Class: Final Year B.Tech(Computer Science and Engineering)

Year: 2025-26 Semester: 1

Course: High Performance Computing Lab

**Practical No. 4**

**Exam Seat No: 23520012**

Title of practical:

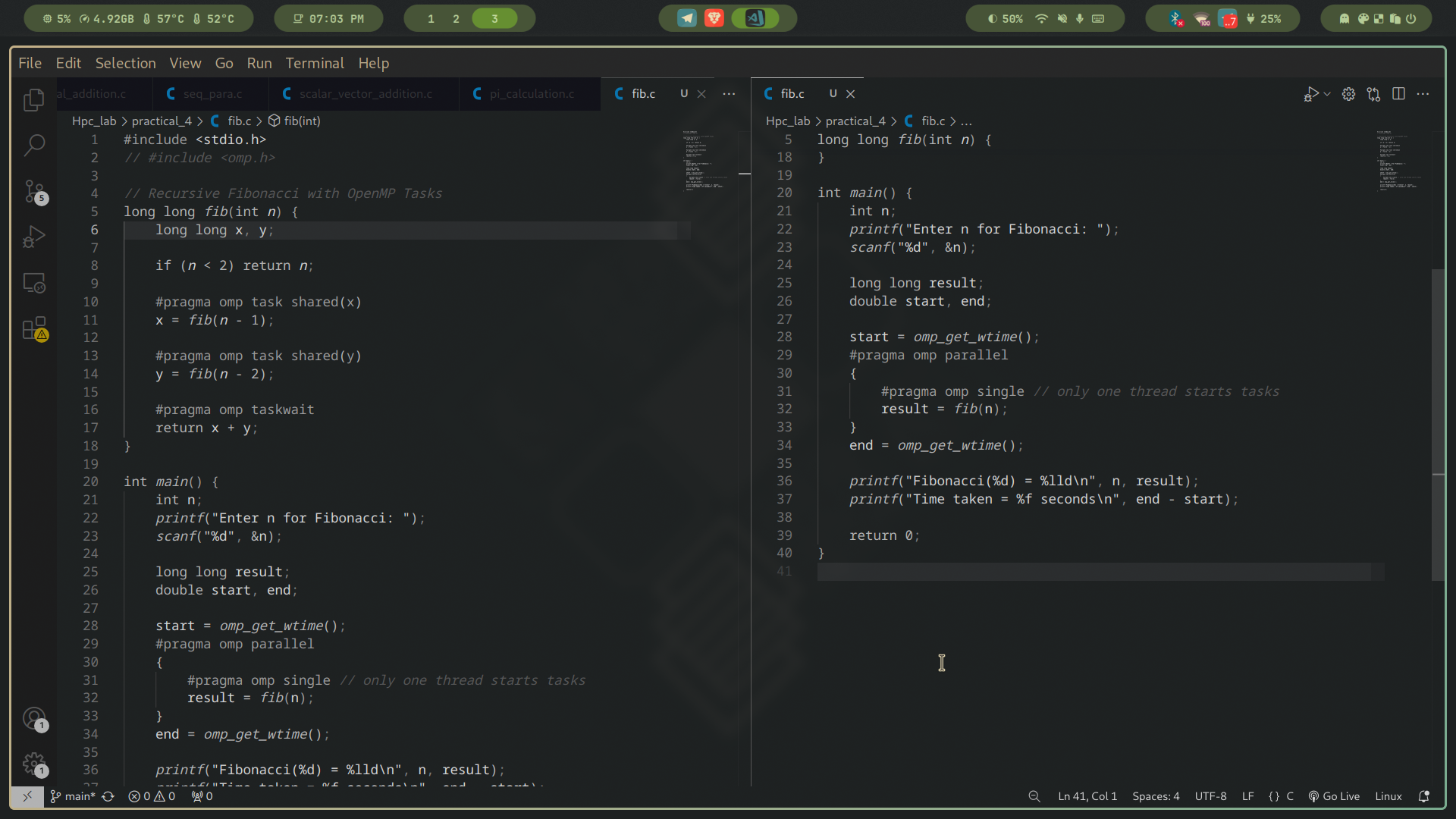
Study and Implementation of Synchronization

