**Course: High Performance Computing Lab**

**Practical No 4**

**PRN : 22510046**

**Name: Tanmay Chavan**

**Batch : B3**

**Title -** 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)

**Solution**:

Fibonacci computation is sequential in nature if done recursively, but with iteration, we can parallelize certain parts. However, due to data dependency (fib[i] = fib[i-1] + fib[i-2]), **synchronization** is needed to avoid race conditions.

We will:

* Use #pragma omp parallel with #pragma omp single for sequential initialization
* Use #pragma omp critical to ensure only one thread updates the shared array at a time.

**#include <bits/stdc++.h>**

**#include <omp.h>**

**using namespace std;**

**int main() {**

**int n;**

**cout << "Enter number of Fibonacci terms: ";**

**cin >> n;**

**vector<long long> fib(n);**

**fib[0] = 0;**

**fib[1] = 1;**

**#pragma omp parallel**

**{**

**#pragma omp single**

**{**

**cout << "Thread " << omp\_get\_thread\_num() << " initializing first terms\n";**

**}**

**for (int i = 2; i < n; i++) {**

**#pragma omp critical**

**{**

**fib[i] = fib[i - 1] + fib[i - 2];**

**cout << "Thread " << omp\_get\_thread\_num()**

**<< " computed fib[" << i << "] = " << fib[i] << "\n";**

**}**

**}**

**}**

**cout << "\nFibonacci Series: ";**

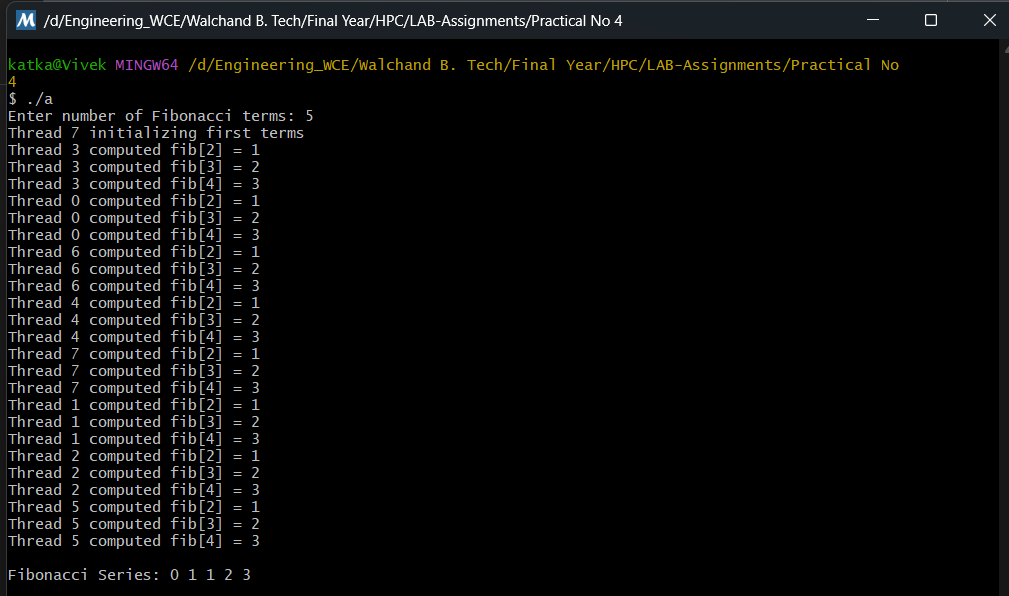
**for (int i = 0; i < n; i++)**

**cout << fib[i] << " ";**

**cout << endl;**

**return 0;**

**}**



**Synchronization Used**

* **critical** → Ensures only one thread updates and prints at a time (avoids race conditions).
* **single** → Only one thread executes the initialization message.

**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)

**Solution:**

* **Producer**: Generates data and places it in a shared buffer.
* **Consumer**: Retrieves data from the buffer.
* **Synchronization**: Prevents producer from overwriting unconsumed data and consumer from reading empty buffer.

In OpenMP:

* Use omp\_lock\_t for explicit locking.
* Use #pragma omp barrier for synchronization between producer and consumer.

