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

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

**Course:** High Performance Computing Lab

**Practical No. 4**

**Exam Seat No:**

**Title of practical:**

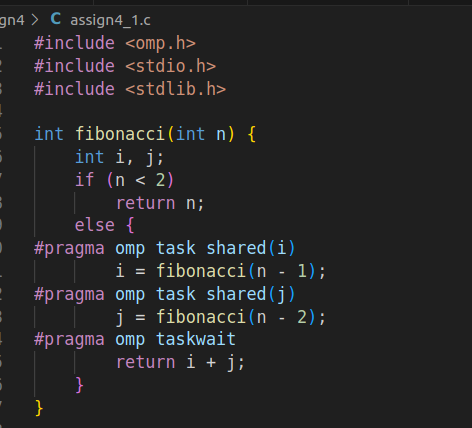
Study and Implementation of Synchronization

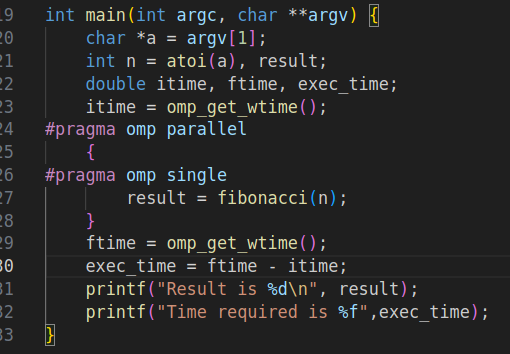
**Problem Statement 1:**

# 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:**

****

****

**Output:**

|  |  |  |
| --- | --- | --- |
| N | fib(N) | Time |
| 10 | 55 | 0.000715 |
| 20 | 6765 | 0.013951 |
| 30 | 832040 | 1.359622 |

**Analysis:**

The time to compute the Fibonacci number grows exponentially as N increases. This is due to the exponential number of recursive calls in the naive recursive approach. Each increase in N approximately doubles the number of recursive calls, resulting in an exponential increase in the number of tasks.

**Information:**

Synchronization in parallel programming ensures that multiple threads or processes work together correctly and avoid issues such as race conditions, data corruption, and inconsistent results

### ****Critical Sections:****Ensures that a section of code is executed by only one thread at a time

### ****Barriers****: A synchronization point where threads wait until all threads in the team reach this point before proceeding.

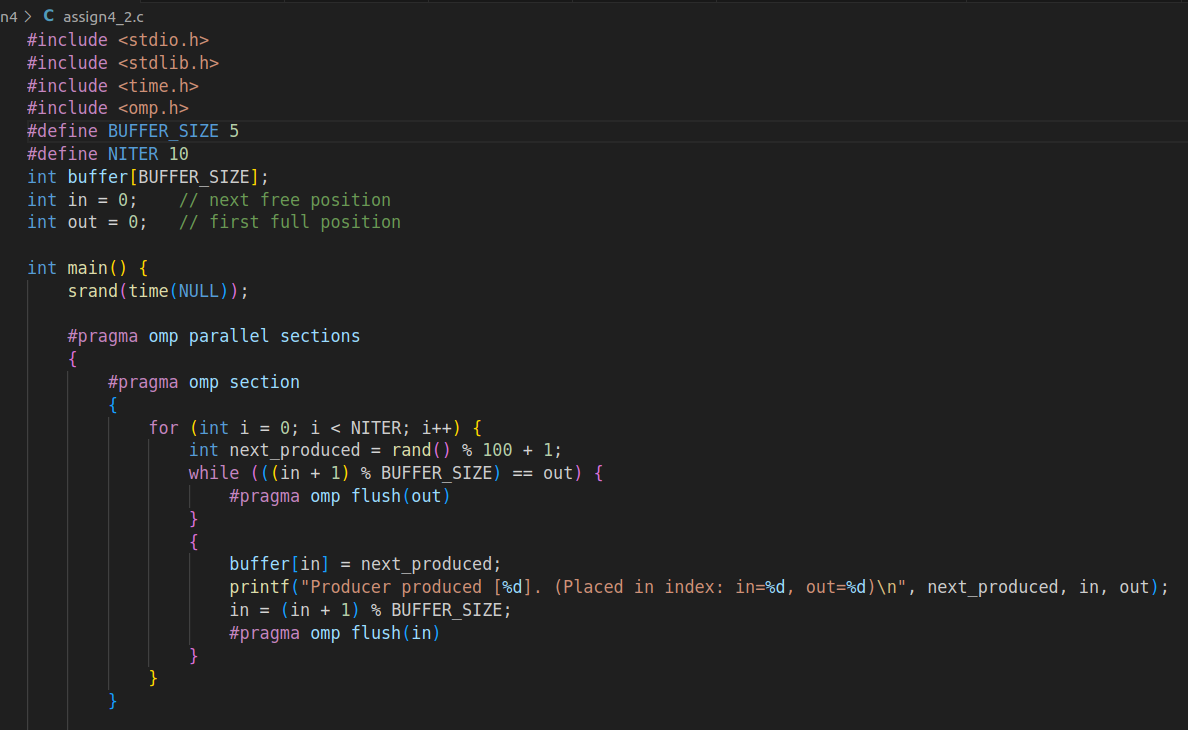
**Locks:** Provides explicit mutual exclusion control over critical sections of code