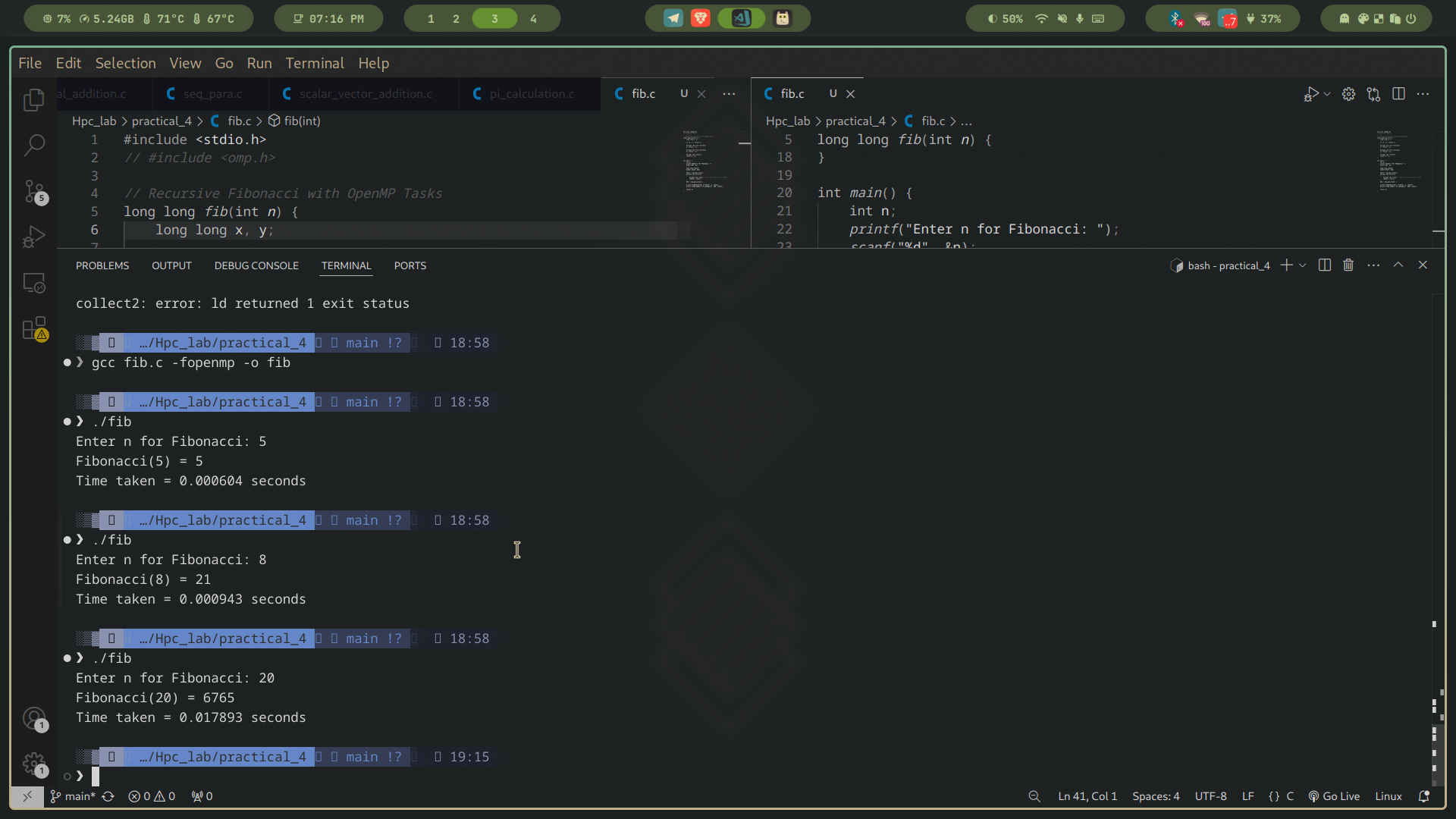
Problem Statement 1:

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:

Screenshots:





Information:

### **Parallelization Approach**

* We use **OpenMP tasks** to parallelize the recursive computation.
* For each recursive call fib(n-1) and fib(n-2), a new **task** can be created so that separate threads can work on them concurrently.
* **Task synchronization (#pragma omp taskwait)** ensures that the parent task waits until both child tasks are finished before combining results.

### **Explanation of Key OpenMP Constructs**

1. **#pragma omp parallel**
   * Creates a team of threads.
2. **#pragma omp single**
   * Ensures only one thread initiates the top-level fib(n) call.
   * Other threads are free to execute created tasks.
3. **#pragma omp task**
   * Creates a new task for computing fib(n-1) or fib(n-2).
   * The OpenMP runtime schedules these tasks among available threads.
4. **#pragma omp taskwait**
   * Ensures both recursive tasks complete before summing results.
   * This maintains correctness (synchronization).

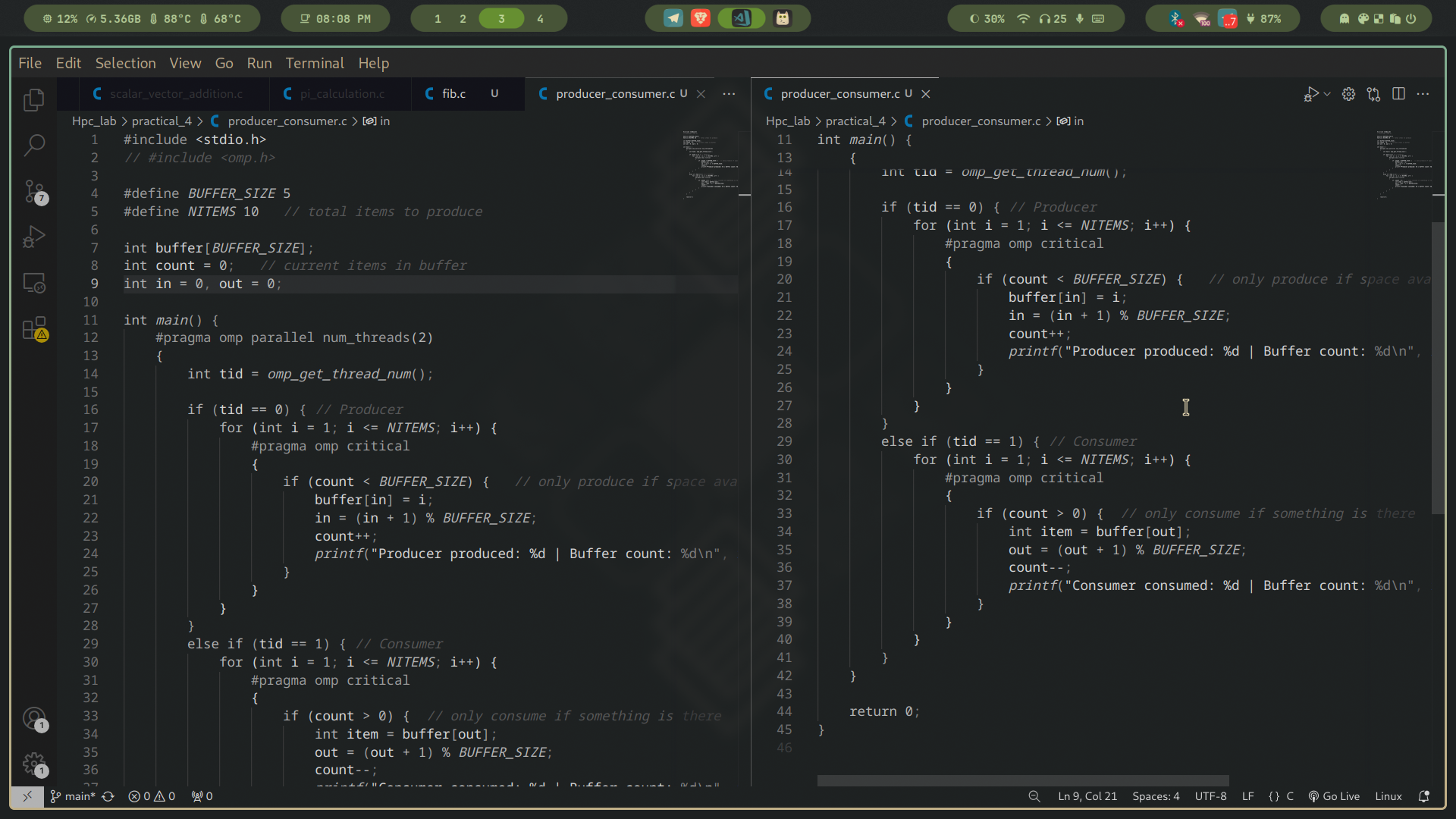
### **Analysis**

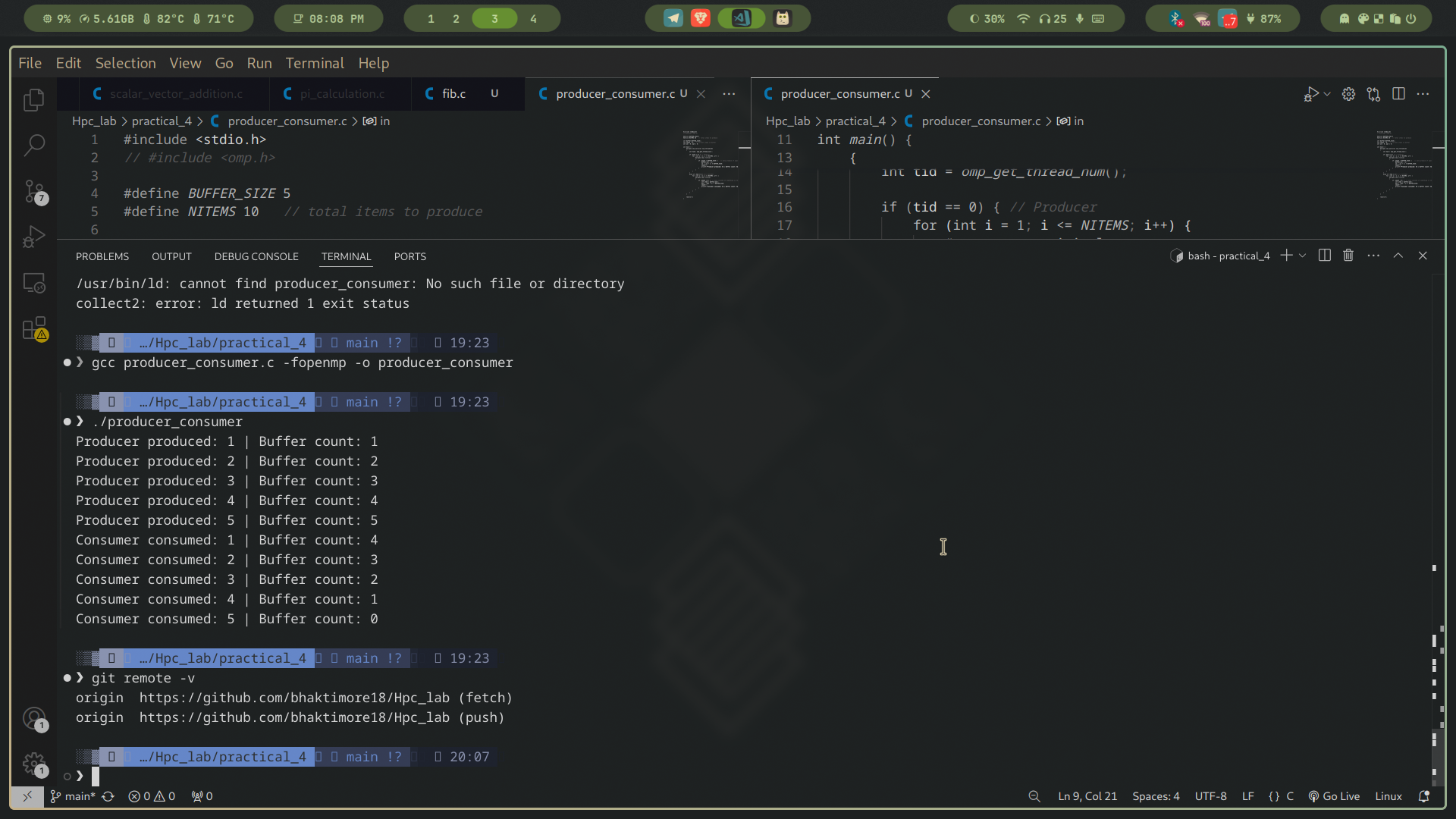
* **Sequential Recursive Fibonacci**: Recomputes the same subproblems multiple times → exponential runtime.
* **Parallel Recursive Fibonacci**: Multiple branches of recursion execute concurrently, significantly reducing runtime for large n.
* **Overhead Consideration**: For small n, task creation overhead may dominate. So parallel recursive Fibonacci is beneficial only for larger values of n.

