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: 2020BTECS00042

Title of practical:

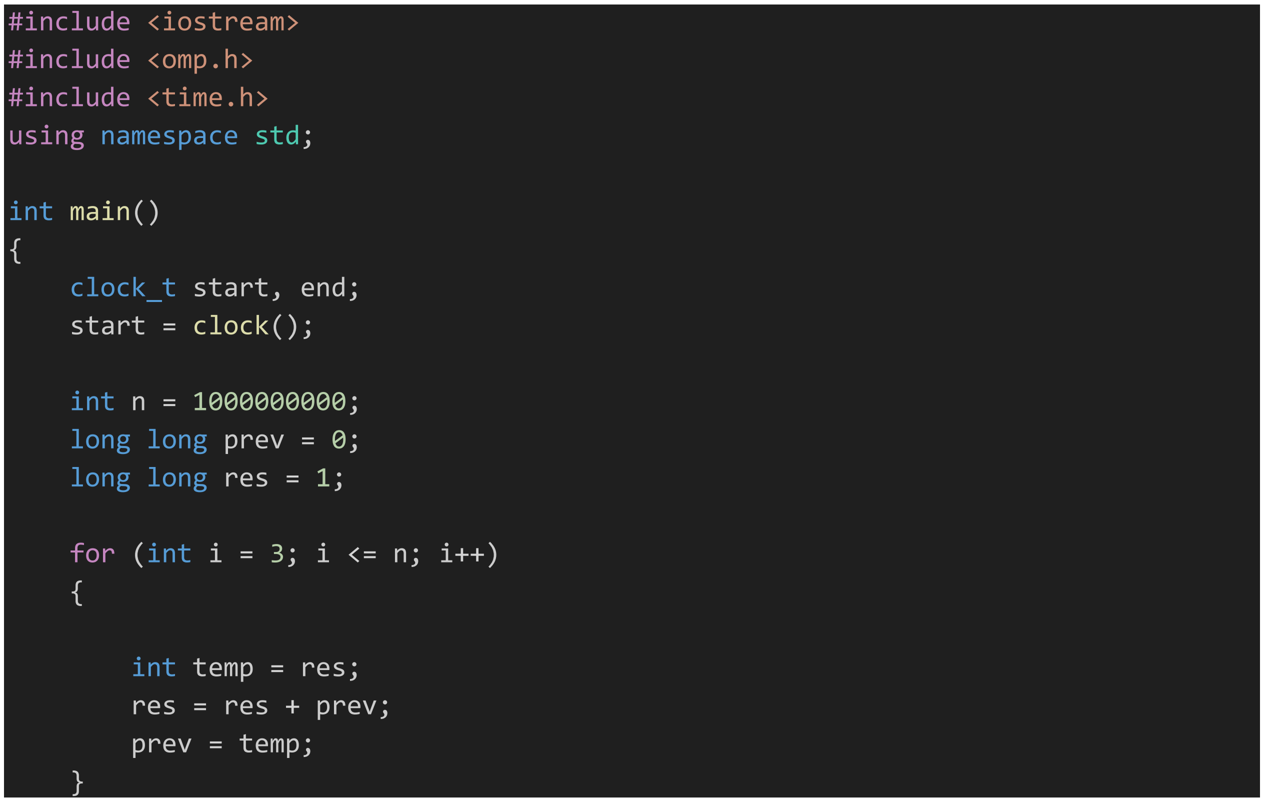
Study and Implementation of Synchronization

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)

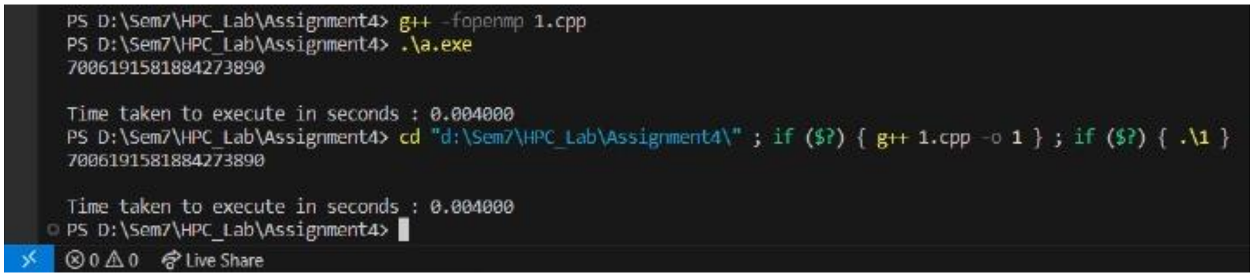
Fibonacci Computation:

Screenshots:



```
#include <iostream>
#include <omp.h>
#include <time.h>
using namespace std;

int main()
{
    clock_t start, end;
    start = clock();

    int n = 1000000000;
    long long prev = 0;
    long long res = 1;

    for (int i = 3; i <= n; i++)
    {

        int temp = res;
        res = res + prev;
        prev = temp;
    }
    cout << res << endl;
    end = clock();
    double duration = ((double)end - start) / CLOCKS_PER_SEC;
    printf("\nTime taken to execute in seconds : %f", duration);
    return 0;
}
```

