

Ritik Pansuriya(181IT237):

Assignment lab_2:

1. Program 1 : **As X is a shared variable, all the threads will change it's value.**
Aim: To understand and analyze shared clause in parallel directive

```
#include<omp.h>

#include<stdio.h>

int main(){

    int x=0;

    #pragma omp parallel shared(x)

    {

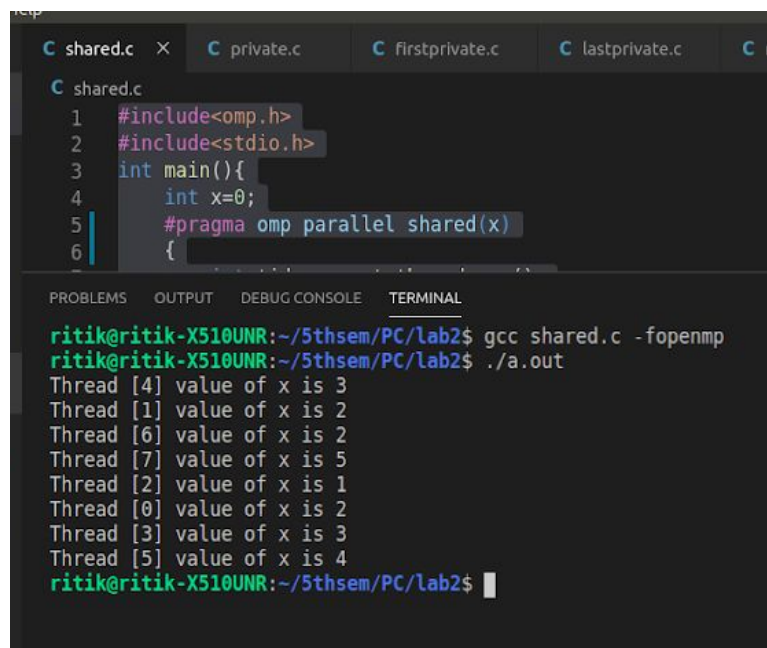
        int tid=omp_get_thread_num();

        x=x+1;

        printf("Thread [%d] value of x is %d \n",tid,x);

    }

}
```



The screenshot shows a code editor with a file named `shared.c` containing the following code:

```
1 #include<omp.h>
2 #include<stdio.h>
3 int main(){
4     int x=0;
5     #pragma omp parallel shared(x)
6     {
```

The terminal output shows the execution of the program using `gcc shared.c -fopenmp` and `./a.out`. The output displays the value of `x` being updated by multiple threads:

```
ritik@ritik-X510UNR:~/5thsem/PC/lab2$ gcc shared.c -fopenmp
ritik@ritik-X510UNR:~/5thsem/PC/lab2$ ./a.out
Thread [4] value of x is 3
Thread [1] value of x is 2
Thread [6] value of x is 2
Thread [7] value of x is 5
Thread [2] value of x is 1
Thread [0] value of x is 2
Thread [3] value of x is 3
Thread [5] value of x is 4
ritik@ritik-X510UNR:~/5thsem/PC/lab2$
```

2. Program 2: Learn the concept of private():

As i variable is private so no thread can see other threads i value and (i=0) in all.

```
#include<stdio.h>

#include<omp.h>

int main()
{
    int i=10;

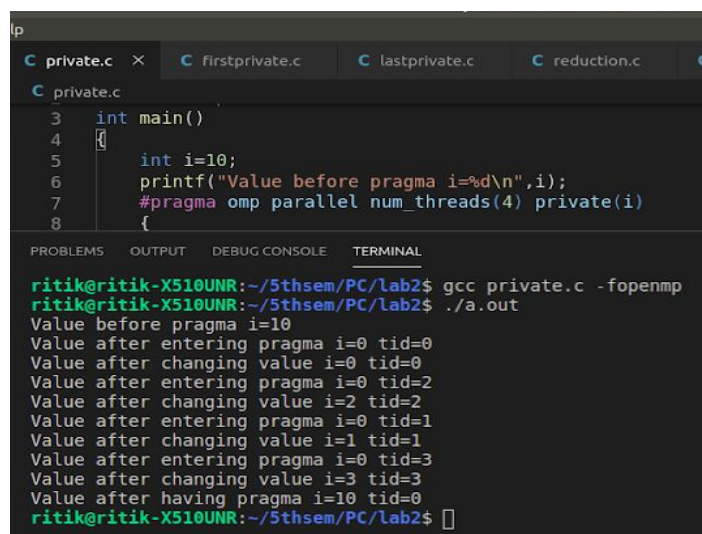
    printf("Value before pragma i=%d\n",i);

    #pragma omp parallel num_threads(4) private(i)
    {
        printf("Value after entering pragma i=%d tid=%d\n",i,
omp_get_thread_num());

        i=i+omp_get_thread_num(); //adds thread_id to i

        printf("Value after changing value i=%d tid=%d\n",i,
omp_get_thread_num());
    }

    printf("Value after having pragma i=%d tid=%d\n",i,
omp_get_thread_num());
}
```



