Walchand College of Engineering, Sangli  
Department of Computer Science and Engineering

Class: Final Year (CSE) Year: 2025-26 Semester: 1

Course: High Performance Computing Lab

**Practical No. 4**

**Exam Seat No: 22510059**

**Name : Chetna Prakash Ghengare**

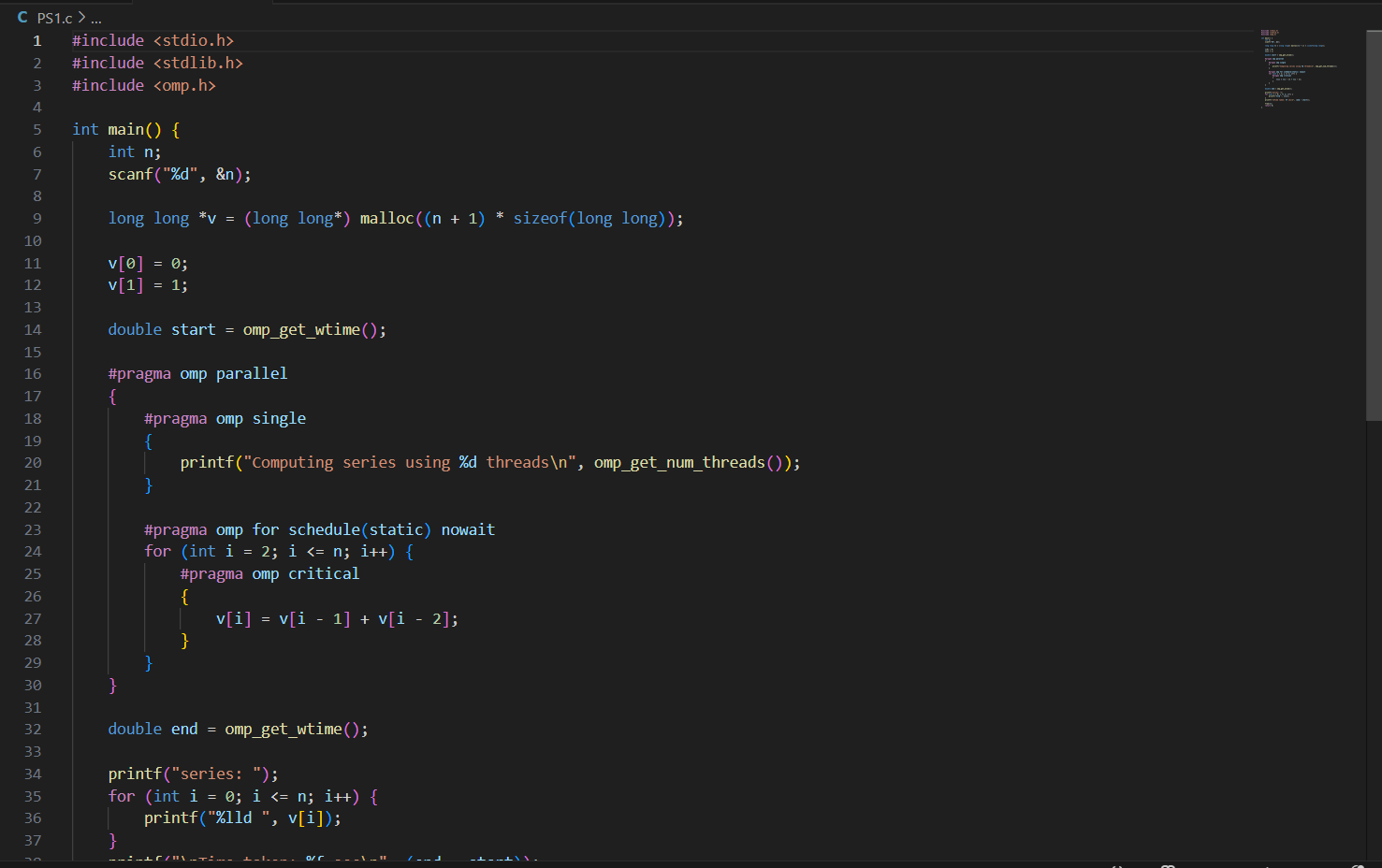
**Batch : B3**

**Title of practical:**

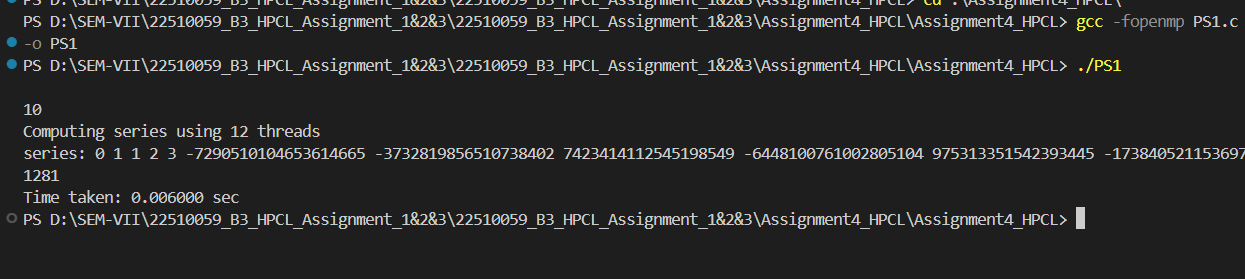
Study and Implementation of Synchronization

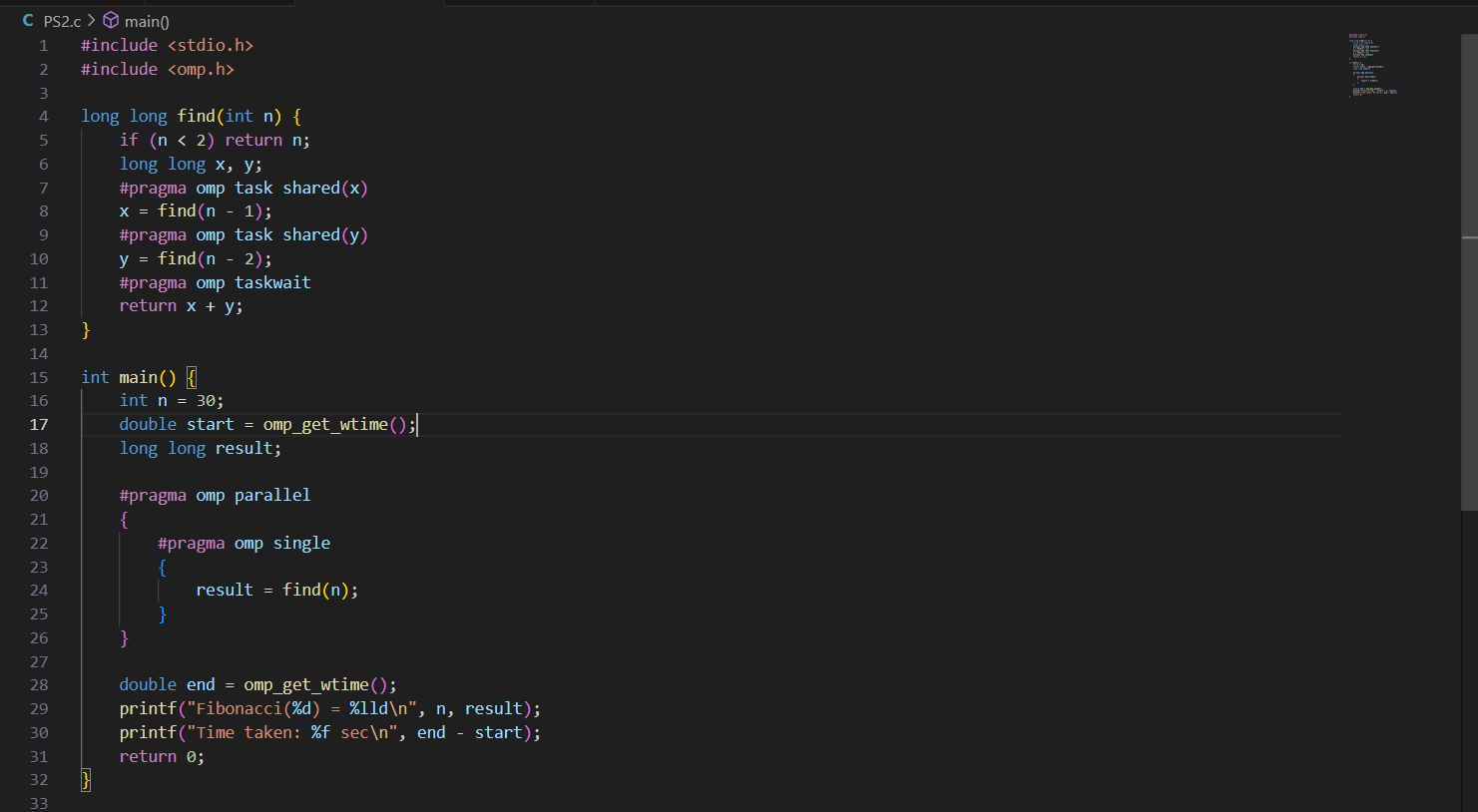
**Problem Statement 1:**

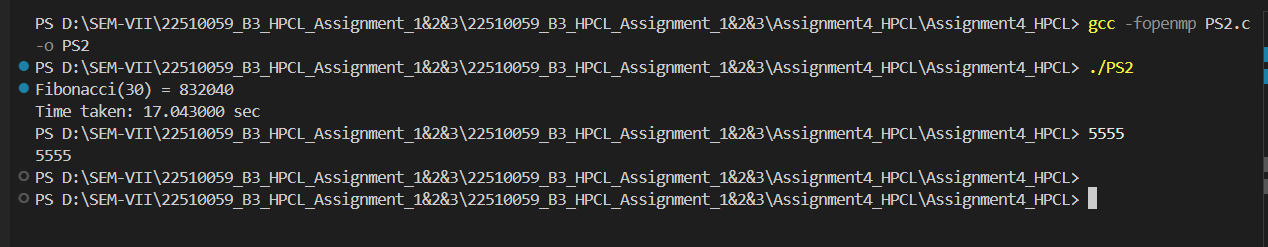
# Fibonacci Computation:

****

****

****

****

****

The idea is to calculate Fibonacci numbers in parallel using OpenMP. While doing this, we also need to handle synchronization so that threads don’t interfere with each other. It also helps us understand how OpenMP clauses like critical, single, task, and taskwait are used.

Problem Description

Fibonacci is a sequence where each number is the sum of the two before it.

It starts with 0 and 1.

Formula:

F(0) = 0

F(1) = 1

F(n) = F(n-1) + F(n-2) for n ≥ 2

Example:

0, 1, 1, 2, 3, 5, 8, 13, 21 …

Parallelization Goal:

We want to make Fibonacci faster by running it on multiple threads.

In the recursive version, calls like fib(n-1) and fib(n-2) can run at the same time, so we use OpenMP tasks.

In the iterative version, values are built step by step, which is more dependent, but we can still use OpenMP with synchronization to avoid mistakes.

Why Synchronization is Needed

In the loop version, each Fibonacci number depends on the previous two. If two threads update or read shared data at the same time, things can go wrong (data race). That’s why we need synchronization.

OpenMP gives us tools for this:

critical → makes sure only one thread updates at a time.

single → only one thread runs a block while others wait.

taskwait → ensures tasks finish before combining results.

flush → makes memory updates visible to all threads.