Output:



```
PS D:\Sem7\HPC_Lab\Assignment4> g++ -fopenmp 1.cpp
PS D:\Sem7\HPC_Lab\Assignment4> .\a.exe
7006191581884273890

Time taken to execute in seconds : 0.004000
PS D:\Sem7\HPC_Lab\Assignment4> cd "d:\Sem7\HPC_Lab\Assignment4\" ; if ($?) { g++ 1.cpp -o 1 } ; if ($?) { .\1 }
7006191581884273890

Time taken to execute in seconds : 0.004000
PS D:\Sem7\HPC_Lab\Assignment4>
```

Information:
In Fibonacci series, calculation of the any term is series totally depends on the previous two results. So, we can’t calculate third term unless its previous two are calculated. Thus, it shows the pure dependency with previous iterations and need of sequential calculations. We can’t perform these operations parallelly. Parallelism can get into the picture, when our series is too large, and we are performing and storing the results after some intervals.

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)

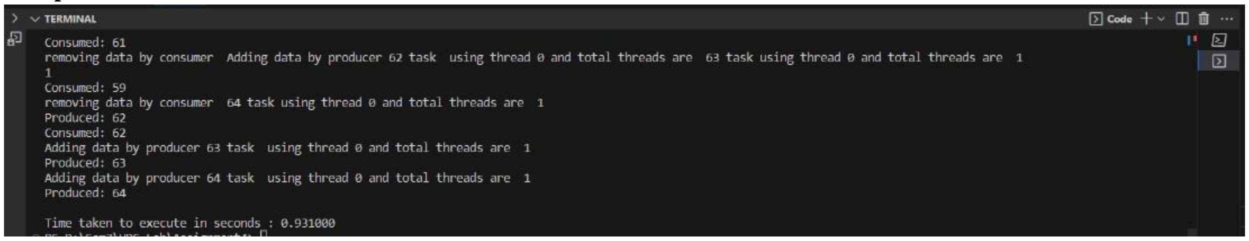
Producer Consumer Problem

Screenshots:

```
#include <stdio.h>
#include <omp.h>
#include <time.h>
#include <iostream>
using namespace std;
#define BUFFER_SIZE 1000
```

```
int buffer[BUFFER_SIZE];
int itemCount = 0;
void producer()
{
#pragma omp parallel
    {
        for (int i = 0; i < 100; ++i)
        {
            cout << "Adding data by producer " << i << " using thread
" << omp_get_thread_num() << " and total threads are  " <<
omp_get_num_threads() << endl;
#pragma omp task
            {
                cout << "Adding data by producer " << i << "
task  using thread " << omp_get_thread_num() << " and total threads
are  " << omp_get_num_threads() << endl;
#pragma omp critical
                {
                    if (itemCount < BUFFER_SIZE)
                    {
                        buffer[itemCount++] = i;
                        printf("Produced: %d\n", i);
                    }
                }
            }
        }
    }
}
void consumer()
{
#pragma omp parallel
    {
        for (int i = 0; i < 100; ++i)
        {
            cout << "removing data by consumer " << i << " using 
thread " << omp_get_thread_num() << " and total threads are  " << 
omp_get_num_threads() << endl;
#pragma omp task
            {
                cout << "removing data by consumer  " << i << " task 
using thread " << omp_get_thread_num() << " and total threads are  " 
<< omp_get_num_threads() << endl;
#pragma omp critical
                {
                    if (itemCount > 0)
                    {
                        int consumedItem = buffer[--itemCount];
                        printf("Consumed: %d\n", consumedItem);
                    }
                }
            }
        }
    }
}
int main()
{
    omp_set_num_threads(4);

    clock_t start, end;
    start = clock();

#pragma omp parallel sections
    {
#pragma omp section
        {
            producer();
        }

#pragma omp section
        {
            consumer();
        }
    }

    end = clock();
    double duration = ((double)end - start) / CLOCKS_PER_SEC;
    printf("\nTime taken to execute in seconds : %f", duration);
    return 0;
}
```

Output:



```
TERMINAL
Consumed: 61
removing data by consumer  Adding data by producer 62 task  using thread 0 and total threads are  63 task using thread 0 and total threads are  1
1
Consumed: 59
removing data by consumer  64 task using thread 0 and total threads are  1
Produced: 62
Consumed: 62
Adding data by producer 63 task  using thread 0 and total threads are  1
Produced: 63
Adding data by producer 64 task  using thread 0 and total threads are  1
Produced: 64

Time taken to execute in seconds : 0.931000
```

Information:
In the producer-consumer problem, you have two types of threads: producers and consumers. Producers generate data items and add them to a shared buffer, while consumers remove data items from the buffer and process them. The challenge is to ensure that producers don't add items to a full buffer and consumers don't remove items from an empty buffer while avoiding race conditions.

Github Link: