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

**Year:** 2024-25 **Semester:** 1

**Course:** High Performance Computing Lab

**Practical No. 4**

**Exam Seat No: 21510036**

**Title of practical:**

Study and Implementation of Synchronization

**Problem Statement 1:**

# Analyse 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:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int fibonacci(int *n*) {

if (*n* <= 1) return *n*;

int x, y;

#pragma omp parallel sections

{

#pragma omp section

x = fibonacci(*n* - 1);

#pragma omp section

y = fibonacci(*n* - 2);

}

return x + y;

}

int main() {

int n = 23;

int result;

double start\_time = omp\_get\_wtime();

#pragma omp parallel

{

#pragma omp single

result = fibonacci(n);

}

double end\_time = omp\_get\_wtime();

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

printf("Runtime: %f seconds\n", end\_time - start\_time);

return 0;

}

Output -



**Information:**

* Parallel Sections:
* #pragma omp parallel sections creates multiple threads to compute fibonacci(n - 1) and fibonacci(n - 2) in parallel.
* The #pragma omp section directives define the separate sections that can be executed in parallel.
* Single Directive:
* #pragma omp single ensures that only one thread executes the fibonacci function call.
* Recursive Calls:
* The function is recursively called, but the computation of fibonacci(n - 1) and fibonacci(n - 2) is done in parallel sections.

**Problem Statement 2:**

## Producer Consumer Problem

**Screenshots:**

#include <stdio.h>

#include <stdlib.h>

#include <time.h>

#include <omp.h>

#define BUFFER\_SIZE 5

#define NITER 10

int buffer[BUFFER\_SIZE];

int in = 0; *// next free position*

int out = 0; *// first full position*

int main() {

srand(time(NULL));

#pragma omp parallel sections

{

#pragma omp section

{

for (int i = 0; i < NITER; i++) {

int next\_produced = rand() % 100 + 1;

*// Wait for an empty slot in the buffer*

while (((in + 1) % BUFFER\_SIZE) == out) {

#pragma omp flush(out) *// Ensures visibility of changes in `out`*

}

*// Critical section to safely access shared buffer*

#pragma omp critical

{

buffer[in] = next\_produced;

printf("Producer produced [%d]. (Placed in index: in=%d, out=%d)\n", next\_produced, in, out);

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

#pragma omp flush(in) *// to ensure visibility of changes in `in`*

}

}

}

#pragma omp section

{

for (int i = 0; i < NITER; i++) {

*// Wait for data to be available in the buffer*

while (in == out) {

#pragma omp flush(in)

}

#pragma omp critical

{

int next\_consumed = buffer[out];

printf("\t\tConsumer consumed [%d]. (in=%d, Consumed from index: out=%d)\n", next\_consumed, in, out);

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

#pragma omp flush(out)

}

}

}

}

return 0;

}

**Output -**



**Information:**

**Used** OpenMP's critical, atomic, and barrier directives to manage synchronization.

* #pragma omp critical: Ensures that only one thread (either producer or consumer) can access the critical section of code at a time, protecting shared resources like the buffer, in, and out indices.
* #pragma omp flush: Ensures memory visibility across threads. It flushes the changes made by one thread so that other threads can see the updated values of shared variables.
* Busy-Waiting Minimization: While busy-waiting is still used to some extent, the use of #pragma omp flush minimizes the time spent in this state by ensuring up-to-date checks on the buffer's status.

**Github Link:**