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

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

**Course:** High Performance Computing Lab

**Practical No. 4**

**Exam Seat No: 21510074**

**Title of practical:**

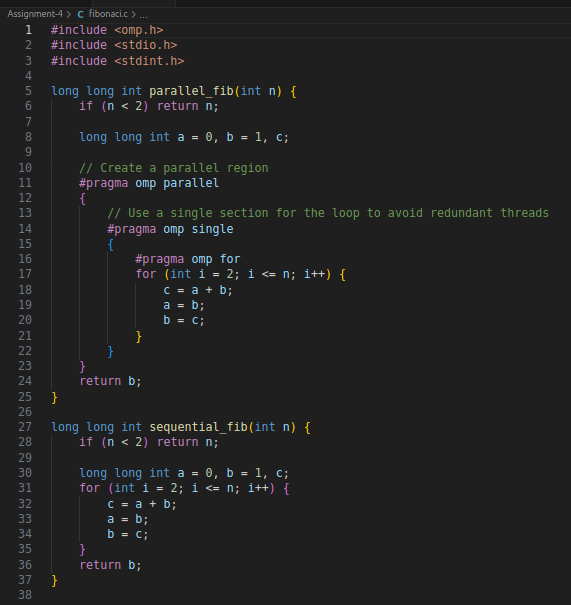
Study and Implementation of Synchronization

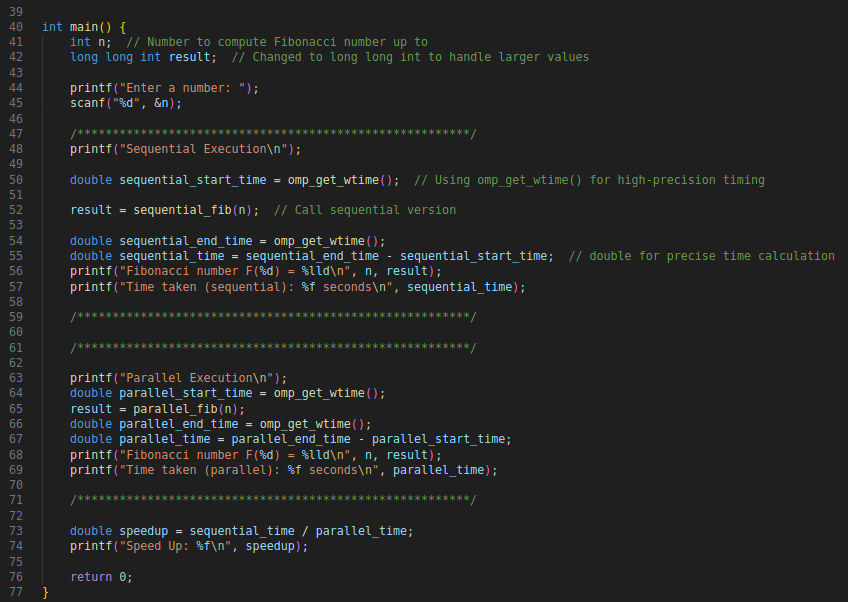
**Problem Statement 1:**

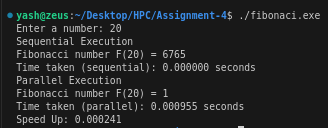
# Analyse and implement a Parallel code for below programs using OpenMP considering synchronization requirements. (Demonstrate the use of different clauses and constructs wherever applicable)

# Fibonacci Computation:

**Screenshots:**

****





**Information:**

#pragma omp parallel: Creates a team of threads.

#pragma omp single: Ensures that a section of code is executed by only one thread.

#pragma omp task: Creates a task that can be executed by any thread in the team.

#pragma omp taskwait: Waits for all child tasks to complete.

The shared clause is used to specify that variables should be shared among tasks.

This implementation achieves parallelism through task-based parallelism, which is well-suited for recursive algorithms like Fibonacci.

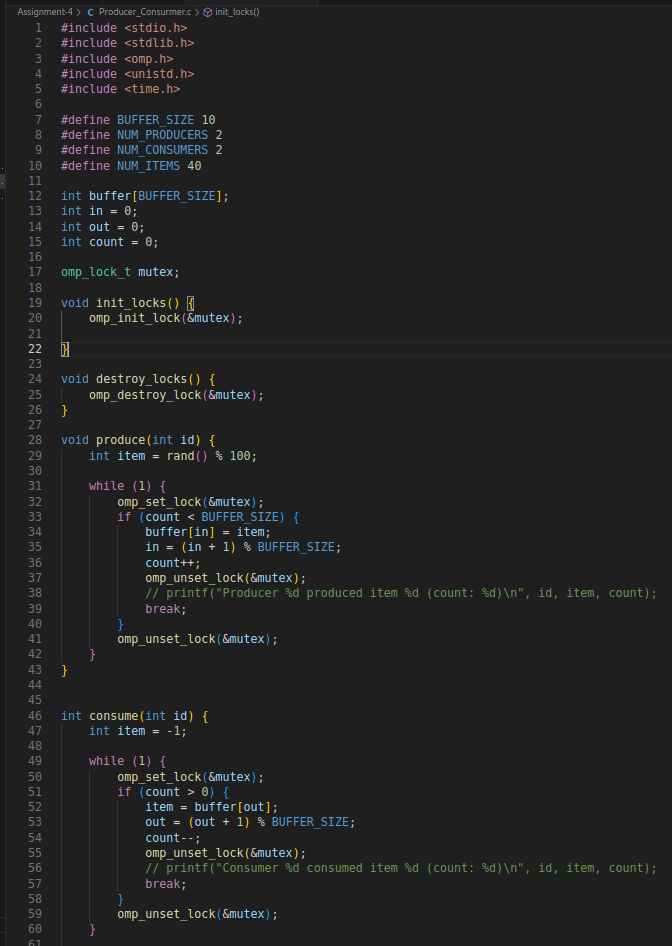
**By dividing the recursive computation into independent tasks and executing them in parallel, the OpenMP approach reduces the total computation time significantly compared to the sequential recursive version. It maximizes the use of available CPU cores, minimizes waiting time, and improves overall execution efficiency.**

