**Course**: High Performance Computing Lab

**PRN**: 22510034

**Name**: Om Kulkarni

**Batch**: B2

**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)

Fibonacci Computation:

**Source Code:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <omp.h>**

**#define N 20**

**int main() {**

**int fib[N];**

**fib[0] = 0;**

**fib[1] = 1;**

**double start, end;**

**start = omp\_get\_wtime();**

**#pragma omp parallel**

**{**

**// Step 1: Only one thread initializes first two terms**

**#pragma omp single**

**{**

**printf("Thread %d initialized first two terms\n", omp\_get\_thread\_num());**

**}**

**// Step 2: Parallel computation of Fibonacci sequence**

**#pragma omp for schedule(static) nowait**

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

**// Protect shared updates**

**#pragma omp critical**

**{**

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

**}**

**printf("Thread %d computed fib[%d] = %d\n", omp\_get\_thread\_num(), i, fib[i]);**

**}**

**// Step 3: Barrier before printing**

**#pragma omp barrier**

**// Step 4: Safe printing**

**#pragma omp for schedule(static)**

**for (int i = 0; i < N; i++) {**

**#pragma omp critical**

**{**

**printf("fib[%d] = %d\n", i, fib[i]);**

**}**

**}**

**}**

**end = omp\_get\_wtime();**

**printf("Execution Time: %f seconds\n", end - start);**

**return 0;**

**}**

Above code forces openMP parallelism, but really doesn’t do parallel computation a better version to parallelize fibonacci sequence is

#include <stdio.h>

#include <omp.h>

int fib\_task(int n) {

if (n < 2) return n;

int x, y;

#pragma omp task shared(x)

{

x = fib\_task(n - 1);

}

#pragma omp task shared(y)

{

y = fib\_task(n - 2);

}

#pragma omp taskwait

return x + y;

}

int main() {

int n = 20;

int result;

double start, end;

start = omp\_get\_wtime();

#pragma omp parallel

{

#pragma omp single

{

result = fib\_task(n);

}

}

end = omp\_get\_wtime();

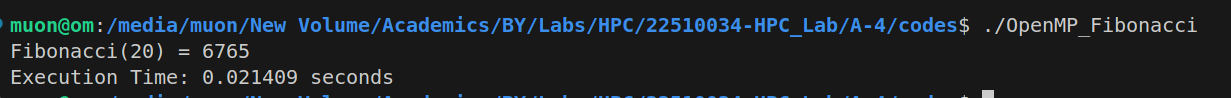
printf("Fibonacci(%d) = %d\n", n, result);

printf("Execution Time: %f seconds\n", end - start);

return 0;

}

**Output:**

****

**Analysis:**

Implemented parallel Fibonacci computation in C using OpenMP.

Used single for initialization, for with schedule(static) nowait for parallel work, critical for safe updates/printing, and barrier for synchronization.

Observed that synchronization ensures correctness but adds overhead, reducing speedup for small tasks.

Concluded that minimal synchronization gives better performance while maintaining correctness.

