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

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

**Course:** High Performance Computing Lab

**Practical No. 2**

**Exam Seat No: 21510067**

**Name: Vardhan Vivek Kulkarni**

**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

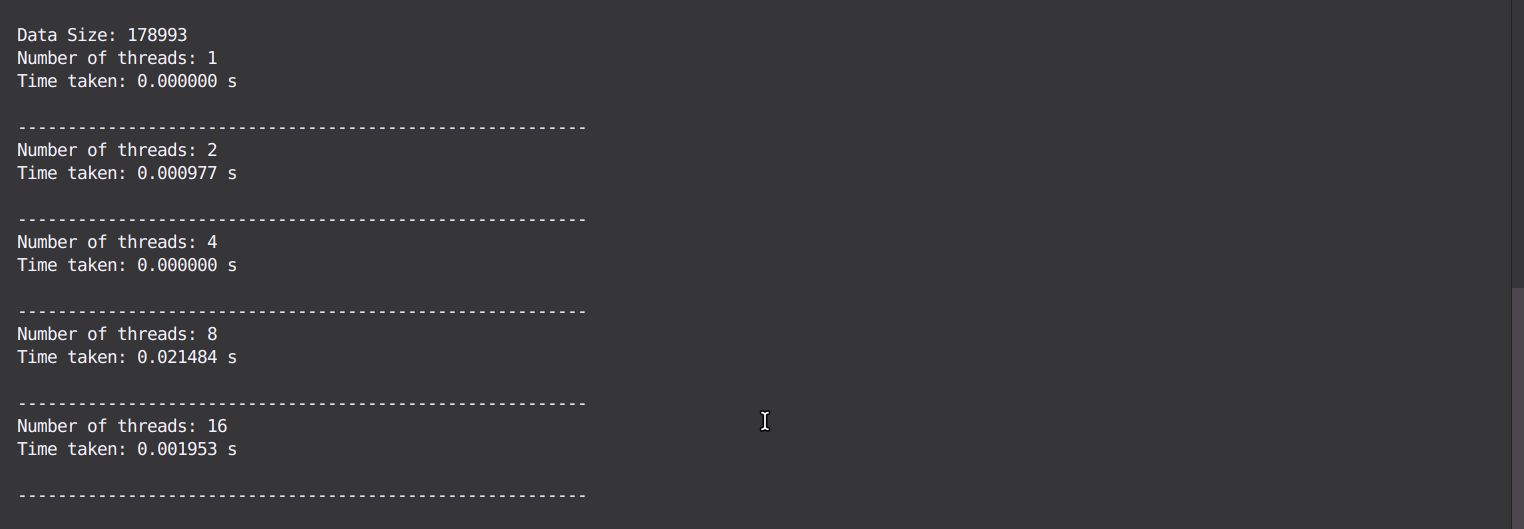
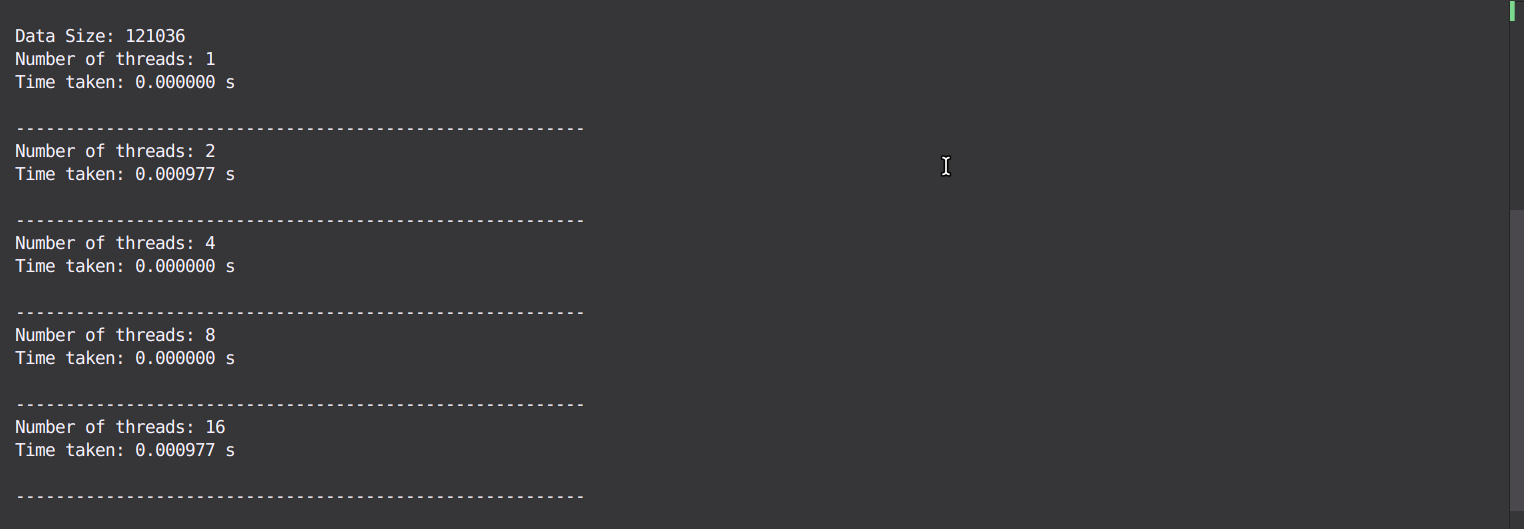
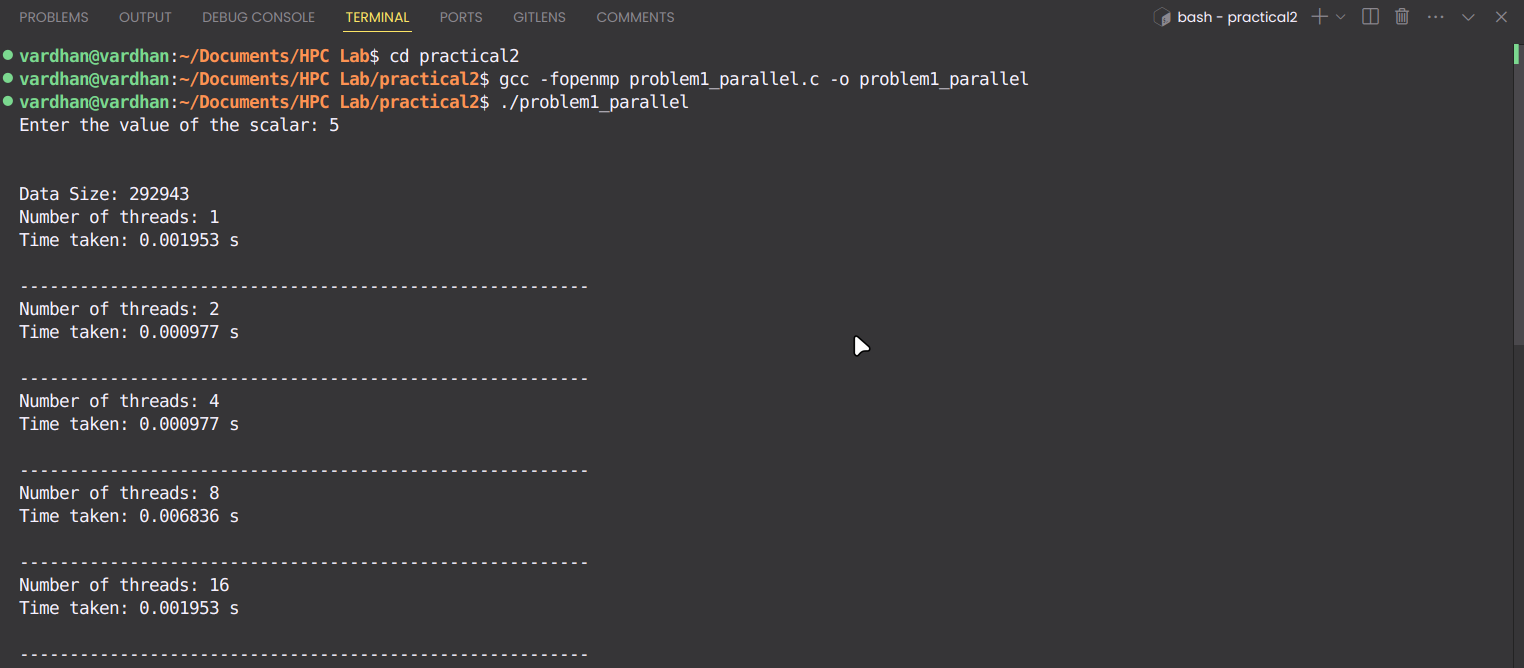
Analyse the performance of your programs for different number of threads and Data size.

