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

Class: Final Year (Computer Science and Engineering)

Year: 2023-24 **Semester:** 1

Course: High Performance Computing Lab

Practical No. 4

Exam Seat No: 2020BTECS00041

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:

```
void parallel_fib(int n) {
    if (n < 2) {
        fib[n] = n;
        return;
    }

    #pragma omp task shared(fib)
    // printf("task1 by %d\n",n-1);
    parallel_fib(n - 1);

    #pragma omp task shared(fib)
    // printf("task2 by %d\n",n-2);
    parallel_fib(n - 2);

    #pragma omp taskwait

    fib[n] = fib[n - 1] + fib[n - 2];
}
```

```c
int main() {
    int n = N; // Calculate Fibonacci(N)

    clock_t start=clock();
        // printf("No. of threads: %d\n",omp_get_num_threads());
    #pragma omp parallel
    {
        #pragma omp single
        parallel_fib(n);
    }
    clock_t end=clock();

    printf("Fibonacci(%d) = %lld\n", n, fib[n]);
    printf("Time: %f",(double)(end-start)/CLOCKS_PER_SEC);

    return 0;
}
```

Screenshots:

```
PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
Time: 0.112000
PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe
Fibonacci(15) = 610
Time: 0.015000
PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe
Fibonacci(10) = 55
Time: 0.002000
PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe
Fibonacci(5) = 5
Time: 0.001000
PS D:\FinalYear\HPCL\Assignment 4>
```

Without single clause

```
#pragma omp parallel
{
    #pragma omp
    parallel_fib(n);
}
```

```
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe
Fibonacci(5) = 5
Time: 0.001000
PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe
Fibonacci(10) = 55
Time: 0.000000
PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe
Fibonacci(15) = 610
Time: 0.085000
PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe
Fibonacci(20) = 6765
Time: 0.901000
PS D:\FinalYear\HPCL\Assignment 4>
```

With 5 threads

```
#pragma omp parallel num_threads(5)
{
    #pragma omp
    parallel_fib(n);
}
```

Using 5 threads

```
PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe
Fibonacci(10) = 55
Time: 0.005000
```

Using fib as a private variable

```
PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe
Fibonacci(10) = 55
Time: 0.000000
```

Information:

The Fibonacci series implementation using various OpenMP clauses include shared, private, single, task, num_threads(), taskwait. The #pragma omp directive communicates with the compiler to execute the program parallelly .The shared clause uses the same copy of variable between all threads while the private clause uses the different copies for all threads. Single clause in OpenMp uses one thread for execution of the program. If no depend clause is present on the taskwait construct, the current task region is suspended at an implicit task scheduling point associated with the construct. The current task region remains suspended until all child tasks that it generated before the taskwait region complete execution. The time taken for execution of program is less when more number of threads are used while the time taken by the program with 'single' clause is more. Large input requires more time then small input. The num_threads() is used to set the number of threads.

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

```
#include <stdio.h>
#include <stdlib.h>
#include <time.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");
    clock_t start=clock();
    #pragma omp parallel
    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;
        }
    }
    clock_t end=clock();
    printf("Time taken : %f\n",(double)(end-start)/CLOCKS_PER_SEC);
}
```

Screenshots:

```
PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q2.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe

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:2

Consumer consumes item 2
Enter your choice:2

Consumer consumes item 1
Enter your choice:2
Buffer is empty!
Enter your choice:3
```

Issue obtained using “#pragma omp parallel”

```
PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q2.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe

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

Producer produces item 1
Enter your choice:1

Producer produces item 2
Enter your choice:2
```

Information:

The Producer-Consumer problem is a classic synchronization problem where there are two types of processes: producers and consumers, sharing a common, fixed-size buffer or queue. Producers produce data items and put them into the buffer, while consumers consume items from the buffer. There are synchronization requirements to ensure that producers do not produce data when the buffer is full and that consumers don't consume data when the buffer is empty.

Using '#pragma omp parallel for' can give wrong value due to synchronization issues such as increase in cache locality and parallelization overheads. Synchronization imposes order constraints and is used to protect access to shared data. High level synchronization is achieved using clauses like critical, atomic, barrier, ordered while low level synchronization is carried out using flush and locks. '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. More than 1 thread can work while using “#pragma omp parallel”. Thus “critical” clause is suitable for problems consisting deadlock conditions or critical regions.

Github Link: https://github.com/Nikita226/HPCL/tree/main/A4