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

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

**Course:** High Performance Computing Lab

**Practical No. 4**

**Exam Seat No: 23520004**

**Name : Harshad Ajit Dange**

**Batch : B4**

**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:**

#include <iostream>

#include <vector>

#include <omp.h>

using namespace std;

int main() {

    int n ;

    cin>>n;

    vector<long long> v(n + 1, 0);

    v[0] = 0;

    v[1] = 1;

    double start = omp\_get\_wtime();

    #pragma omp parallel

    {

        #pragma omp single

        {

            cout << "Computing series using " << omp\_get\_num\_threads() << " threads\n";

        }

        #pragma omp for schedule(static) nowait

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

            #pragma omp critical

            {

                v[i] = v[i - 1] + v[i - 2];

            }

        }

    }

    double end = omp\_get\_wtime();

    cout << "series: ";

    for (int i = 0; i <= n; i++)

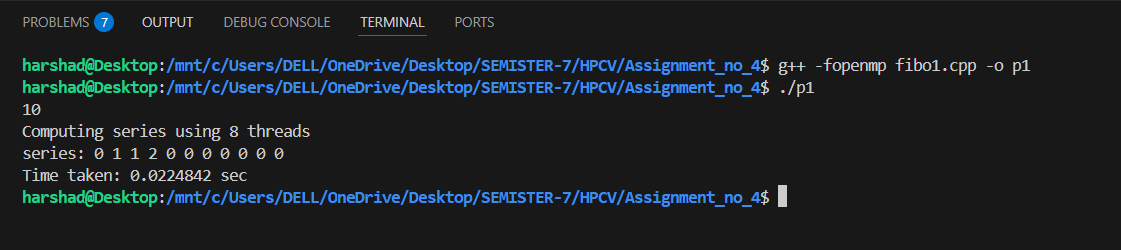
        cout << v[i] << " ";

    cout << "\nTime taken: " << (end - start) << " sec\n";

    return 0;

}

**Screenshot:**



#include <iostream>

#include <omp.h>

using namespace std;

long long find(int n) {

    if (n < 2) return n;

    long long x, y;

    #pragma omp task shared(x)

    x = find(n - 1);

    #pragma omp task shared(y)

    y = find(n - 2);

    #pragma omp taskwait

    return x + y;

}

int main() {

    int n = 30;

    double start = omp\_get\_wtime();

    long long result;

    #pragma omp parallel

    {

        #pragma omp single

        {

            result = find(n);

        }

    }

    double end = omp\_get\_wtime();

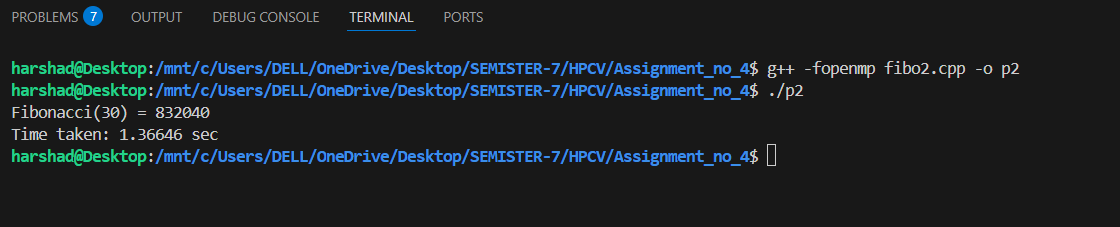
    cout << "Fibonacci(" << n << ") = " << result << endl;

    cout << "Time taken: " << (end - start) << " sec\n";

    return 0;

}

**Screenshot:**



**Information:**

To implement Fibonacci number computation in parallel using OpenMP, considering synchronization requirements and demonstrating the use of different clauses and constructs.

### Problem Description

The Fibonacci sequence is a series of numbers in which each number is the sum of the two preceding ones. It starts with 0 and 1, and the formula is:

 F(0)=0,F(1)=1,F(n)=F(n−1)+F(n−2)for n≥2

Example sequence:  
0, 1, 1, 2, 3, 5, 8, 13, 21, …

### Parallelization Goal

The task is to speed up Fibonacci computation by using multiple threads.

* In the recursive approach, independent recursive calls can be executed as parallel tasks.
* In the iterative approach, computations are sequentially dependent but can still be parallelized in some stages or restructured.