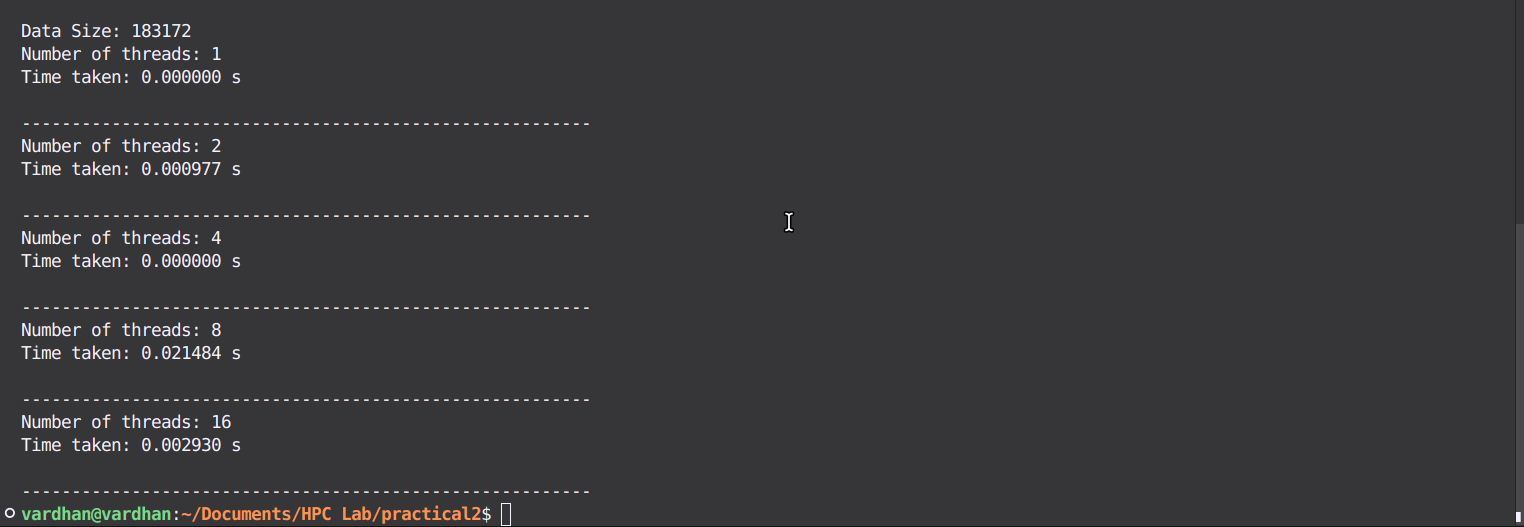
**Problem Statement 1:**

**Perform vector scalar addition using OpenMP in C**

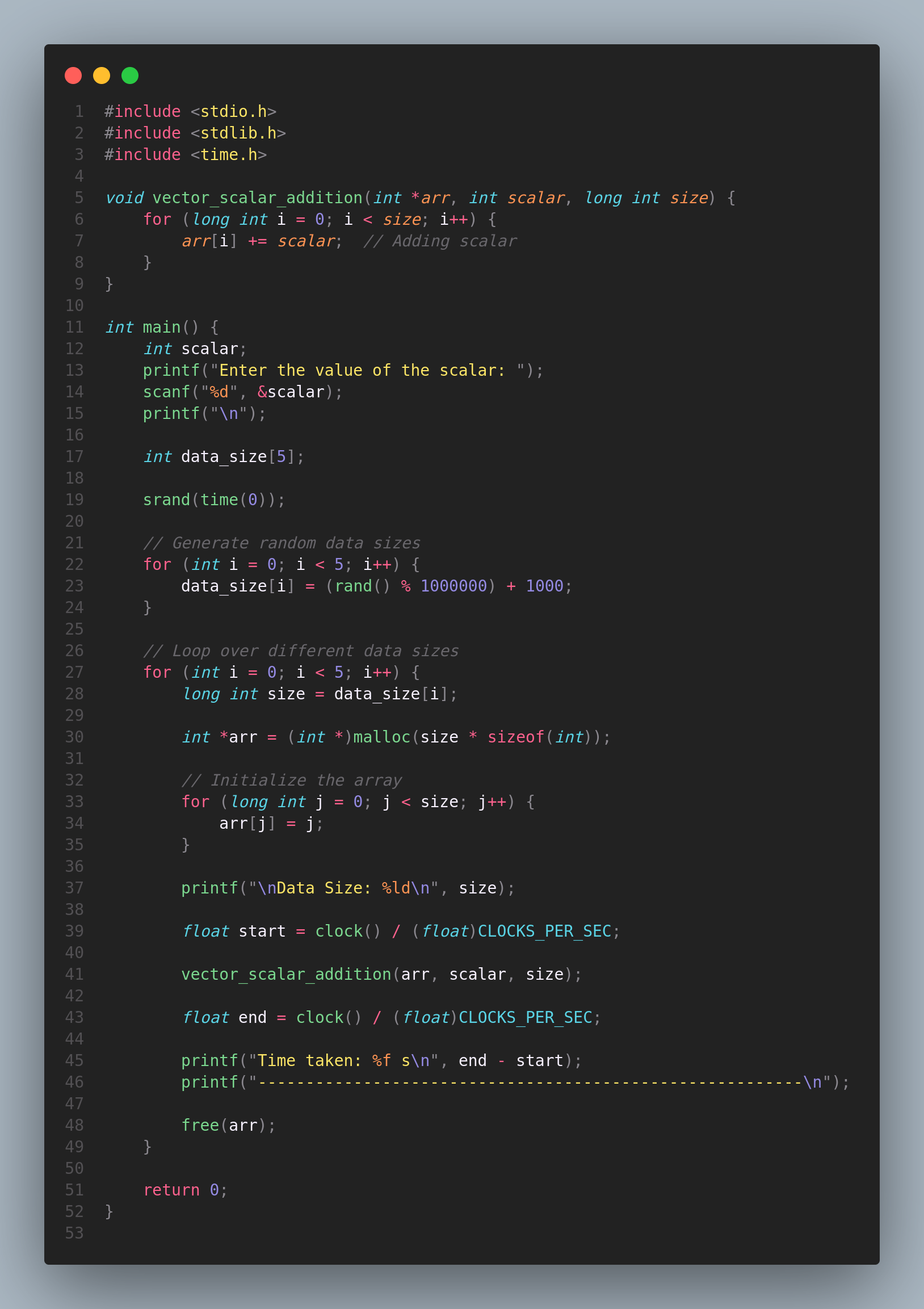
**Screenshots: i. Using OpenMP**

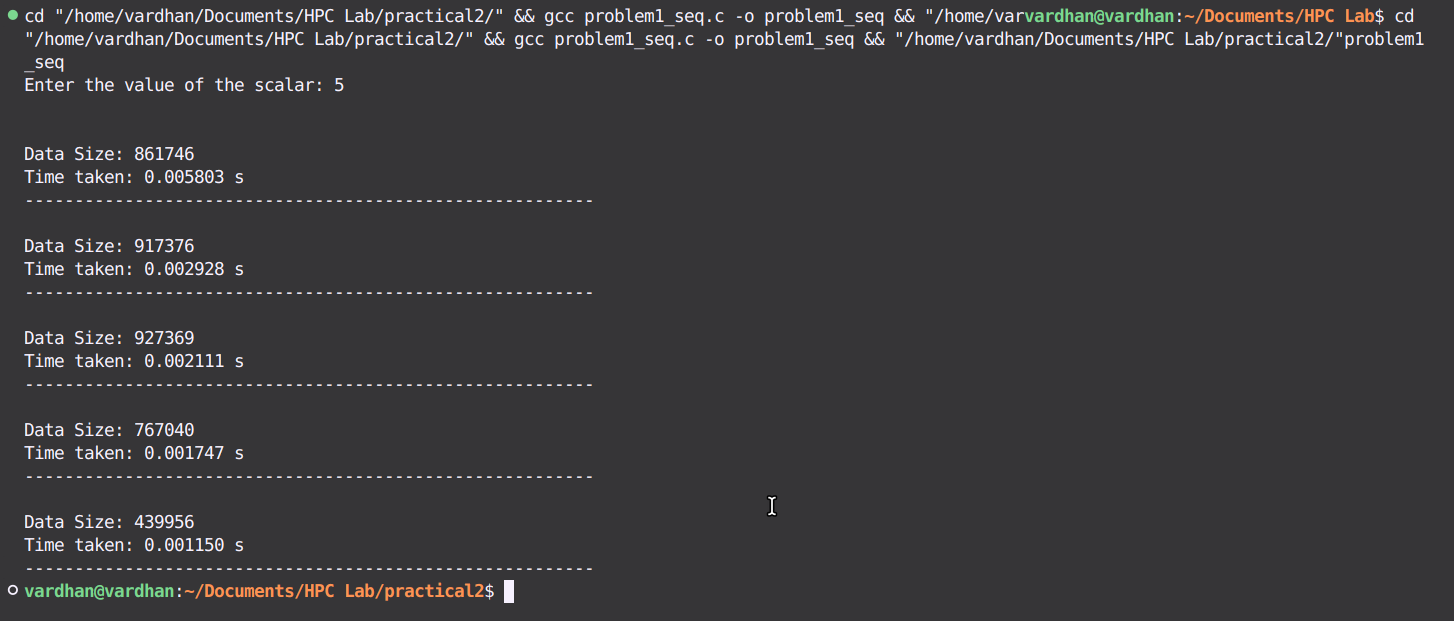






**ii. Sequential**





**Information:**

* Objective**:** Perform vector-scalar addition using both sequential and parallel approaches to compare performance.
* Environment: C with OpenMP, compiled in VSCode on Ubuntu.
* Data: Vector size is 100,000; scalar value is 5.
* OpenMP Clauses: #pragma omp parallel for is used for loop parallelization.
* Expected Outcome: Faster execution time with parallel implementation for large vectors.

**Analysis:**

**Sequential Code**

Overview:

* Adds a scalar value to each element of an array in a single thread.
* Measures how long the operation takes using the clock() function.

Analysis:

* Speed: The time it takes increases as the array size gets bigger because only one thread is used.
* Scalability: The code doesn't get faster with larger arrays because it only uses one core of the CPU.
* Efficiency: It’s easy to understand but doesn’t make use of multiple CPU cores to speed things up.