**#include <bits/stdc++.h>**

**#include <omp.h>**

**using namespace std;**

**const int BUFFER\_SIZE = 5;**

**int buffer[BUFFER\_SIZE];**

**int count\_items = 0;  // number of items in buffer**

**int in\_pos = 0, out\_pos = 0;**

**omp\_lock\_t lock\_var;**

**void produce\_item(int item) {**

**buffer[in\_pos] = item;**

**in\_pos = (in\_pos + 1) % BUFFER\_SIZE;**

**count\_items++;**

**}**

**int consume\_item() {**

**int item = buffer[out\_pos];**

**out\_pos = (out\_pos + 1) % BUFFER\_SIZE;**

**count\_items--;**

**return item;**

**}**

**int main() {**

**omp\_init\_lock(&lock\_var);**

**#pragma omp parallel num\_threads(2)**

**{**

**int tid = omp\_get\_thread\_num();**

**if (tid == 0) { // Producer**

**for (int i = 1; i <= 10; i++) {**

**bool produced = false;**

**while (!produced) {**

**omp\_set\_lock(&lock\_var);**

**if (count\_items < BUFFER\_SIZE) {**

**produce\_item(i);**

**cout << "Producer produced: " << i << "\n";**

**produced = true;**

**}**

**omp\_unset\_lock(&lock\_var);**

**}**

**}**

**} else { // Consumer**

**for (int i = 1; i <= 10; i++) {**

**bool consumed = false;**

**while (!consumed) {**

**omp\_set\_lock(&lock\_var);**

**if (count\_items > 0) {**

**int item = consume\_item();**

**cout << "Consumer consumed: " << item << "\n";**

**consumed = true;**

**}**

**omp\_unset\_lock(&lock\_var);**

**}**

**}**

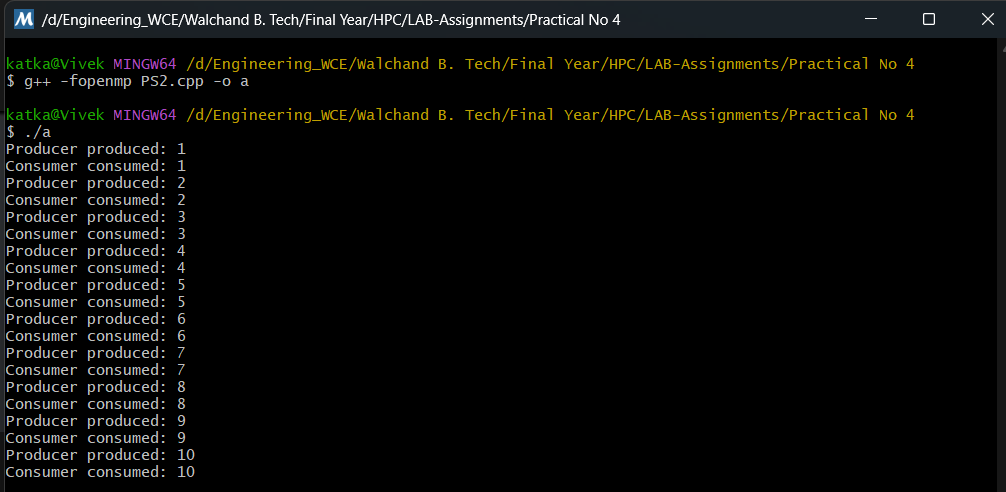
**}**

**}**

**omp\_destroy\_lock(&lock\_var);**

**return 0;**

**}**



**Synchronization Used**

* **omp\_lock\_t with omp\_set\_lock / omp\_unset\_lock`** → Ensures exclusive access to shared buffer.
* **Busy-wait loops** ensure producer waits if buffer is full and consumer waits if empty.
* **No race condition** occurs because of proper locking.

**Github Link: https://github.com/Kodar11/HPC-Lab**

**Conclusion**

In this practical, we studied and implemented synchronization techniques in OpenMP through two classic problems — Fibonacci computation and the Producer-Consumer problem. We explored how data dependencies and shared resources in parallel programs can cause race conditions if not handled properly. Using constructs like critical, single, and explicit locks (omp\_lock\_t), we ensured correct execution by controlling access to shared variables. The experiments demonstrated that proper synchronization guarantees correctness but can introduce overhead if overused. Thus, in high-performance computing, synchronization mechanisms must be applied judiciously to balance between **data safety** and **execution efficiency**.