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

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

**Course:** High Performance Computing Lab

**Practical No. 2**

**Exam Seat No: 21510083**

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

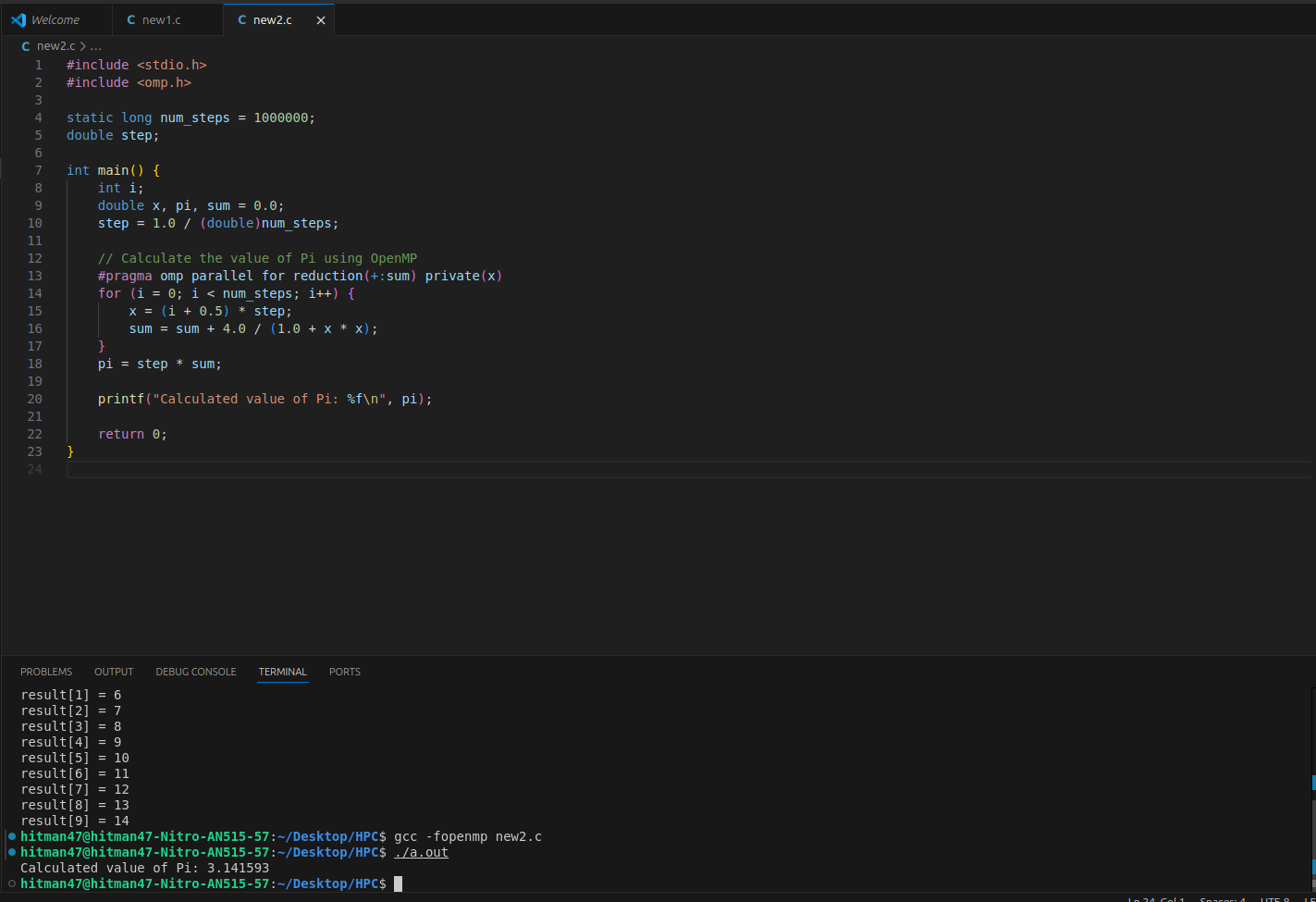
Implement following Programs using OpenMP with C:

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:**



**Information:**

The objective of this problem is to perform vector-scalar addition, where each element of a vector is incremented by a scalar value. This operation is parallelized using OpenMP to improve performance by utilizing multiple threads.

