Nithin S
221IT085

IT301 Lab Assignment 3

Q1. Write an OpenMP program to demonstrate default clause (default: shared) in parallel directive. Observe the results and analyze it.

Code

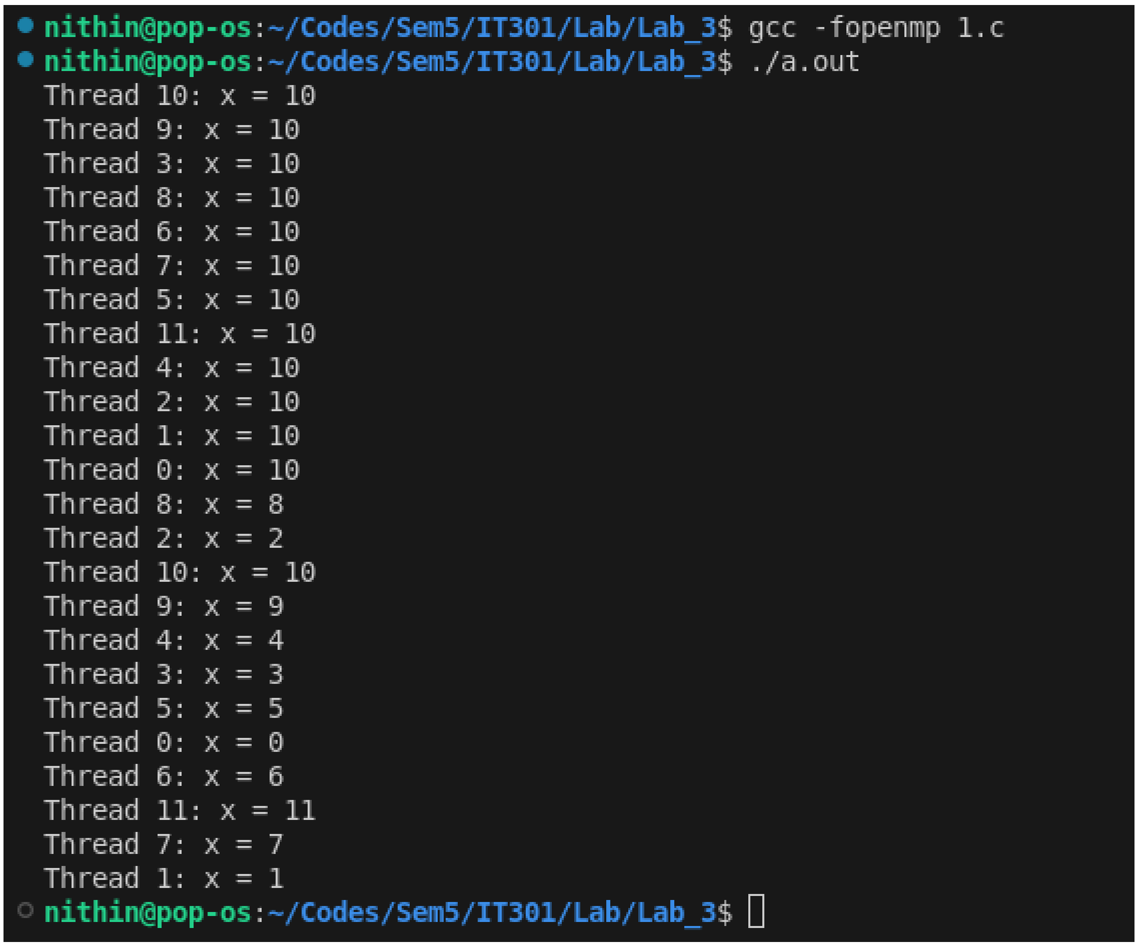
```
#include <omp.h>
#include <stdio.h>

int main() {
    int x = 10;
    #pragma omp parallel default(shared)
    {
        int tid = omp_get_thread_num();
        printf("Thread %d: x = %d\n", tid, x);
    }
    #pragma omp parallel default(none) private(x)
    {
        int tid = omp_get_thread_num();
        x = tid;  // each thread has its own x
        printf("Thread %d: x = %d\n", tid, x);
    }

    return 0;
```

}

Output



```
● nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_3$ gcc -fopenmp 1.c
● nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_3$ ./a.out
  Thread 10: x = 10
  Thread 9: x = 10
  Thread 3: x = 10
  Thread 8: x = 10
  Thread 6: x = 10
  Thread 7: x = 10
  Thread 5: x = 10
  Thread 11: x = 10
  Thread 4: x = 10
  Thread 2: x = 10
  Thread 1: x = 10
  Thread 0: x = 10
  Thread 8: x = 8
  Thread 2: x = 2
  Thread 10: x = 10
  Thread 9: x = 9
  Thread 4: x = 4
  Thread 3: x = 3
  Thread 5: x = 5
  Thread 0: x = 0
  Thread 6: x = 6
  Thread 11: x = 11
  Thread 7: x = 7
  Thread 1: x = 1
○ nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_3$ 
```

Analysis

In OpenMP, the `default` clause is used to specify the default data-sharing attributes of variables in parallel regions. When you create a parallel region using the `#pragma omp parallel` directive, each variable used within the parallel region can either be shared among all threads or private to each thread. The `default` clause allows you to control how OpenMP treats variables by default if you don't explicitly specify their data-sharing attributes.