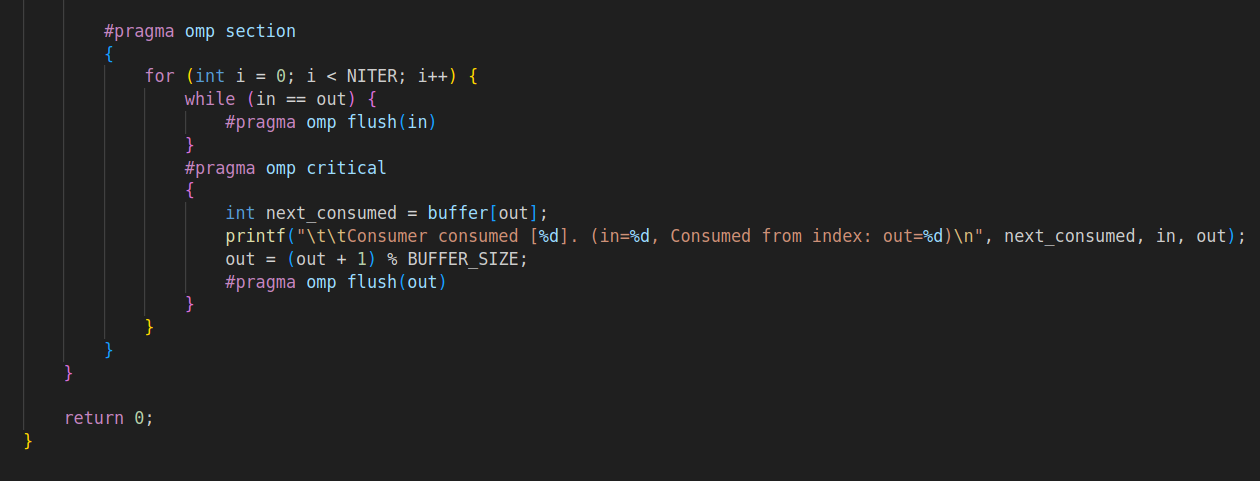
**Problem Statement 2:**

# 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:**

****

****

**Output:**



**Information:**

**The Producer-Consumer problem is a classic example of synchronization in concurrent programming. It involves two types of processes:**

1. **Producer:** Generates data or items and places them into a shared buffer.
2. **Consumer:** Takes items from the buffer and processes them.

In a parallel programming context, this problem highlights the challenges of coordinating access to shared resources between concurrent processes or threads.

**Producer Section:**

* The producer generates a random number and tries to place it into the buffer.
* It uses a while loop to wait if the buffer is full. This is checked using the condition ((in + 1) % BUFFER\_SIZE) == out, which ensures there is space in the buffer.
* The producer uses #pragma omp flush(out) to ensure visibility of changes to out by other threads.
* After placing an item in the buffer, the producer updates the in index and uses #pragma omp flush(in) to ensure the update is visible to other threads.
* **Consumer Section:**
* The consumer waits if the buffer is empty (while (in == out)) and uses #pragma omp flush(in) to ensure visibility of changes to in by other threads.
* The consumer accesses the buffer in a critical section to avoid race conditions when updating out. This ensures that the buffer is accessed safely.
* The #pragma omp flush(out) is used to ensure that updates to out are visible to other threads.

**Github Link:**

https://github.com/sujanM14/HPC\_Lab