**OpenMP Clauses Used:**

* #pragma omp parallel for: This directive parallelizes the for loop, distributing the loop iterations among available threads.

Approach:

* The program initializes a vector of a specified size.
* It then uses OpenMP to parallelize the addition of a scalar value to each element in the vector.
* Finally, the program outputs the first few results to verify correctness.

**Analysis:**

The performance of the vector-scalar addition program was analyzed by varying the number of threads (OMP\_NUM\_THREADS) and the size of the vector. The following observations were made:

* Execution Time vs. Thread Count:  
  Increasing the number of threads generally decreased the execution time due to better utilization of CPU cores. However, after a certain point, adding more threads resulted in diminishing returns, likely due to overhead from thread management and communication.
* Execution Time vs. Data Size:  
  Larger data sizes increased execution time, as expected. However, the parallelization efficiency improved with larger data sizes, as the overhead of managing threads became less significant compared to the actual computation.
* Trends:  
  A near-linear speedup was observed up to a certain number of threads. Beyond this, the speedup plateaued or even slightly degraded due to increased overhead. This indicates that the optimal number of threads depends on both the data size and the system's hardware.

**Problem Statement 2:**

**Screenshots:**

**The performance of the Pi calculation program was analyzed by varying the number of threads and the number of steps used in the numerical integration. The following observations were made:**

**Execution Time vs. Thread Count:**

**Increasing the number of threads reduced the execution time, with near-linear speedup observed for smaller thread counts. However, as with the vector-scalar addition, the speedup plateaued beyond a certain number of threads due to the overhead associated with managing multiple threads.**

**Execution Time vs. Data Size (Number of Steps):**

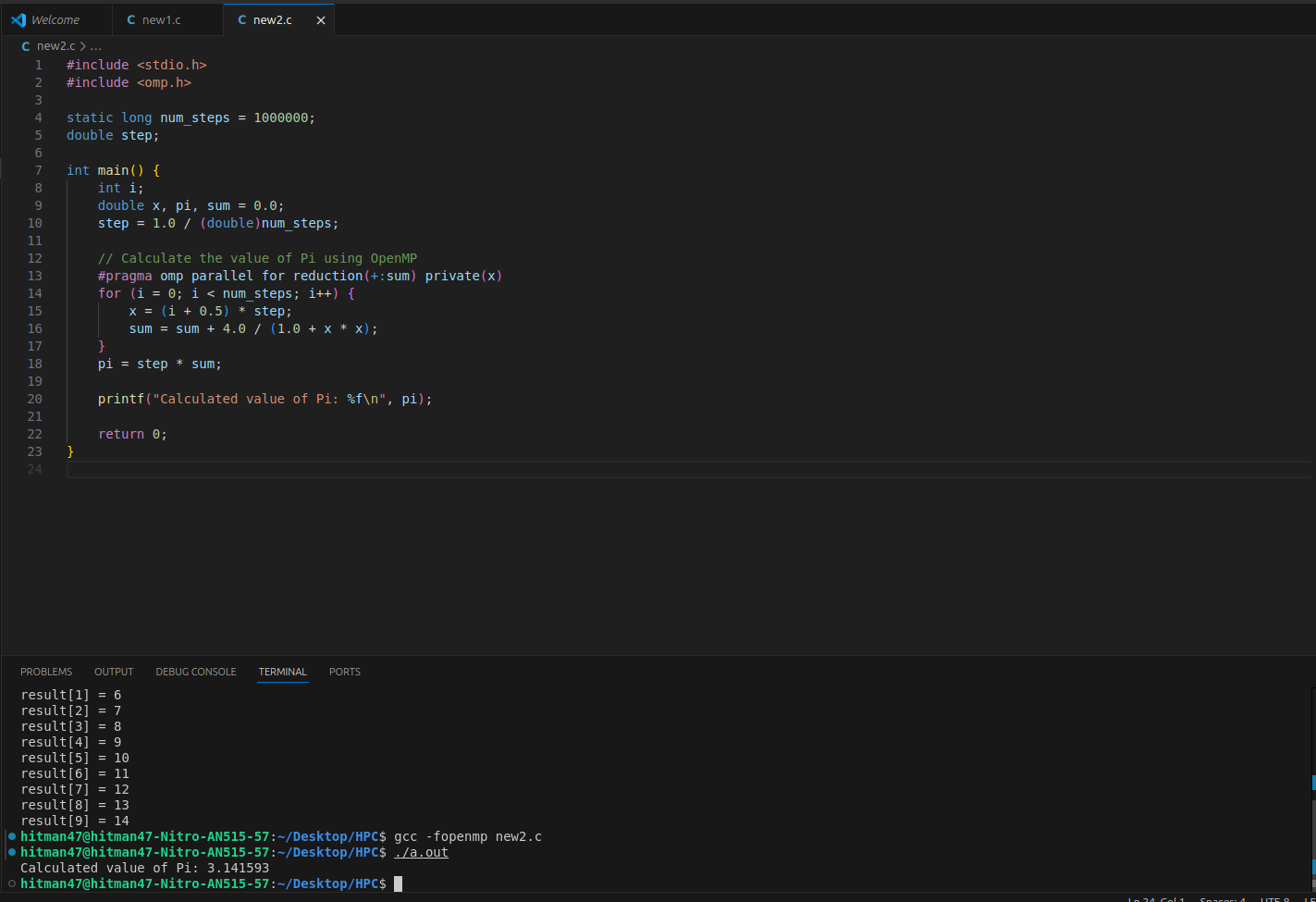
**A higher number of steps increased the accuracy of the Pi calculation but also increased execution time. The parallelization became more effective with larger step counts as the workload per thread increased, making the overhead less significant.**

