Walchand College of Engineering, Sangli
Department of Computer Science and Engineering

Class: Final Year (Computer Science and Engineering)

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

Course: High Performance Computing Lab

Practical No. 4

Exam Seat No: 22520007

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)

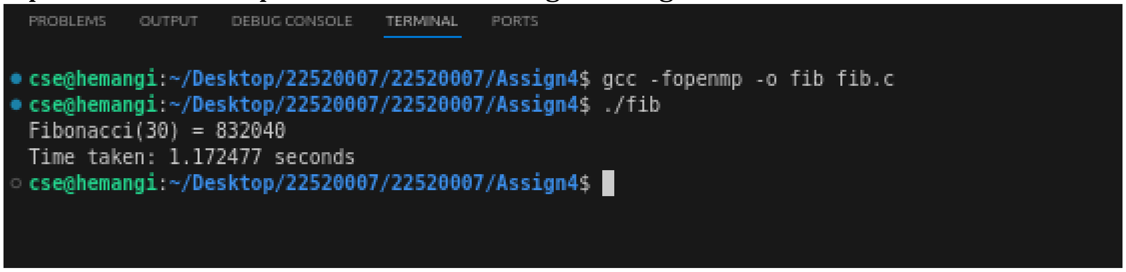
Fibonacci Computation:

Screenshots:

fib.c

Assign4 > fib.c > ...

```c
#include <stdio.h>
#include <omp.h>

int fibonacci(int n) {
    if (n <= 1) return n;

    int x, y;

    #pragma omp task shared(x)
    x = fibonacci(n - 1);

    #pragma omp task shared(y)
    y = fibonacci(n - 2);

    #pragma omp taskwait // Wait for both tasks to complete before proceeding
    return x + y;
}

int main() {
    int n = 30;
    int fib;

    double start_time = omp_get_wtime(); // Start timing

    #pragma omp parallel
    {
        #pragma omp single
        {
            fib = fibonacci(n);
        }
    }

    double end_time = omp_get_wtime(); // End timing

    printf("Fibonacci(%d) = %d\n", n, fib);
    printf("Time taken: %f seconds\n", end_time - start_time);

    return 0;
}
```



Analysis

1. **Parallelism and Task Overhead**:

    - **Task Creation**: The use of `#pragma omp task` creates multiple tasks, which may lead to high overhead due to the creation and management of these tasks.
    - **Task Synchronization**: The `#pragma omp taskwait` ensures that the main thread waits for all created tasks to complete. This is necessary but can introduce additional overhead.

2. **Performance Considerations:**

    - **Scalability**: While OpenMP allows parallel execution, the recursive approach does not scale well for large `n` due to the exponential growth in the number of tasks.
    - **Efficiency**: Recursive approaches with OpenMP are generally inefficient for Fibonacci computation due to redundant calculations. For large `n`, the performance might degrade rapidly.

3. **Practical Limitations:**

    - **Segmentation Faults**: As noted in your earlier issue with larger values of `n`, very deep recursion and excessive task creation can lead to segmentation faults due to stack overflow.

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:

```
b.c > producer()
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int full = 0;
int empty = 10, x = 0;
omp_lock_t lock;

void producer()
{
    omp_set_lock(&lock);
    if (empty > 0)
    {
        full++;
        empty--;
        x++;
        printf("\nProducer produces item %d\n", x);
    }
    else
    {
        printf("\nBuffer is full!\n");
    }
    omp_unset_lock(&lock);
}

void consumer()
{
    omp_set_lock(&lock);
    if (full > 0)
    {
        full--;
        empty++;
        printf("\nConsumer consumes item %d\n", x);
        x--;
    }
    else
```

```c
    else
    {
        printf("\nBuffer is empty!\n");
    }
    omp_unset_lock(&lock);
}

int main()
{
    int n;

    omp_init_lock(&lock);

    while (1)
    {
        printf(
            "\n1. Press 1 for Producer"
            "\n2. Press 2 for Consumer"
            "\n3. Press 3 for Exit");

        printf("\nEnter your choice: ");
        scanf("%d", &n);

        switch (n)
        {
        case 1:
#pragma omp task
        {
            producer();
        }
        break;

        case 2:
#pragma omp task
        {
            consumer();
        }
        break;

        case 3:
            omp_destroy_lock(&lock);
            exit(0);

        default:
            printf("\nInvalid choice! Please try again.");
            break;
        }
    }

    return 0;
}
```

```
1. Press 1 for Producer
2. Press 2 for Consumer
3. Press 3 for Exit
Enter your choice:1

Producer produces item 1

Enter your choice:1

Producer produces item 2

Enter your choice:1

Producer produces item 3

Enter your choice:1

Producer produces item 4

Enter your choice:1

Producer produces item 5

Enter your choice:1

Producer produces item 6

Enter your choice:1

Producer produces item 7

Enter your choice:1

Producer produces item 8

Enter your choice:1

Producer produces item 9

Enter your choice:1

Producer produces item 10

Enter your choice:1

Buffer is full!

Enter your choice:
```

```
Enter your choice:1
Buffer is full!
Enter your choice:2
Consumer consumes item 10
Enter your choice:2
Consumer consumes item 9
Enter your choice:2
Consumer consumes item 8
Enter your choice:2
Consumer consumes item 7
Enter your choice:2
Consumer consumes item 6
Enter your choice:2
Consumer consumes item 5
Enter your choice:2
Consumer consumes item 4
Enter your choice:2
Consumer consumes item 3
Enter your choice:2
Consumer consumes item 2
Enter your choice:2
Consumer consumes item 1
Enter your choice:2
Buffer is empty!
Enter your choice:2
Buffer is empty!
Enter your choice:2
```

Github Link: