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

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

**Course:** High Performance Computing Lab

**Practical No. 4**

**Exam Seat No: 22510023**

**Name : Ganesh Chavhan**

**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:

### Design and Approach

Used **OpenMP task parallelism** for recursive decomposition.

**Synchronization:**

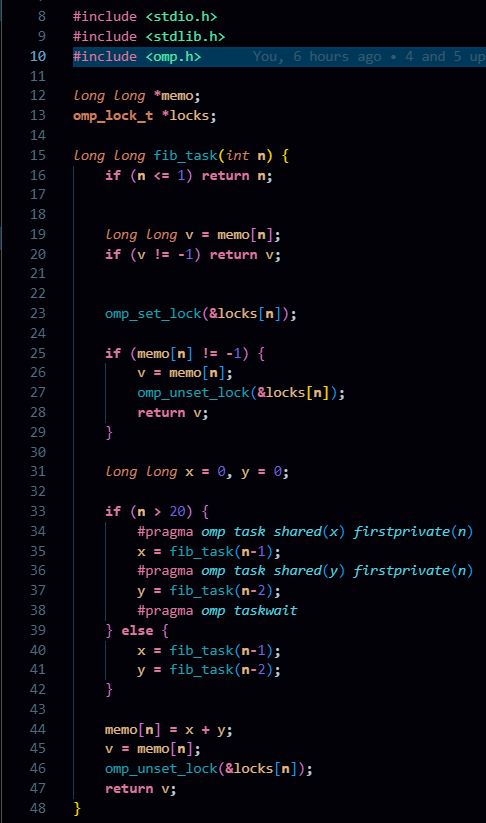
#pragma omp taskwait ensures child tasks complete.

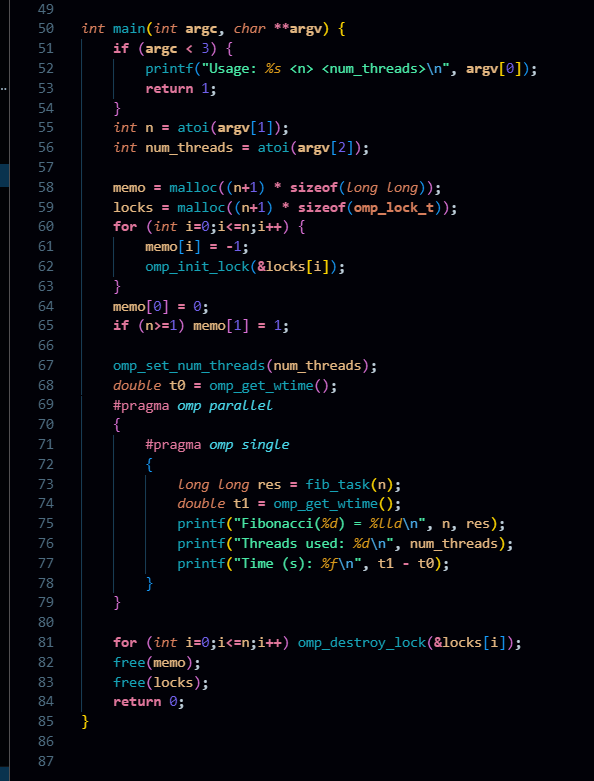
#pragma omp taskgroup ensures all tasks finish before leaving region.

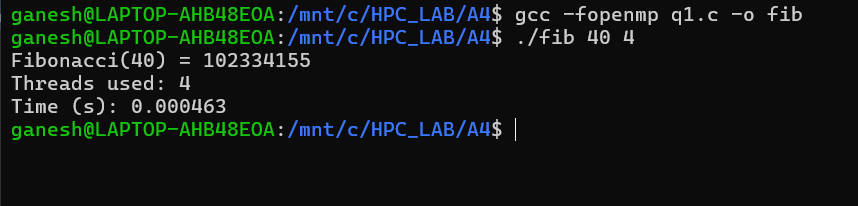
**Clauses Used:** parallel, single, task, taskwait, taskgroup, firstprivate, shared.

Cutoff technique used: small subproblems solved sequentially to reduce task overhead.

**Screenshots:**







**Information:**

Uses recursive Fibonacci with OpenMP task and taskwait.

**Memoization** array (memo[]) avoids recomputation.

Each memo entry guarded by an **OpenMP lock** to ensure correctness.

Threads created once; tasks spawned for recursive branches.

Execution time measured with omp\_get\_wtime().

**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

### Design and Approach

Used **circular buffer** for shared data.

**Synchronization:**

#omp\_lock\_t for mutual exclusion in buffer push/pop.

#pragma omp atomic for counters (produced, consumed, count).

Producers and consumers created as **OpenMP tasks** inside a single region.

Spin-wait used for full/empty buffer states.