**Problem Statement 2:**

Producer Consumer Problem

#include <stdio.h>

#include <omp.h>

#define BUFF\_SIZE 5

#define PRODUCE\_COUNT 10

int buffer[BUFF\_SIZE];

int in = 0, out = 0;

int count = 0;

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(buffer, in, out, count, prod\_item, cons\_item)

    {

        // Producer

        #pragma omp section

        {

            while (prod\_item < PRODUCE\_COUNT) {

                #pragma omp critical

                {

                    if (count < BUFF\_SIZE) {

                        int item = prod\_item + 1;

                        buffer[in] = item;

                        in = (in + 1) % BUFF\_SIZE;

                        prod\_item++;

                        count++;

                        printf("Producer produced: %d\n", item);

                    }

                }

                #pragma omp flush

            }

        }

        // Consumer

        #pragma omp section

        {

            while (cons\_item < PRODUCE\_COUNT) {

                #pragma omp critical

                {

                    if (count > 0) {

                        int item = buffer[out];

                        out = (out + 1) % BUFF\_SIZE;

                        cons\_item++;

                        count--;

                        printf("Consumer consumed: %d\n", item);

                    }

                }

                #pragma omp flush

            }

        }

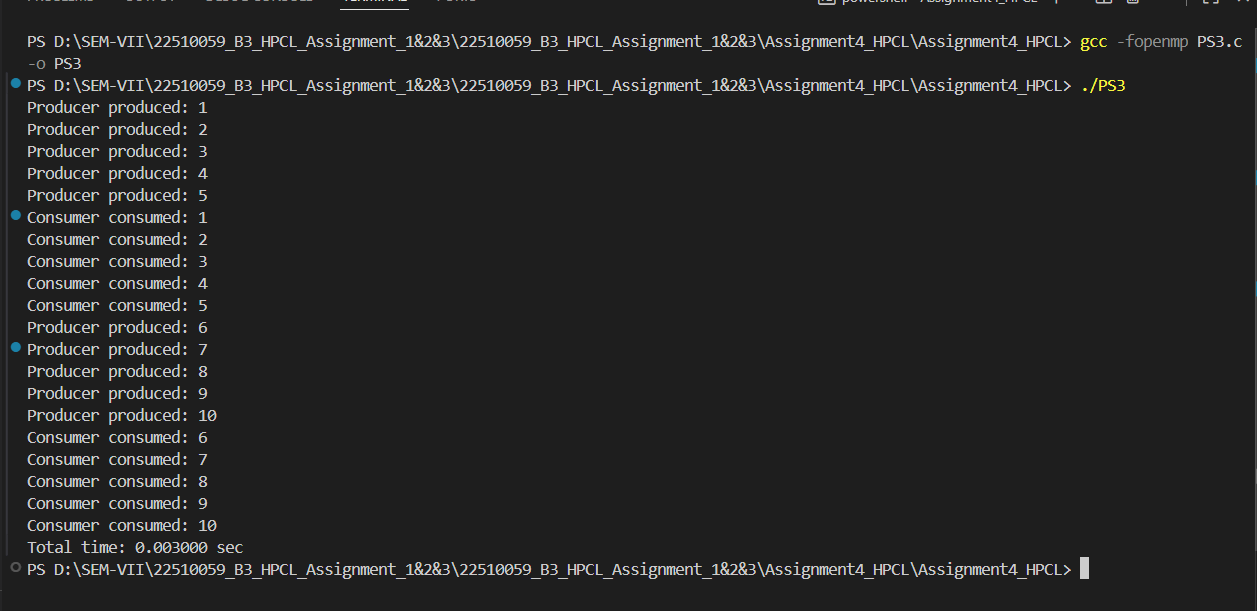
    }

    double end = omp\_get\_wtime();

    printf("Total time: %f sec\n", end - start);

    return 0;

}

****

**Information:**

Producer–Consumer Problem

The Producer–Consumer problem is a well-known example in concurrent programming that deals with synchronization.

The producer creates items and puts them into a common buffer.

The consumer takes items from the buffer and uses them.

Proper coordination is needed to avoid issues like:

Buffer getting full (overflow)

Buffer becoming empty (underflow)

Multiple threads accessing shared data incorrectly

Using OpenMP for Parallelization

OpenMP makes it easier to implement this problem with multiple threads by:

Using parallel sections so one thread acts as the producer and another as the consumer.

Applying synchronization constructs to make sure the shared buffer is accessed safely without conflicts.

https://github.com/Chetnaghengare/22510059\_HPCL\_Assignment.git