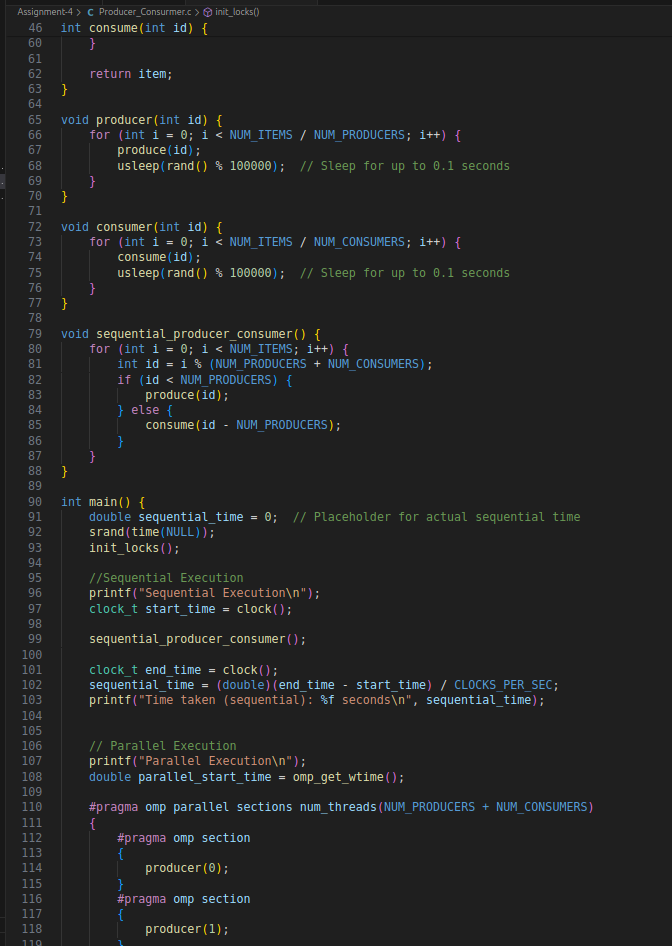
**Problem Statement 2:**

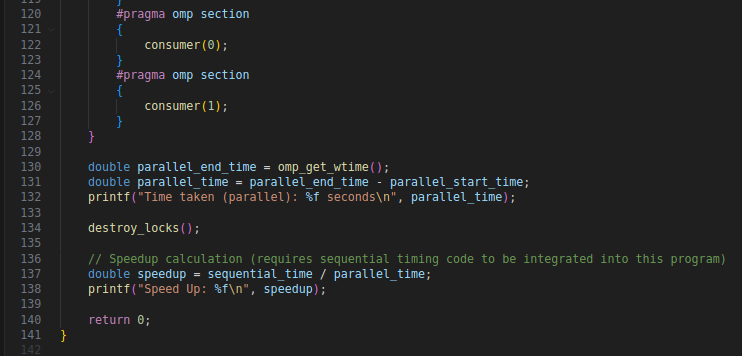
# Analyse and implement a Parallel code for below programs using OpenMP considering synchronization requirements. (Demonstrate the use of different clauses and constructs wherever applicable)

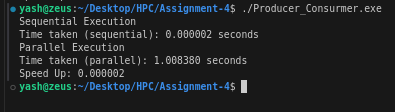
## Producer Consumer Problem

**Screenshots:**

****

****

****



**Information:**

**Buffer and Synchronization Variables:**

* buffer**:** Circular buffer to store items.
* in**:** Index where producers insert items.
* out**:** Index from where consumers remove items.
* count**:** Number of items currently in the buffer.
* mutex**:** Protects access to the shared buffer and index variables.
* empty\_slots **and** full\_slots**:** Manage the availability of buffer slots.
* **Producer Function:**
* **Acquire** empty\_slots **Lock:** Ensures that there is space in the buffer before adding an item.
* **Critical Section:** Updates the buffer and index in.
* **Release** full\_slots **Lock:** Signals that there is now at least one item in the buffer.
* **Consumer Function:**
* **Acquire** full\_slots **Lock:** Ensures that there is at least one item in the buffer before removing it.
* **Critical Section:** Updates the buffer and index out.
* **Release** empty\_slots **Lock:** Signals that there is now an empty slot in the buffer.
* **Parallel Region:**
* #pragma omp parallel**:** Starts a parallel region.
* #pragma omp single**:** Ensures that the producer and consumer tasks are started only once.
* #pragma omp task**:** Creates tasks for producers and consumers to run concurrently.
* **Lock Management:**
* omp\_init\_lock()**:** Initializes the locks.
* omp\_set\_lock() **and** omp\_unset\_lock()**:** Manage access to critical sections.
* omp\_destroy\_lock()**:** Cleans up the locks after execution.

This implementation effectively manages synchronization and allows for concurrent execution of producer and consumer tasks, demonstrating the use of different OpenMP constructs to handle the Producer-Consumer problem.

**Github Link:** <https://github.com/YashNawale26/High-Performance-Computing.git>