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

**Year:** 2025-26 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 4**

**Exam Seat No:**

**Title of practical:**

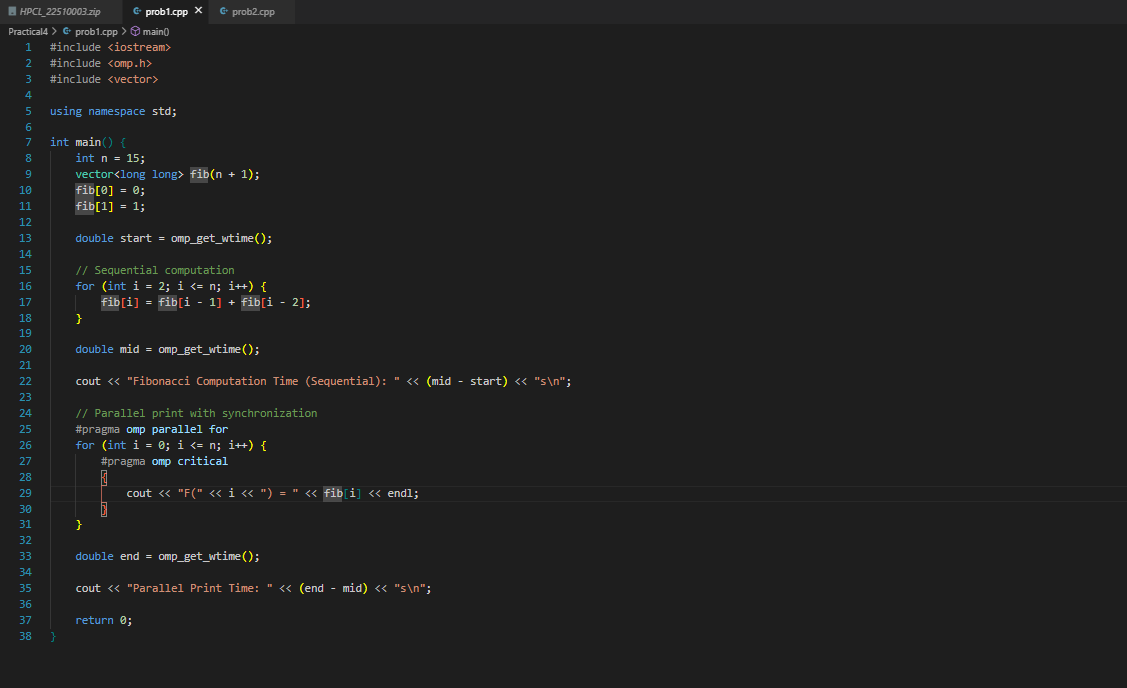
Study and Implementation of Synchronization

**Problem Statement 1:**

# Analyze and implement a Parallel code for below programs using OpenMP considering synchronization requirements. (Demonstrate the use of different clauses and constructs wherever applicable)

# Fibonacci Computation:

**Code:**

****

**Output:**

**A computer screen shot of a program

AI-generated content may be incorrect.**

**Information:**

We can not parallelize the computation of the Fibonacci series but we can use the parallelism for printing the values of fibonacci series.

1. **Sequential Dependency:** Fibonacci numbers depend on previous terms, making the main computation inherently sequential and unsuitable for simple parallel loops.
2. **Race Conditions:** Parallelizing the loop without synchronization causes race conditions due to simultaneous updates of shared variables, leading to incorrect results.
3. **Synchronization Use:** Employ OpenMP constructs like critical to safely manage shared resources and prevent conflicts during parallel execution.
4. **Clause Selection:** Choose parallel clauses based on task independence and workload balance avoid parallelizing dependent computations and focus on sections or tasks for concurrency.

**Problem Statement 2:**

# Analyze and implement a Parallel code for below programs using OpenMP considering synchronization requirements. (Demonstrate the use of different clauses and constructs wherever applicable)

## Producer Consumer Problem

**Code:**

****

**Output:**

**A screenshot of a computer

AI-generated content may be incorrect.**

**Observation:**

* The producer fills the buffer up to its maximum size, after which the consumer starts consuming items; this cycle repeats while maintaining **FIFO order**.
* The buffer count never exceeds its capacity or drops below zero, proving correct synchronization and safe parallel execution.
* #pragma omp parallel sections is used to divide work between two independent tasks (producer and consumer), each handled by a separate thread.
* #pragma omp critical ensures mutual exclusion so that shared variables (buffer, countItems, in, out) are updated safely without race conditions.
* A circular buffer design with busy-waiting loops is used to simulate waiting when the buffer is full/empty, ensuring correct coordination between producer and consumer.