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:

Sequential

```
#include <stdio.h>
#include <omp.h>
#include <time.h>
#define N 10
long long fib[N];
void parallel_fib(int n)
{
    if (n < 2)
    {
        fib[n] = n;
        return;
    }
    parallel_fib(n - 1);
    parallel_fib(n - 2);
    fib[n] = fib[n - 1] + fib[n - 2];
}
int main()
{
    int n = N;
    clock_t start=clock();
    #pragma omp parallel num_threads(5)
    {
        #pragma omp
```

```
        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;
}
```

Output of Sequential:

```
PS D:\FinalYear\HPCL\Assignment 4> gcc q1sequential.cpp
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe

Fibonacci(10) = 55    Time: 0.000000

PS D:\FinalYear\HPCL\Assignment 4> gcc q1sequential.cpp
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe

Fibonacci(20) = 6765    Time: 0.000000

PS D:\FinalYear\HPCL\Assignment 4> gcc q1sequential.cpp
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe

Fibonacci(30) = 832040    Time: 0.009000

PS D:\FinalYear\HPCL\Assignment 4> gcc q1sequential.cpp
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe

Fibonacci(40) = 102334155    Time: 0.752000
```

Parallel:

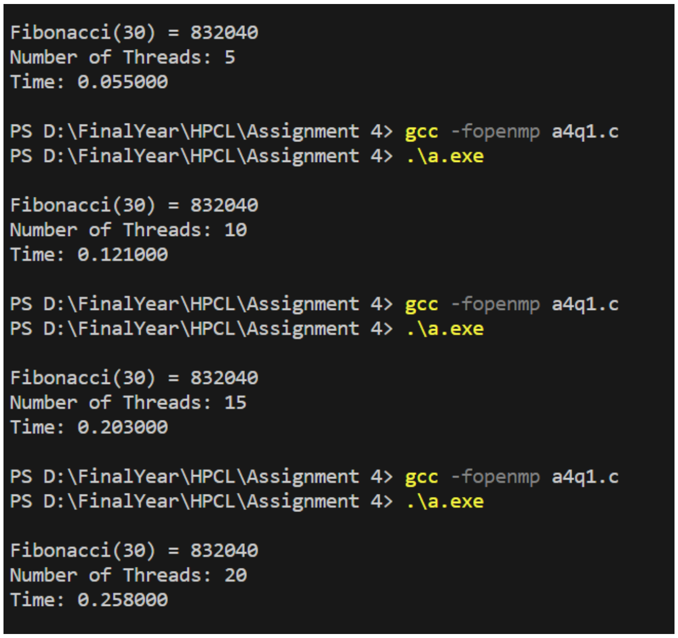
```
#include <stdio.h>
#include <omp.h>
#include <time.h>
#define N 10
#define threads 20
long long fib[N];
void parallel_fib(int n)
{
    if (n < 2)
    {
        fib[n] = n;
        return;
    }
    parallel_fib(n - 1);
    parallel_fib(n - 2);
    fib[n] = fib[n - 1] + fib[n - 2];
```

```
}
int main()
{
    int n = N;
    clock_t start=clock();
    #pragma omp parallel num_threads(threads)
    {
        #pragma omp
        parallel_fib(n);
    }
    clock_t end=clock();
    printf("\nFibonacci(%d) = %lld", n, fib[n]);
    printf("\nNumber of Threads: %d", threads);
    printf("\nTime: %f\n\n",(double)(end-start)/CLOCKS_PER_SEC);
    return 0;
}
```