**Screenshots:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

*int* main(*int* **argc,** *char* **\*\*argv**) {

**if** (**argc** **<** 5) {

        printf("Usage: *%s* <bufsize> <items> <producers> <consumers>\n"**,** **argv**[0])**;**

**return** 1**;**

    }

*int* bufsize      **=** atoi(**argv**[1])**;**

*int* total\_items  **=** atoi(**argv**[2])**;**

*int* nprod        **=** atoi(**argv**[3])**;**

*int* ncons        **=** atoi(**argv**[4])**;**

*int* **\***buffer **=** malloc(bufsize **\*** **sizeof**(*int*))**;**

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

*int* count **=** 0**;**

**omp\_lock\_t** lock\_in**,** lock\_out**;**

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

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

*static* *int* global\_produced **=** 0**;**

*static* *int* global\_consumed **=** 0**;**

    omp\_set\_num\_threads(nprod **+** ncons)**;**

*double* t0 **=** omp\_get\_wtime()**;**

    #pragma *omp* *parallel*

    {

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

**if** (tid **<** nprod) {

*// Producer threads*

**while** (1) {

*int* myglobal**;**

                #pragma *omp* *atomic* *read*

                myglobal **=** global\_produced**;**

**if** (myglobal **>=** total\_items)

**break;**

                #pragma *omp* *atomic* *update*

                global\_produced**++;**

*int* item **=** myglobal **+** 1**;**

*int* placed **=** 0**;**

**while** (**!**placed) {

**if** (count **<** bufsize) {

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

**if** (count **<** bufsize) {

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

                            in **=** (in **+** 1) **%** bufsize**;**

                            #pragma *omp* *atomic* *update*

                            count**++;**

                            placed **=** 1**;**

                            printf("[Producer *%d*] produced *%d* (count=*%d*)\n"**,** tid**,** item**,** count)**;**

                        }

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

                    }

                }

            }

        } **else** {

*// Consumer threads*

**while** (1) {

*int* mycon**;**

                #pragma *omp* *atomic* *read*

                mycon **=** global\_consumed**;**

**if** (mycon **>=** total\_items)

**break;**

                #pragma *omp* *atomic* *update*

                global\_consumed**++;**

*int* got **=** 0**;**

*int* item **=** **-**1**;**

**while** (**!**got) {

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

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

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

                            item **=** buffer[out]**;**

                            out **=** (out **+** 1) **%** bufsize**;**

                            #pragma *omp* *atomic* *update*

                            count**--;**

                            got **=** 1**;**

                            printf("    [Consumer *%d*] consumed *%d* (count=*%d*)\n"**,** tid**,** item**,** count)**;**

                        }

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

                    }

                }

            }

        }

    }

*double* t1 **=** omp\_get\_wtime()**;**

    printf("All done. Time: *%f* sec\n"**,** t1 **-** t0)**;**

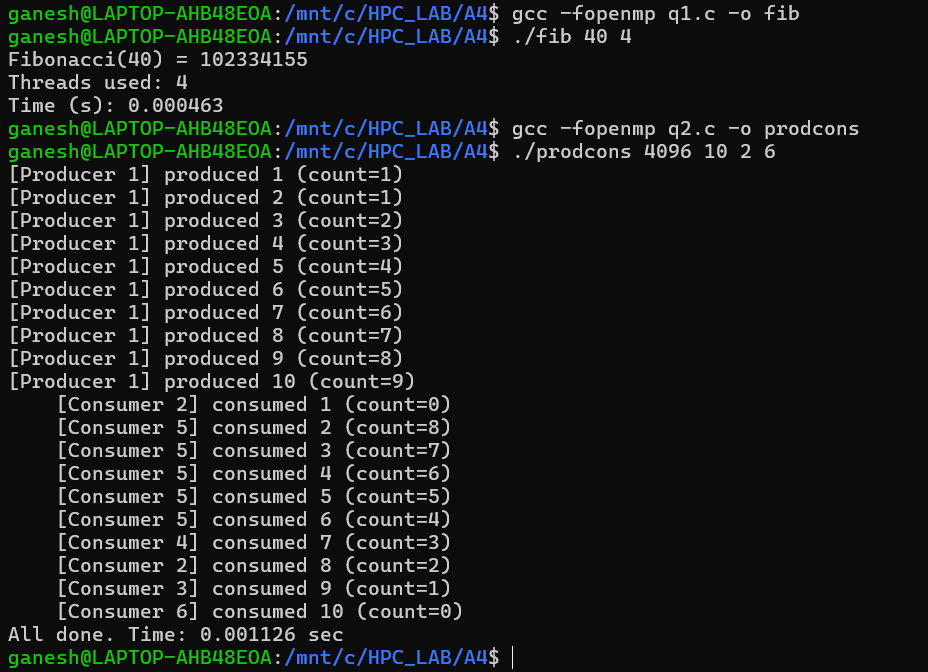
    omp\_destroy\_lock(**&**lock\_in)**;**

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

    free(buffer)**;**

**return** 0**;**

}



**Information:**

Shared **circular buffer** of given size.

Uses **locks** (omp\_lock\_t) for enqueue (producer) and dequeue (consumer).

Uses **atomics** for counters: global\_produced, global\_consumed, count.

Producers add items to buffer until target count is reached.

Consumers remove items until all items consumed.

**Analysis:**

Classic synchronization problem demonstrating OpenMP locks and atomics.

Ensures 0 <= count <= buffer\_size invariant always holds.

Too many producers/consumers → higher contention on locks.

Parallel runtime depends on buffer size: larger buffers reduce blocking.

Demonstrates coordination of multiple producer and consumer threads in parallel.

**Github Link: <https://github.com/Ganesh-Chavhan/HPC_LAB/tree/main/A4>**