Assignment No. 4

PRN: 2020BTECS00025
Name: Sahil Santosh Otari
Course: High Performance Computing Lab
Title of practical: Study and implementation of synchronization.
Q1: 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:
Source Code:

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

int fibonacci(int n){
    int i, j;
    if (n < 2)
        return n;
    else{
#pragma omp task shared(i)
        i = fibonacci(n - 1);
#pragma omp task shared(j)
        j = fibonacci(n - 2);

#pragma omp taskwait
        return i + j;
    }
}
int main(int argc, char **argv){
    char *a = argv[1];
    int n = atoi(a), result;
#pragma omp parallel
    {
```

```
#pragma omp single


        result = fibonacci(n);
    }
    printf("Result is %d\n", result);
}
```

Output:

```
PS D:\HPC\Assignment4> gcc -o .\fibonacci -fopenmp .\fibonacci.cpp
PS D:\HPC\Assignment4> .\fibonacci
Result is 0
PS D:\HPC\Assignment4> .\fibonacci 10
Result is 55
PS D:\HPC\Assignment4>
```

Information and Analysis:

This C code uses OpenMP to parallelize the computation of the Fibonacci sequence recursively. Here's a brief analysis and information:

Parallelization (OpenMP): The code leverages OpenMP's task parallelism to compute the Fibonacci sequence efficiently. It decomposes the problem into smaller tasks and computes them concurrently.

Fibonacci Function: The `fibonacci` function is a recursive function to calculate the Fibonacci number for a given input `n`. It uses OpenMP tasks to parallelize the computation of Fibonacci values for `n-1` and `n-2`.

Taskwait: The `#pragma omp taskwait` directive ensures that the two tasks (for `n-1` and `n-2`) complete before proceeding to compute the final result.

Input: The value of `n` is taken as a command-line argument (provided through `argv`) and converted to an integer for calculation.

Parallel Region: The code creates a single OpenMP parallel region using `#pragma omp parallel`, and within this region, a single thread (the master thread) initiates the calculation by calling `fibonacci`.

Q2: Analyse and implement a Parallel code for below programs using Open MP considering

synchronization requirements. (Demonstrate the use of different clauses and constructs
wherever applicable).

Producer Consumer Problem:

Using critical clause, which allows only one thread to execute part of a program.

Source Code:

```c
#include <stdio.h>
#include <stdlib.h>
// Initialize a mutex to 1
int mutex = 1;
// Number of full slots as 0a
int full = 0;
// Number of empty slots as size
// of buffer

int empty = 10, x = 0;
// Function to produce an item and
// add it to the buffer
void producer()
{
    // Decrease mutex value by 1
    --mutex;
    // Increase the number of full
    // slots by 1
    ++full;
    // Decrease the number of empty
    // slots by 1
    --empty;
    // Item produced
    x++;
    printf("\nProducer produces item %d",x);
    // Increase mutex value by 1
    ++mutex;
```

```c
}
// Function to consume an item and
// remove it from buffer
void consumer()
{

    // Decrease mutex value by 1
    --mutex;
    // Decrease the number of full
    // slots by 1
    --full;
    // Increase the number of empty
    // slots by 1
    ++empty;
    printf("\nConsumer consumes item %d",x);
    x--;
    // Increase mutex value by 1
    ++mutex;
}
// Driver Code
int main()
{
    int n, i;
    printf("\n1. Press 1 for Producer"
           "\n2. Press 2 for Consumer"
           "\n3. Press 3 for Exit");
// Using '#pragma omp parallel for'
// can give wrong value due to
// synchronization issues.
// 'critical' specifies that code is
// executed by only one thread at a
// time i.e., only one thread enters
// the critical section at a given time
```

```c
#pragma omp critical
    for (i = 1; i > 0; i++)
    {
        printf("\nEnter your choice:");
        scanf("%d", &n);

        // Switch Cases
        switch (n)
        {
        case 1:
            // If mutex is 1 and empty
            // is non-zero, then it is
            // possible to produce
            if ((mutex == 1) && (empty != 0))
            {
                producer();
            }
            // Otherwise, print buffer
            // is full
            else
            {
                printf("Buffer is full!");
            }
            break;
        case 2:
            // If mutex is 1 and full
            // is non-zero, then it is
            // possible to consume
            if ((mutex == 1) && (full != 0))
            {
                consumer();
            }
            // Otherwise, print Buffer
```

```c
                // is empty
            else
            {
                printf("Buffer is empty!");
            }
            break;
        // Exit Condition
        case 3:
            exit(0);
            break;
        }
    }
}
```

Output:

```
PS D:\HPC\Assignment4> .\producerConsumer

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:2

Consumer consumes item 7
Enter your choice:2

Consumer consumes item 6
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:3
PS D:\HPC\Assignment4> █
```

Information and Analysis:

This C code is an example of a simple producer-consumer problem implemented using a mutex and OpenMP. Here's some information and analysis of the code:

- **Producer-Consumer Problem:** The code simulates the classic producer-consumer problem, where a producer produces items and adds them to a buffer, while a consumer consumes items from the same buffer. The goal is to ensure that the producer doesn't add items when the buffer is full and the consumer doesn't consume items when the buffer is empty.
- **Shared Variables:** The code uses shared integer variables `mutex`, `full`, `empty`, and `x` to manage access to the buffer and keep track of the number of

items in it. `mutex` is used to ensure that only one thread can access the critical sections of the producer and consumer at a time.

- **Producer Function:** The `producer` function is responsible for producing an item, updating the buffer state, and printing a message. It decreases `mutex`, increases `full`, and decreases `empty`.
- **Consumer Function:** The `consumer` function consumes an item from the buffer, updates the buffer state, and prints a message. It also uses the `mutex` variable to ensure exclusive access to the critical section.
- **User Interaction:** The code uses a loop to continuously prompt the user for input, allowing them to choose whether to run the producer or consumer or exit the program.
- **Open MP Critical Section:** A `#pragma omp critical` directive is used to ensure that only one thread at a time can execute the code within the loop that reads user input and invokes producer or consumer functions. This avoids race conditions in user input handling.
- **Switch Cases**: The code uses a switch-case structure to handle user input. Depending on the input, it either runs the producer, consumer, or exits the program.
- **Exit Condition:** The user can exit the program by selecting option 3.
- **Synchronization:** The `mutex` variable is used for synchronization to prevent multiple threads from concurrently modifying the shared variables and buffer.
- **Buffer Handling:** If the buffer is full when the producer is called or empty when the consumer is called, appropriate error messages are displayed.
- Overall, it demonstrates a basic implementation of the producer-consumer problem with a simple buffer management mechanism and uses Open MP for thread synchronization. It's a simplified illustration and doesn't cover more advanced aspects of concurrent programming like thread safety or handling more complex data structures.