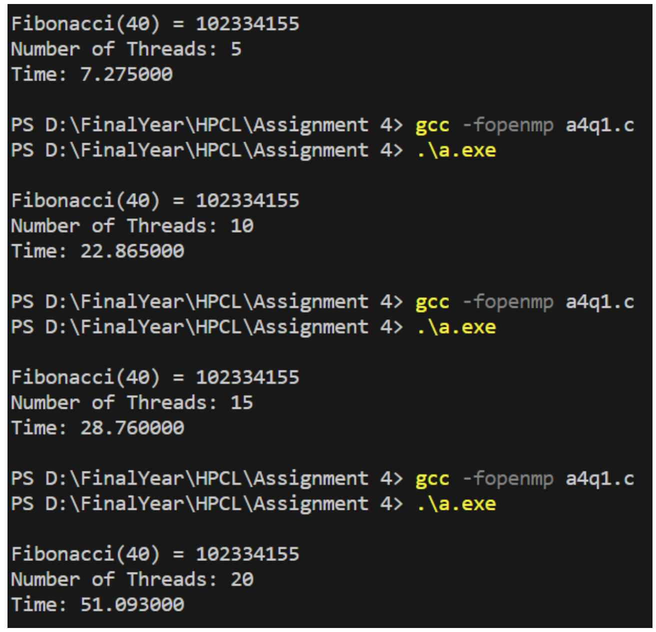
Output:

```
Fibonacci(10) = 55
Number of Threads: 5
Time: 0.000000

PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe

Fibonacci(10) = 55
Number of Threads: 10
Time: 0.006000

PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe

Fibonacci(10) = 55
Number of Threads: 15
Time: 0.003000

PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe

Fibonacci(10) = 55
Number of Threads: 20
Time: 0.007000
```

```
Fibonacci(20) = 6765
Number of Threads: 5
Time: 0.002000

PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe

Fibonacci(20) = 6765
Number of Threads: 10
Time: 0.006000

PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe

Fibonacci(20) = 6765
Number of Threads: 15
Time: 0.002000

PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe

Fibonacci(20) = 6765
Number of Threads: 20
Time: 0.008000
```



```
Fibonacci(30) = 832040
Number of Threads: 5
Time: 0.055000

PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe

Fibonacci(30) = 832040
Number of Threads: 10
Time: 0.121000

PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe

Fibonacci(30) = 832040
Number of Threads: 15
Time: 0.203000

PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe

Fibonacci(30) = 832040
Number of Threads: 20
Time: 0.258000
```



```
Fibonacci(40) = 102334155
Number of Threads: 5
Time: 7.275000

PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe

Fibonacci(40) = 102334155
Number of Threads: 10
Time: 22.865000

PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe

Fibonacci(40) = 102334155
Number of Threads: 15
Time: 28.760000

PS D:\FinalYear\HPCL\Assignment 4> gcc -fopenmp a4q1.c
PS D:\FinalYear\HPCL\Assignment 4> .\a.exe

Fibonacci(40) = 102334155
Number of Threads: 20
Time: 51.093000
```

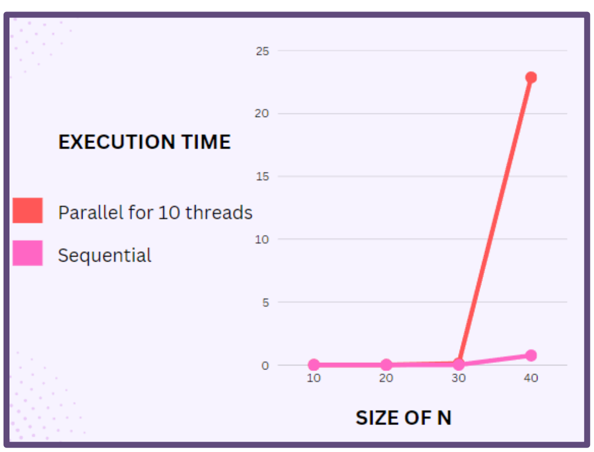
Value of N (Fibonacci series nth term)	Sequential Execution Time
10	0.000
20	0.000
30	0.009
40	0.752

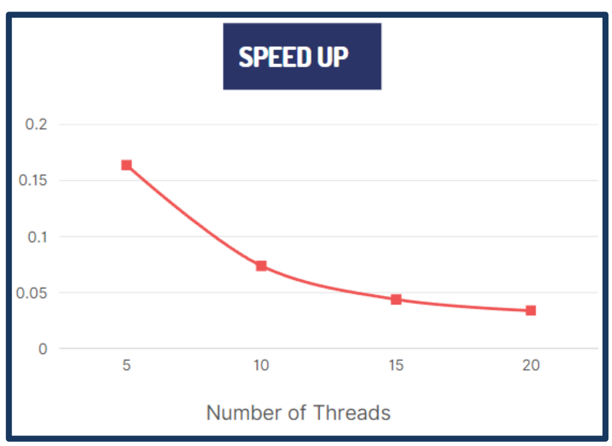
N \ threads	5	10	15	20
10	0.000	0.006	0.003	0.007
20	0.002	0.006	0.002	0.008
30	0.055	0.121	0.203	0.258
40	7.275	22.865	28.760	51.093

Value of N = 30 constant

Threads	Speed Up
5	0.164
10	0.074
15	0.044
20	0.034

Here due to excess execution time, execution time required for parallel program is much more than the sequential program hence speed up is less than 1.





Problem Statement 2:
Analyse and implement Parallel code for below programs using OpenMP considering synchronization requirements. (Demonstrate use of different clauses & constructs wherever applicable) **Producer Consumer Problem**

```
#include <stdio.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 critical
    for (i = 1; i > 0; i++)
    {
        printf("\nEnter your choice:");
        scanf("%d", &n);
        switch (n) {
        case 1:
            if ((mutex == 1) && (empty != 0)) {
                producer();
```

```c
            }
            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:

```
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: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:1

Producer produces item 1
Enter your choice:3
```

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