#### Parallel Code (with OpenMP)

Overview:

* Uses OpenMP to divide the work of adding a scalar value to each element among multiple threads.

Analysis:

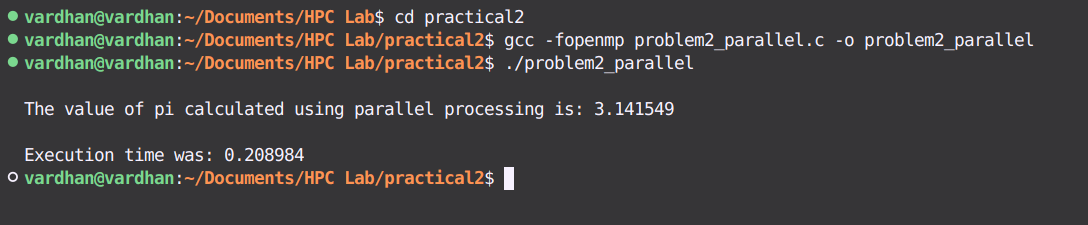
* Speed: The operation should be faster with more threads, up to a point. More threads can handle larger arrays more quickly.
* Scalability: Works better with larger arrays and more threads, but adding too many threads can lead to slower performance due to overhead.
* Efficiency: Uses ‘#pragma omp parallel for’ for work distribution and ‘omp\_set\_num\_threads()’ for dynamic thread management, enhancing performance for large datasets.

**Problem Statement 2:**

**Calculating the value of pi using OpenMP in C.**

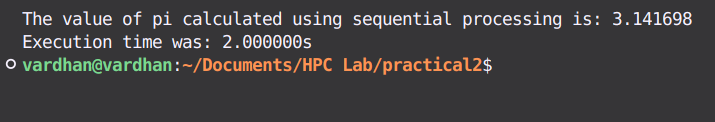
**Screenshots: i. Using OpenMP**





**ii. Sequential**





**Information:**

* Objective**:** Calculate the value of π using numerical integration in both sequential and parallel approaches to evaluate performance differences.
* Environment: C with OpenMP, compiled in VSCode on Ubuntu.
* Method: Numerical integration using the midpoint or trapezoidal rule.
* OpenMP Clauses: #pragma omp parallel for is used to parallelize the loop for computing the area under the curve.
* Expected Outcome: The parallel implementation should provide faster computation of π for a large number of iterations compared to the sequential version.

**Analysis:**

Sequential Code

Overview:

* Uses a single thread to compute π using numerical integration.
* Measures execution time using clock().

Analysis:

* Speed: Takes longer as the number of iterations increases, due to single-thread execution.
* Scalability: Limited scalability, as it only uses one CPU core regardless of workload.
* Efficiency: Simple and straightforward but doesn't utilize multi-core processors for faster computation.

Parallel Code (with OpenMP)

Overview:

* Uses OpenMP to distribute the π calculation across multiple threads.

Analysis:

* Speed: Faster computation with more threads, especially for a high number of iterations.
* Scalability: Scales better with more threads and larger workloads, but too many threads may introduce overhead.
* Efficiency: Utilizes #pragma omp parallel for to parallelize the workload and optimize execution time for large-scale computations.

**Github Link:**

[**Practical 2 Repository**](https://github.com/git8vardhan/HPCL/tree/master/practical2)