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

**Year:** 2022-23 **Semester:**1

**Course:** High Performance Computing Lab

**Practical No. 4**

**Exam Seat No:**

2019BTECS00005 – Ashish Sutar

**Title of practical:**

Study and implementation of Synchronization constructs

**Problem Statement 1:**

Implement a parallel code for Fibonacci Series using Dynamic Programming

**Screenshot 1:**

// Fib

#include <stdio.h>

#include <time.h>

#include <omp.h>

int fib(int n)

{

    /\* Declare an array to store Fibonacci numbers. \*/

    int a[n + 2];

    // 1 extra to handle case, n = 0

    int i;

    /\* 0th and 1st number oa the series are 0 and 1\*/

    a[0] = 0;

    a[1] = 1;

#pragma omp parallel

    {

#pragma omp single

        for (i = 2; i <= n; i++)

        {

            a[i] = a[i - 2] + a[i - 1];

            printf("id of thread involved in the computation of fib no %d is=%d\n", i + 1, omp\_get\_thread\_num());

        }

#pragma omp barrier

#pragma omp single

        {

            printf("the elements of fib series are\n");

            for (i = 0; i < n; i++)

                printf("%d,id of the thread displaying this no is =  %d\n", a[i], omp\_get\_thread\_num());

        }

    }

    return a[n];

}

int main()

{

    int n = 10;

    clock\_t begin = clock();

    double time1 = omp\_get\_wtime();

    int ans = fib(n);

    printf("%d\n", ans);

    double time2 = omp\_get\_wtime();

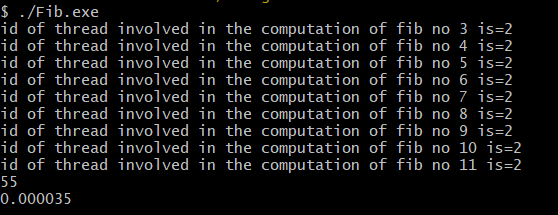
    double time\_requried = time2-time1;

    printf("%lf\n", time\_requried);

    return 0;

}

**Screenshot 2:**



**Information 1:**

Ordered directive refers to the in-order execution of a for loop, it must be within the scope of a for or parallel for directive. Furthermore, the for or parallel for directive must have the ordered clause specified to indicate that the loop contains an ordered block.

**Problem Statement 2:**

Implement a parallel code for Producer Consumer Problem

**Screenshot 3:**

#include <stdio.h>

#include <stdlib.h>

#include <omp.h>

int mutex = 1;

int full = 0;

int empty = 10, x = 0;

void producer()

{

    --mutex;

    ++full;

    --empty;

    x++;

    printf("\nProducer produces"

           "item %d",

           x);

    ++mutex;

}

void consumer()

{

    --mutex;

    --full;

    ++empty;

    printf("\nConsumer consumes "

           "item %d",

           x);

    x--;

    ++mutex;

}

int main()

{

    int n, i;

    printf("\n1. Press 1 for Producer"

           "\n2. Press 2 for Consumer"

           "\n3. Press 3 for Exit");

#pragma omp critical

    for (i = 1; i > 0; i++)

    {

        printf("\nEnter your choice:");

        scanf("%d", &n);

        switch (n)

        {

        case 1:

            if ((mutex == 1) && (empty != 0))

            {

                producer();

            }

            else

            {

                printf("Buffer is full!");

            }

            break;

        case 2:

            if ((mutex == 1) && (full != 0))

            {

                consumer();

            }

            else

            {

                printf("Buffer is empty!");

            }

            break;

        case 3:

            exit(0);

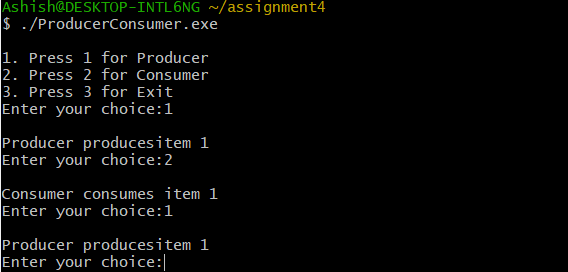
            break;

        }

    }

}

**Screenshot 4:**

****

**Information 2:**

The critical directive ensures that at any point in the execution of the program, only one thread is within a critical section specified by a certain name. If a thread is already inside a critical section (with a name), all others must wait until it is done before entering the named critical section. The name field is optional. If no name is specified, the critical section maps to a default name that is the same for all unnamed critical sections. The names of critical sections are global across the program.

**Github Link:**