In the first parallel region (default(shared)), x is shared among all threads, so each thread prints the same value of x.
In the second parallel region (default(none)), x is declared private, so each thread has its own instance of x, and the printed value is different for each thread.

Q2. Demonstrate the usage of lastprivate() clause in a parallel directive in an OpenMP program. Write your observation

Code

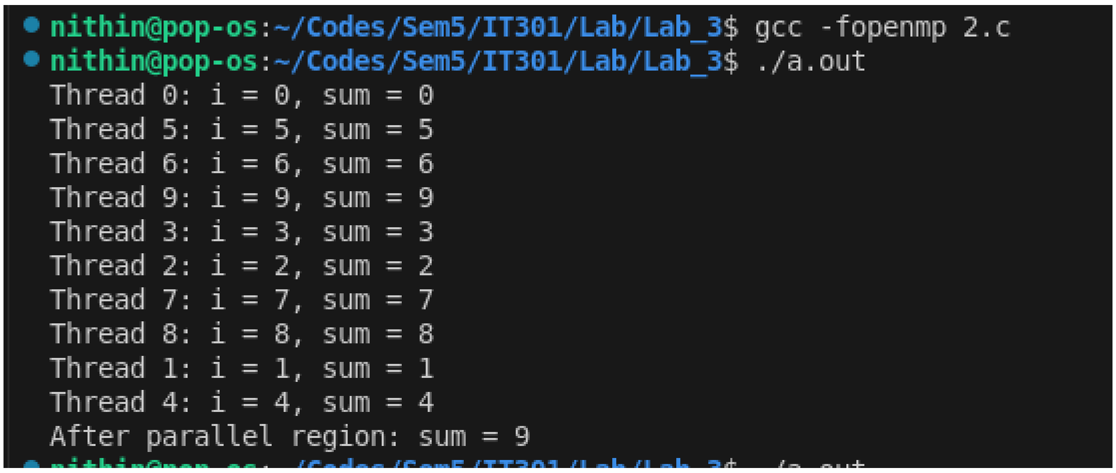
```
#include <omp.h>
#include <stdio.h>

int main() {
    int i;
    int sum = 0;

    #pragma omp parallel for lastprivate(sum)
    for (i = 0; i < 10; i++) {
        sum = i;
        printf("Thread %d: i = %d, sum = %d\n",
omp_get_thread_num(), i, sum);
    }

    printf("After parallel region: sum = %d\n", sum);
    return 0;
}
```

Output



```
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_3$ gcc -fopenmp 2.c
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_3$ ./a.out
Thread 0: i = 0, sum = 0
Thread 5: i = 5, sum = 5
Thread 6: i = 6, sum = 6
Thread 9: i = 9, sum = 9
Thread 3: i = 3, sum = 3
Thread 2: i = 2, sum = 2
Thread 7: i = 7, sum = 7
Thread 8: i = 8, sum = 8
Thread 1: i = 1, sum = 1
Thread 4: i = 4, sum = 4
After parallel region: sum = 9
```

Analysis

The lastprivate clause in OpenMP is used to ensure that the value of a variable after the parallel region corresponds to the value of the variable from the last iteration of the loop executed by any thread. This can be useful in scenarios where you need to preserve the final value of a variable after parallel execution.

Within the Parallel Region: The value of sum varies based on which thread is executing which iteration. The threads execute different iterations in a non-deterministic order.

After the Parallel Region: The value of sum is 9, which is the value of the variable from the last iteration of the loop (i

= 9). This demonstrates that the lastprivate clause successfully updated sum with the value from the last iteration of the loop executed by any thread.

Q3. Write an OpenMP program to demonstrate sharing of section work by performing arithmetic operations on one dimensional array by threads.

Code

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

int main()
{
    int n = 10;
    int arr[] = {1, 2, 3, 4, 5};

#pragma omp parallel
    {
#pragma omp single
        {
#pragma omp critical
            {
                for (int i = 0; i < 5; i++)
                {
                    arr[i] *= 2;
                }
                printf("\nThe first section was executed by thread
no %d", omp_get_thread_num());
```

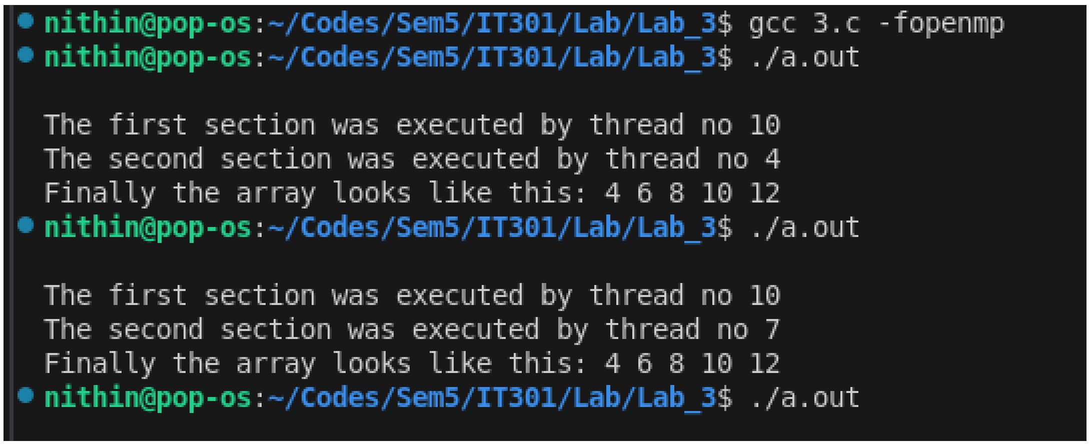
```
            }
        }