The screenshot shows a code editor with a file named `private.c` open. The code is the same as shown in the previous block. Below the editor, a terminal window shows the output of the program. The terminal prompt is `ritik@ritik-X510UNR:~/5thsem/PC/lab2$`. The command `gcc private.c -fopenmp` is executed, followed by `./a.out`. The output shows the value of `i` before and after the parallel region for each thread. The value of `i` is 10 before the parallel region. Inside the parallel region, each thread increments `i` by its thread ID. The output shows that the value of `i` is 0 for all threads after the parallel region, which is incorrect. This is because the `private(i)` pragma creates a private copy of `i` for each thread, and the main thread's copy of `i` remains 10. The output shows that the value of `i` is 0 for all threads after the parallel region, which is incorrect. This is because the `private(i)` pragma creates a private copy of `i` for each thread, and the main thread's copy of `i` remains 10. The output shows that the value of `i` is 0 for all threads after the parallel region, which is incorrect. This is because the `private(i)` pragma creates a private copy of `i` for each thread, and the main thread's copy of `i` remains 10.

```
ritik@ritik-X510UNR:~/5thsem/PC/lab2$ gcc private.c -fopenmp
ritik@ritik-X510UNR:~/5thsem/PC/lab2$ ./a.out
Value before pragma i=10
Value after entering pragma i=0 tid=0
Value after changing value i=0 tid=0
Value after entering pragma i=0 tid=2
Value after changing value i=2 tid=2
Value after entering pragma i=0 tid=1
Value after changing value i=1 tid=1
Value after entering pragma i=0 tid=3
Value after changing value i=3 tid=3
Value after having pragma i=10 tid=0
ritik@ritik-X510UNR:~/5thsem/PC/lab2$
```

*Note down the result by changing private() to firstprivate() : **Here When we change to Firstprivate i=10 is initialized to all(private) i in each thread thus we can observe Addition in output.**

```
#include<stdio.h>

#include<omp.h>

int main()
{
    int i=10;

    printf("Value before pragma i=%d\n",i);

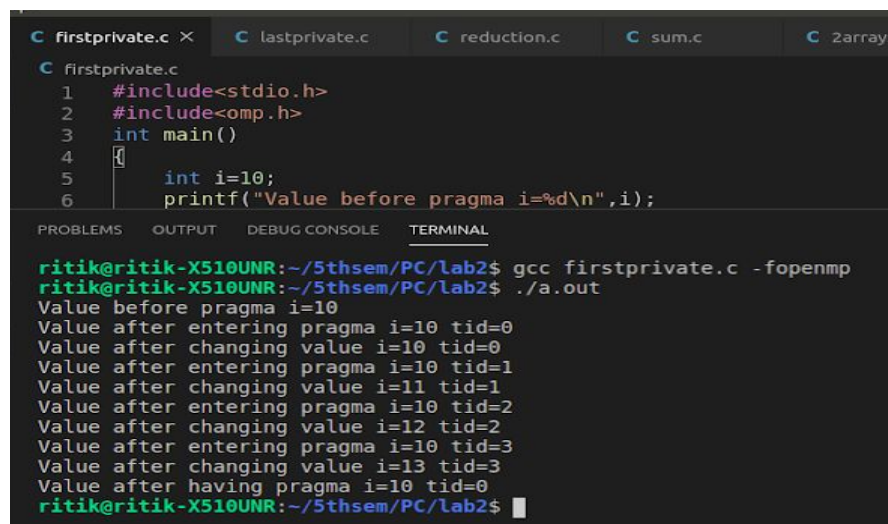
    #pragma omp parallel num_threads(4) firstprivate(i)
    {
        printf("Value after entering pragma i=%d tid=%d\n",i,
omp_get_thread_num());

        i=i+omp_get_thread_num(); //adds thread_id to i

        printf("Value after changing value i=%d tid=%d\n",i,
omp_get_thread_num());

    }

    printf("Value after having pragma i=%d tid=%d\n",i,
omp_get_thread_num());
}
```



The screenshot shows a code editor with several tabs: 'firstprivate.c', 'lastprivate.c', 'reduction.c', 'sum.c', and '2array'. The 'firstprivate.c' tab is active, displaying the following code:

```
1 #include<stdio.h>
2 #include<omp.h>
3 int main()
4 {
5     int i=10;
6     printf("Value before pragma i=%d\n",i);
```