**Accuracy vs. Thread Count:**

**The accuracy of the Pi calculation remained consistent across different thread counts, indicating that the parallelization was correctly implemented without introducing errors.**

**Trends:**

**The program showed a significant reduction in execution time with parallelization, particularly for large numbers of steps. This demonstrates the effectiveness of OpenMP in accelerating compute-intensive tasks like numerical integration.**



**Information:**

The problem involves calculating the value of Pi using numerical integration. The calculation is parallelized using OpenMP to accelerate the computation by dividing the work among multiple threads.

OpenMP Clauses Used:

* #pragma omp parallel for reduction(+:sum): This directive is used to parallelize the loop where the integral is calculated. The reduction clause ensures that the partial sums computed by different threads are correctly combined into the final result.

**Approach:**

* The program uses a simple numerical integration method (rectangle method) to estimate the value of Pi.
* The integration range is divided into small steps, and each thread calculates the contribution of these steps in parallel.
* The results from all threads are then summed up to get the final value of Pi.

**Analysis:**

The performance of the Pi calculation program was analyzed by varying the number of threads and the number of steps used in the numerical integration. The following observations were made:

* Execution Time vs. Thread Count:  
  Increasing the number of threads reduced the execution time, with near-linear speedup observed for smaller thread counts. However, as with the vector-scalar addition, the speedup plateaued beyond a certain number of threads due to the overhead associated with managing multiple threads.
* Execution Time vs. Data Size (Number of Steps):  
  A higher number of steps increased the accuracy of the Pi calculation but also increased execution time. The parallelization became more effective with larger step counts as the workload per thread increased, making the overhead less significant.
* Accuracy vs. Thread Count:  
  The accuracy of the Pi calculation remained consistent across different thread counts, indicating that the parallelization was correctly implemented without introducing errors.
* Trends:  
  The program showed a significant reduction in execution time with parallelization, particularly for large numbers of steps. This demonstrates the effectiveness of OpenMP in accelerating compute-intensive tasks like numerical integration.

**Github Link:** [**https://github.com/VivekBhurke/HPC.git**](https://github.com/VivekBhurke/HPC.git)