### Why Synchronization is Needed

In iterative computation, each Fibonacci number depends on the two preceding numbers.  
If multiple threads update or read shared variables without synchronization:

* Data races may occur.
* Results can be inconsistent or incorrect.

OpenMP provides constructs like critical, single, taskwait, and flush to ensure data consistency

**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

#include <iostream>

#include <queue>

#include <omp.h>

using namespace std;

const int buff\_SIZE = 5;

const int PRODUCE\_COUNT = 10;

queue<int> buff;

int prod\_item = 0;

int cons\_item = 0;

int main() {

    omp\_set\_num\_threads(2);

    double start = omp\_get\_wtime();

    #pragma omp parallel sections shared(buff, prod\_item, cons\_item)

    {

        #pragma omp section

        {

            while (prod\_item < PRODUCE\_COUNT) {

                #pragma omp critical

                {

                    if (buff.size() < buff\_SIZE) {

                        int item = prod\_item + 1;

                        buff.push(item);

                        prod\_item++;

                        cout << "Producer produced: " << item << "\n";

                    }

                }

                #pragma omp flush

            }

        }

        #pragma omp section

        {

            while (cons\_item < PRODUCE\_COUNT) {

                #pragma omp critical

                {

                    if (!buff.empty()) {

                        int item = buff.front();

                        buff.pop();

                        cons\_item++;

                        cout << "Consumer consumed: " << item << endl;

                    }

                }

                #pragma omp flush

            }

        }

    }

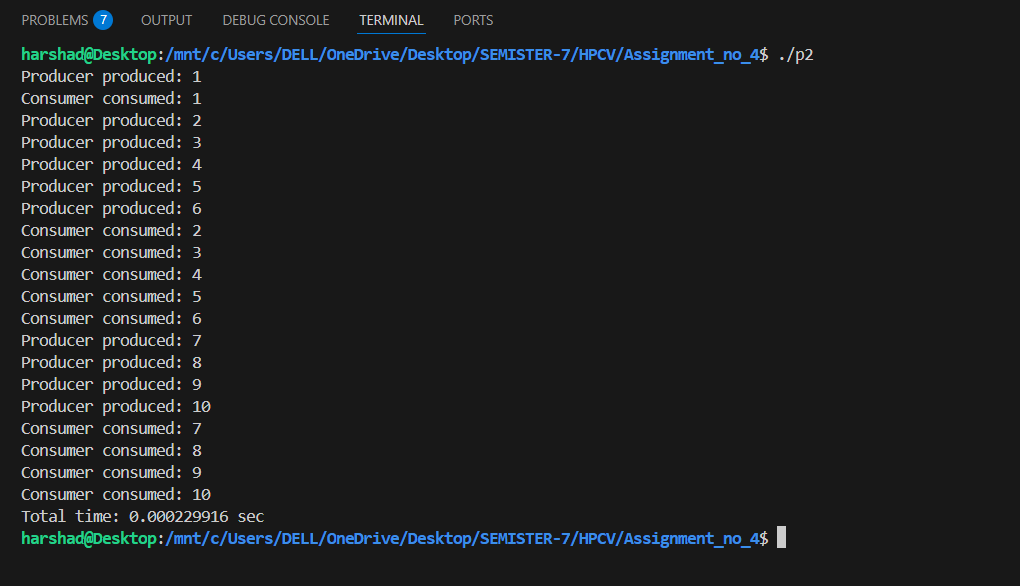
    double end = omp\_get\_wtime();

    cout << "Total time: " << (end - start) << " sec\n";

    return 0;

}

**Screenshots:**

****

**Information:**

The Producer–Consumer problem is a classic synchronization example in concurrent programming where:

* Producer generates data items and places them into a shared buffer.
* Consumer removes data from the buffer and processes them.
* Synchronization is required to prevent:
  1. Buffer overflow
  2. Buffer underflow
  3. Data races

### **Parallelization with OpenMP :**

OpenMP allows us to create multi-threaded producer–consumer code using:

1. parallel sections — to assign separate threads for producing and consuming.
2. Synchronization constructs — to coordinate access to the shared buffer.

**Github Link:**

https://github.com/harshad4507/23520004-HPCV/tree/main/Assignment\_no\_4