#pragma omp sections
        {
#pragma omp section
            {
#pragma omp critical
                {
                    for (int i = 0; i < 5; i++)
                    {
                        arr[i] += 2;
                    }
                    printf("\nThe second section was executed by
thread no %d", omp_get_thread_num());
                }
            }
        }
    }

    printf("\nFinally the array looks like this: ");
    for (int i = 0; i < 5; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");

    return 0;
}
```

Output



```
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_3$ gcc 3.c -fopenmp
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_3$ ./a.out

The first section was executed by thread no 10
The second section was executed by thread no 4
Finally the array looks like this: 4 6 8 10 12
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_3$ ./a.out

The first section was executed by thread no 10
The second section was executed by thread no 7
Finally the array looks like this: 4 6 8 10 12
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_3$ ./a.out
```

Analysis

There are two sections in this code. Each section will be executed by different threads. The first section will be executed first because the single clause is used. Later the second section will be executed by a different thread. You can see which thread is executing which section in the output screenshot. The critical clause is used to avoid race conditions.

Q4. Write an OpenMP program to demonstrate reduction clause in a parallel directive. Write your observation.

Code

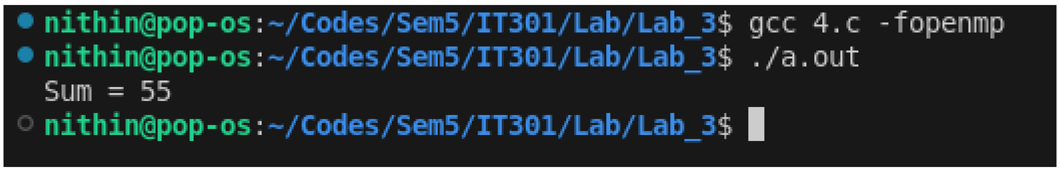
```
#include <omp.h>
#include <stdio.h>

int main() {
    int arr[]={1,2,3,4,5,6,7,8,9,10};
    int sum = 0;

    #pragma omp parallel for reduction(+:sum)
    for (int i = 0; i < 10; i++) {
        sum += arr[i];
    }

    printf("Sum = %d\n", sum);
    return 0;
}
```

Output



```
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_3$ gcc 4.c -fopenmp
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_3$ ./a.out
Sum = 55
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_3$
```

Analysis

The reduction clause in OpenMP is used to handle the situation where multiple threads need to update a shared variable without causing race conditions. It performs a reduction operation on variables in parallel, meaning it combines values from different threads into a single result

The reduction(+:sum) clause tells OpenMP to perform a summation reduction on the sum variable.
Each thread will calculate a partial sum using its own private copy of sum.
After the parallel region, these partial sums are combined into the final value of sum.

Q5. Write a parallel program to find the minimum number in a given array. Use ‘for’ directive for the same along with reduction clause. Write the code, execution results and your observation.

Code

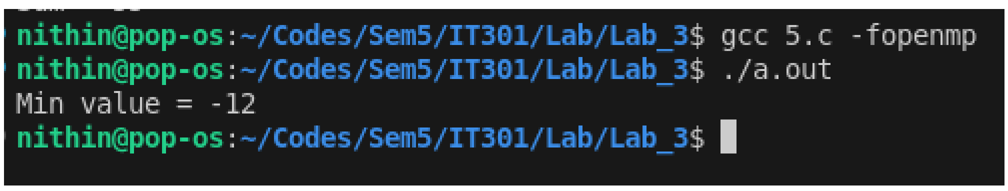
```
#include <omp.h>
#include <stdio.h>

int main() {
    int array[]={100,12,34,56,78,9,3,-12};
    int min_val = 1e9;

    #pragma omp parallel for reduction(min:min_val)
    for (int i = 0; i < 8; i++) {
        if (array[i] < min_val) {
            min_val = array[i];
        }
    }

    printf("Min value = %d\n", min_val);
    return 0;
}
```

Output



```
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_3$ gcc 5.c -fopenmp
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_3$ ./a.out
Min value = -12
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_3$
```

Analysis

The reduction(min:min_val) clause specifies that we want to find the minimum value of min_val across all threads. Each thread updates its own private copy of min_val. After the parallel region, OpenMP combines these private values to compute the final minimum value.