

Date:12.09.2022

Final Year B. Tech., Sem VII 2021-22

High Performance Computing Lab

Assignment submission

PRN No: 2019BTECS00070

Full name: Killedar Prathmesh

Batch: B3

Assignment: 4

Title of assignment: Study and Implementation of schedule, nowait, reduction, ordered and collapse clauses

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 Series using Dynamic Programming #include<stdio.h>
```

```
int fib(int n)
```

```
{
```

```
    /* Declare an array to store Fibonacci numbers. */ int f[n+2]; // 1  
    extra to handle case, n = 0
```

```
    int i;
```

```
    /* 0th and 1st number of the series are 0 and 1 */ f[0] = 0;
```

```

f[1] = 1;

for (i = 2; i <= n; i++)
{
    /* Add the previous 2 numbers in the series and store it */
    f[i] = f[i-1] + f[i-2];
}
return f[n];
}

int main ()
{
    int n = 9; printf("%d", fib(n)); getchar();
    return 0;
}

```

Ans:

The following code is the recursive parallel code for the Fibonacci series. As there are some programs which can only be solved serially because more number of dependencies get involved to make them parallel which increases the overhead. So DP solution of Fibonacci problem is difficult to do in parallel

In the below code, the parallel directive denotes a parallel region which will be executed by four threads. In the parallel construct, the single directive is used to indicate that only one of the threads will execute the print statement that calls fib(n).

The call to fib(n) generates two tasks, indicated by the task directive. One of the tasks computes fib(n-1) and the other computes fib(n-2), and the return values are added together to produce the value returned by fib(n). Each of the calls to fib(n-1) and fib(n-2) will in turn generate two tasks. Tasks will be recursively generated until the argument passed to fib() is less than 2.

The taskwait directive ensures that the two tasks generated in an invocation of fib() are completed (that is. the tasks compute i and j) before that invocation of fib() returns.

Note that although only one thread executes the single directive and hence the call to fib(n), all four threads will participate in executing the tasks generated.

Code:

```
//Fibonacci Series using Recursion
```

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

```
int fib(int n)
```

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

        #pragma omp task shared(j) firstprivate(n)
        j = fib(n-2);

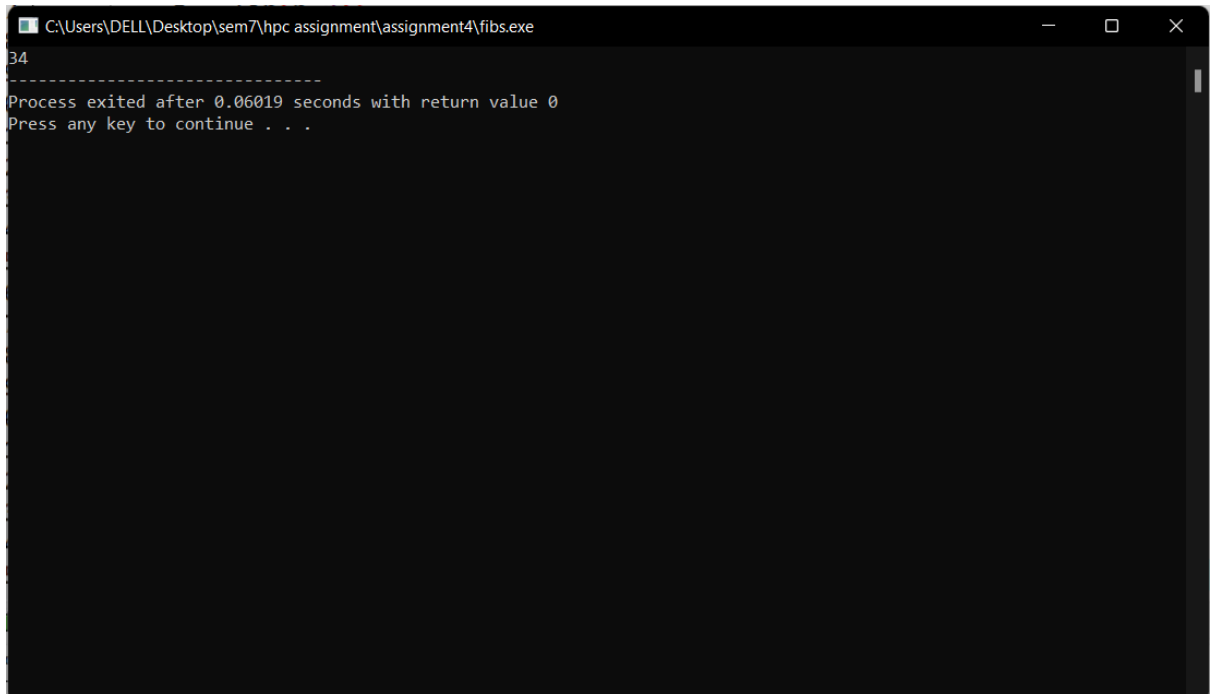
        #pragma omp taskwait
        return i+j;
    }
}
```

```
int main ()
```

```
{
```

```
int n = 9;
#pragma omp parallel shared(n)
{
    #pragma omp single
    printf("%d", fib(n));
}
// getchar();
return 0;
}
```

Output:



A screenshot of a Windows command prompt window. The title bar shows the file path: C:\Users\DELL\Desktop\sem7\hpc assignment\assignment4\fib.exe. The window content shows the output of the program: 34, followed by a separator line of dashes, and then the message "Process exited after 0.06019 seconds with return value 0" and "Press any key to continue . . .".

```
C:\Users\DELL\Desktop\sem7\hpc assignment\assignment4\fib.exe
34
-----
Process exited after 0.06019 seconds with return value 0
Press any key to continue . . .
```

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)

```
// C program for the above approach
```

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

```
#include <stdlib.h>
```

```
// Initialize a mutex to 1
```

```
int mutex = 1;
```

```
// Number of full slots as 0
```

```
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;

    }

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

```

Ans:

In producer consumer problem the producer generates the data and put in the buffer which is consumed by the consumer. This putting and taking of the data by producer and consumer is in parallel.

So, to do this we use the pragma omp file given by cpp predefined making the for loop parallel. But if we do this then there is a synchronisation issue and the data would go wrong. So to avoid this we have to synchronise the code which is common in them (i.e. the critical part)

In the above code we have used “#pragma omp critical” so the critical section works properly. This line ‘critical’ specifies that code is executed by only one thread at a time i.e., only one thread can enter the critical section at a time.