**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

**Source Code:**

**#include <stdio.h>**

**#include <stdlib.h>**

**#include <omp.h>**

**#define BUFFER\_SIZE 5**

**#define PRODUCE\_COUNT 10**

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

**int count = 0;**

**int main() {**

**double start, end;**

**start = omp\_get\_wtime();**

**#pragma omp parallel sections**

**{**

**// Producer Section**

**#pragma omp section**

**{**

**for (int i = 1; i <= PRODUCE\_COUNT; i++) {**

**int produced\_item = i;**

**int placed = 0;**

**while (!placed) {**

**#pragma omp critical**

**{**

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

**buffer[count] = produced\_item;**

**count++;**

**printf("Producer (Thread %d) produced item %d (Buffer count: %d)\n",**

**omp\_get\_thread\_num(), produced\_item, count);**

**placed = 1;**

**}**

**}**

**}**

**}**

**}**

**// Consumer Section**

**#pragma omp section**

**{**

**for (int i = 1; i <= PRODUCE\_COUNT; i++) {**

**int consumed\_item = 0;**

**int taken = 0;**

**while (!taken) {**

**#pragma omp critical**

**{**

**if (count > 0) {**

**consumed\_item = buffer[count - 1];**

**count--;**

**printf("Consumer (Thread %d) consumed item %d (Buffer count: %d)\n",**

**omp\_get\_thread\_num(), consumed\_item, count);**

**taken = 1;**

**}**

**}**

**}**

**}**

**}**

**}**

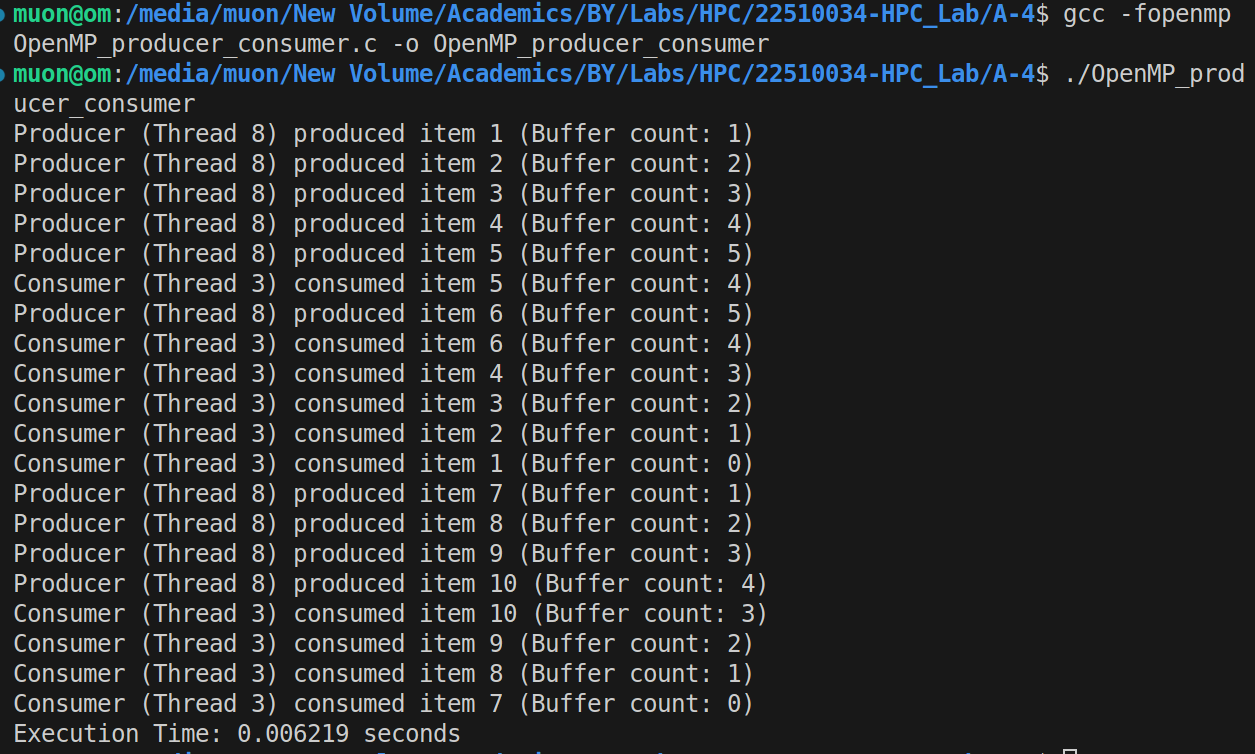
**end = omp\_get\_wtime();**

**printf("Execution Time: %f seconds\n", end - start);**

**return 0;**

**}**

**Output:**

**  
Analysis:**

Without critical, multiple threads could modify count or buffer at the same time, causing data corruption.

For a small buffer, synchronization overhead dominates.

The while loops simulate blocking behavior until a slot/item is available.

OpenMP is not ideal for producer–consumer compared to message queues, but works for demonstration.