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





This is a **producer** that generates items and a **consumer** that removes them from a **bounded circular buffer**. We must prevent:

* Producer from writing when the buffer is **full**.
* Consumer from reading when the buffer is **empty**.
* **Race conditions** when both modify shared state (buffer, in, out, count).

Data structure: **circular buffer** of size BUFFER\_SIZE.

Shared state:

* buffer[BUFFER\_SIZE] – storage
* in – write index
* out – read index
* count – number of items currently in the buffer

Parallelization:

* 2 OpenMP threads: **thread 0 = producer**, **thread 1 = consumer**.

Synchronization:

* Use a **#pragma omp critical** section as a simple “monitor” to protect *all* shared updates.
* Use a **retry loop** that checks the condition *inside* the same critical section (to avoid data races and lost items).  
   (This replaces busy-waiting outside the critical, keeps correctness simple.)
* #pragma omp parallel num\_threads(2)  
   Creates exactly two threads running the same main region. We assign roles by thread ID:  
  + tid == 0 → producer
  + tid != 0 → consumer
* #pragma omp critical  
   Forms a **mutual exclusion** region so only one thread at a time reads/writes buffer, in, out, count.  
   Also provides the necessary **memory synchronization** (implicit flush on entry/exit) so updates are visible across threads.
* **Shared vs private**
  + buffer, in, out, count are **shared**.
  + Loop counters and temporaries like value, taken, item, did\_produce, did\_consume are **private** to each thread by default.
* **No data races**: All reads/writes of shared state that could conflict occur **inside the same critical region**.
* **Bounded buffer discipline**:  
  + Producer inserts **only if** count < BUFFER\_SIZE.
  + Consumer removes **only if** count > 0.
  + If the condition isn’t met, the thread exits the critical region and **retries** (simple, busy-retry design).
* **No lost items / missed consumes**:  
   The producer increments value **only after** a successful insert; the consumer increments taken **only after** a successful remove. Thus exactly NITEMS are produced and consumed.
* Time complexity remains **O(NITEMS)** for both roles.
* The retry loops are a simple form of **busy waiting**. For teaching/demo this is fine; for high performance, you’d consider:  
  + OpenMP **locks** (omp\_lock\_t) or
  + **Tasks + dependencies**, or
  + OS primitives/condition variables (outside OpenMP’s scope).
* The critical block serializes buffer access; with *multiple* producers/consumers, this remains correct but may limit throughput (acceptable for demonstration).