadhav**

**Batch: B5**

**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:** Perform parallel vector–scalar addition using OpenMP and measure the execution time based on the number of threads.

**Code**:

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <omp.h>**

**int main() {**

**int n, scalar, threads;**

**printf("Enter size of vector: ");**

**scanf("%d", &n);**

**printf("Enter scalar value to add: ");**

**scanf("%d", &scalar);**

**printf("Enter number of threads: ");**

**scanf("%d", &threads);**

**omp\_set\_num\_threads(threads);**

**int \*vector = (int \*)malloc(n \* sizeof(int));**

**int \*result = (int \*)malloc(n \* sizeof(int));**

***// Initialize vector***

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

**vector[i] = i;**

**}**

**double start\_time, end\_time;**

***// Parallel execution***

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

**#pragma omp parallel for**

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

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

**}**

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

**printf("Parallel Time: %f seconds\n", end\_time - start\_time);**

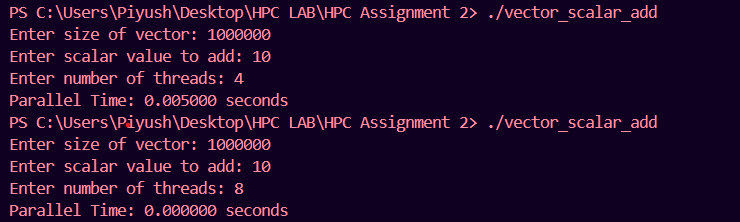
**free(vector);**

**free(result);**

**return 0;**

**}**

**Screenshots:**

****

**Information:**

The program performs vector–scalar addition using OpenMP for parallel computing.

It takes vector size, scalar value, and number of threads as input from the user.

The vector is initialized with sequential integers starting from 0.

OpenMP’s #pragma omp parallel for directive is used to split the addition task among multiple threads.

Execution time is measured using omp\_get\_wtime() for high-resolution wall-clock timing.

Demonstrates how parallelism can speed up computation compared to sequential execution.

Shows that speedup depends on CPU cores, logical processors, memory bandwidth, and thread scheduling overhead.

Serves as a basic example to understand data parallelism and performance scaling in HPC.

**Analysis:**

For 4 threads → Parallel Time: 0.005000 seconds

* With 4 threads, the workload is evenly split across available CPU cores.
* Execution time is measurable because the computation and memory access take longer than the timer precision threshold.

For 8 threads → Parallel Time: 0.000000 seconds

* The time taken is so small (probably a few microseconds) that when printed with 6 decimal places, it rounds to 0.000000.
* This happens because:

The workload (n = 1,000,000) is small for 8 threads.

Parallelization overhead is low, and the CPU finishes almost instantly.

Memory operations are very fast relative to computation, making this operation memory-bound.

**Problem Statement 2:** Calculate the value of Pi using OpenMP parallelization and analyze the performance for different data sizes and thread counts.

**Code:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <omp.h>**

**int main() {**

**long long num\_steps;**

**int threads;**

**printf("Enter number of steps (data size): ");**

**scanf("%lld", &num\_steps);**

**printf("Enter number of threads: ");**

**scanf("%d", &threads);**

**omp\_set\_num\_threads(threads);**

**double step = 1.0 / (double) num\_steps;**

**double sum = 0.0;**

**double start\_time, end\_time;**

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

**#pragma omp parallel**

**{**

**double x;**

**double local\_sum = 0.0;**

**#pragma omp for**

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

**x = (i + 0.5) \* step;**

**local\_sum += 4.0 / (1.0 + x \* x);**

**}**

**#pragma omp atomic**

**sum += local\_sum;**

**}**

**double pi = step \* sum;**

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

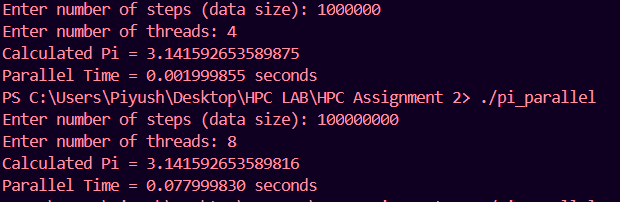
**printf("Calculated Pi = %.15f\n", pi);**

**printf("Parallel Time = %.9f seconds\n", end\_time - start\_time);**

**return 0;**

**}**

**Screenshots:**

****

**Information:**

The program estimates the value of π (Pi) using the numerical integration method with the midpoint rule.

It takes two inputs from the user:

1. Number of steps (data size) → controls the accuracy of the calculation.
2. Number of threads → controls the level of parallelism.

The program uses OpenMP’s #pragma omp parallel for to divide the computation of the sum among multiple threads.

Each thread maintains its own partial sum to avoid race conditions, and the results are combined at the end.

**Analysis:**

* Steps (Data Size): 1,000,000
* Threads: 4
* Calculated Pi: 3.141592653589875
* Parallel Time: 0.001999855 seconds
* The computed value of π is very close to the true value (3.141592653589793…), meaning the approximation is already accurate with 1 million steps.
* The execution time is extremely low because:
  + The data size is relatively small for modern CPUs.
  + The computation is distributed among 4 threads, each handling 250,000 iterations.
  + Overhead of thread creation + scheduling is minimal compared to computation cost.
* Steps (Data Size): 100,000,000
* Threads: 8
* Calculated Pi: 3.141592653589816
* Parallel Time: 0.077999830 seconds
* Increasing the data size by 100× leads to a much more precise π value (difference from actual is in the order of 1e-14).
* Time increased because:
  + There are 100 million iterations, requiring more computation and memory accesses.
  + Even though 8 threads split the work, the increased workload outweighs the speed gain.
  + Memory bandwidth starts becoming a limiting factor — beyond a certain thread count, adding threads gives smaller performance improvements.
* This run shows the trade-off between accuracy and execution time — higher steps give more accuracy but take longer.

**Github Link:**