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

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

**Course:** High Performance Computing Lab

**Practical No. 4**

**Exam Seat No: 21510074**

**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 <omp.h>

#define THRESHOLD 30

#define DISPLAY\_LIMIT 10

long long fibonacci\_sequential(int n) {

    if (n <= 1) return n;

    return fibonacci\_sequential(n - 1) + fibonacci\_sequential(n - 2);

}

long long fibonacci\_parallel(int n) {

    if (n <= THRESHOLD) {

        return fibonacci\_sequential(n);

    }

    long long x, y;

    #pragma omp task shared(x)

    x = fibonacci\_parallel(n - 1);

    #pragma omp task shared(y)

    y = fibonacci\_parallel(n - 2);

    #pragma omp taskwait

    return x + y;

}

void print\_fibonacci\_sequence(int n, long long result) {

    printf("Fibonacci Sequence (first %d numbers):\n", DISPLAY\_LIMIT);

    long long a = 0, b = 1;

    for (int i = 0; i < DISPLAY\_LIMIT && i <= n; i++) {

        if (i == n) {

            printf("%lld", result);

        } else if (i < 2) {

            printf("%lld", i);

        } else {

            long long temp = a + b;

            a = b;

            b = temp;

            printf("%lld", temp);

        }

        if (i < DISPLAY\_LIMIT - 1 && i < n) printf(", ");

    }

    if (n >= DISPLAY\_LIMIT) printf(", ...");

    printf("\n");

}

int main() {

    int n = 45;

    double start\_time, end\_time;

    printf("Computing Fibonacci(%d)\n\n", n);

    // Sequential computation

    start\_time = omp\_get\_wtime();

    long long result\_seq = fibonacci\_sequential(n);

    end\_time = omp\_get\_wtime();

    printf("Sequential Result:\n");

    print\_fibonacci\_sequence(n, result\_seq);

    printf("F(%d) = %lld\n", n, result\_seq);

    printf("Sequential Time: %.6f seconds\n\n", end\_time - start\_time);

    // Parallel computation

    start\_time = omp\_get\_wtime();

    long long result\_par;

    #pragma omp parallel

    {

        #pragma omp single

        {

            result\_par = fibonacci\_parallel(n);

        }

    }

    end\_time = omp\_get\_wtime();

    printf("Parallel Result:\n");

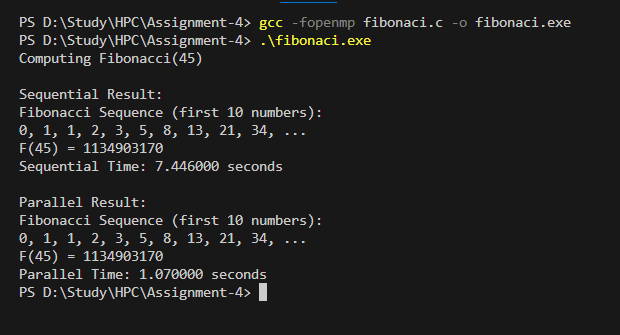
    print\_fibonacci\_sequence(n, result\_par);

    printf("F(%d) = %lld\n", n, result\_par);

    printf("Parallel Time: %.6f seconds\n", end\_time - start\_time);

    return 0;

}

****

**Information:**

We define a THRESHOLD value to determine when to switch from parallel to sequential computation.

fibonacci\_sequential is a standard recursive implementation of Fibonacci.

fibonacci\_parallel is our parallel implementation:

If n is less than or equal to THRESHOLD, we call the sequential version to avoid excessive task creation for small computations.

For larger values, we use #pragma omp task to create tasks for computing F(n-1) and F(n-2).

We use shared(x) and shared(y) clauses to ensure that the results are visible to the parent task.

#pragma omp taskwait is used to wait for both child tasks to complete before returning the sum.

In the main function:

We first compute Fibonacci sequentially and measure the time.

For parallel computation, we use #pragma omp parallel to create a team of threads.

Inside the parallel region, we use #pragma omp single to ensure that only one thread creates the initial task.

We use omp\_get\_wtime() to measure the execution time for both sequential and parallel versions.

This implementation demonstrates the use of several OpenMP constructs:

#pragma omp parallel: Creates a team of threads.

#pragma omp single: Ensures that a section of code is executed by only one thread.

#pragma omp task: Creates a task that can be executed by any thread in the team.

#pragma omp taskwait: Waits for all child tasks to complete.

The shared clause is used to specify that variables should be shared among tasks.

This implementation achieves parallelism through task-based parallelism, which is well-suited for recursive algorithms like Fibonacci. The use of a threshold helps to balance the overhead of task creation with the benefits of parallelism.

Keep in mind that the actual speedup will depend on various factors, including the number of available cores, the specific OpenMP implementation, and the chosen threshold value. You may need to experiment with different threshold values to find the optimal performance for your specific system.

**Problem Statement 2:**

# Analyse 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

**Screenshots:**

**Information:**

**Github Link:**