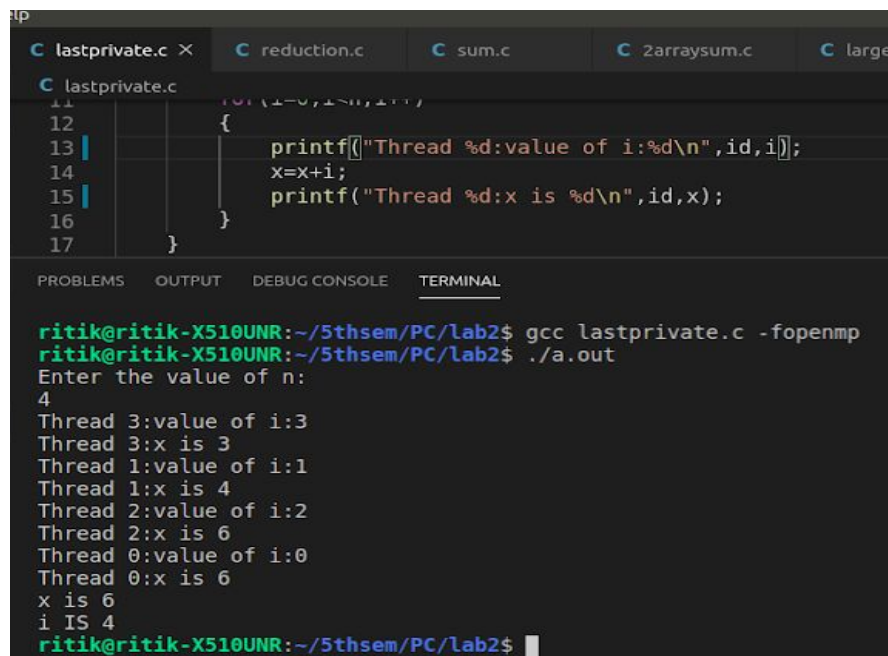
Below the code editor, there is a terminal window with the following output:

```
ritik@ritik-X510UNR:~/5thsem/PC/lab2$ gcc firstprivate.c -fopenmp
ritik@ritik-X510UNR:~/5thsem/PC/lab2$ ./a.out
Value before pragma i=10
Value after entering pragma i=10 tid=0
Value after changing value i=10 tid=0
Value after entering pragma i=10 tid=1
Value after changing value i=11 tid=1
Value after entering pragma i=10 tid=2
Value after changing value i=12 tid=2
Value after entering pragma i=10 tid=3
Value after changing value i=13 tid=3
Value after having pragma i=10 tid=0
ritik@ritik-X510UNR:~/5thsem/PC/lab2$
```

3. Program 3 : Learn the working of lastprivate() clause:

Here we used lastprivate so we can see that value of x is stored as the value of x In last iteration (which is thread 0) and equals 6.

```
#include<stdio.h>
#include<omp.h>
void main(){
    int x=0,i,n;
    printf("Enter the value of n: \n");
    scanf("%d",&n);
    #pragma omp parallel
    {
        int id=omp_get_thread_num();
        #pragma omp for lastprivate(i)
        for(i=0;i<n;i++)
        {
            printf("Thread %d:value of i:%d\n",id,i);
            x=x+i;
            printf("Thread %d:x is %d\n",id,x);
        }
    }
    printf("x is %d\n",x);
    printf("i IS %d\n",i);
}
```



```
C lastprivate.c x C reduction.c C sum.c C 2arraysum.c C large
C lastprivate.c
11 // i=0,1,2,3,4
12 {
13     printf("Thread %d:value of i:%d\n",id,i);
14     x=x+i;
15     printf("Thread %d:x is %d\n",id,x);
16 }
17 }

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
ritik@ritik-X510UNR:~/5thsem/PC/lab2$ gcc lastprivate.c -fopenmp
ritik@ritik-X510UNR:~/5thsem/PC/lab2$ ./a.out
Enter the value of n:
4
Thread 3:value of i:3
Thread 3:x is 3
Thread 1:value of i:1
Thread 1:x is 4
Thread 2:value of i:2
Thread 2:x is 6
Thread 0:value of i:0
Thread 0:x is 6
x is 6
i IS 4
ritik@ritik-X510UNR:~/5thsem/PC/lab2$
```

4. Demonstration of reduction clause in parallel directive.

Here x is used as in reduction clause where each thread will perform execution And then automatically adds the partial value of x for all threads to shared x. Also value of x in each thread is initialized as outer value of x.

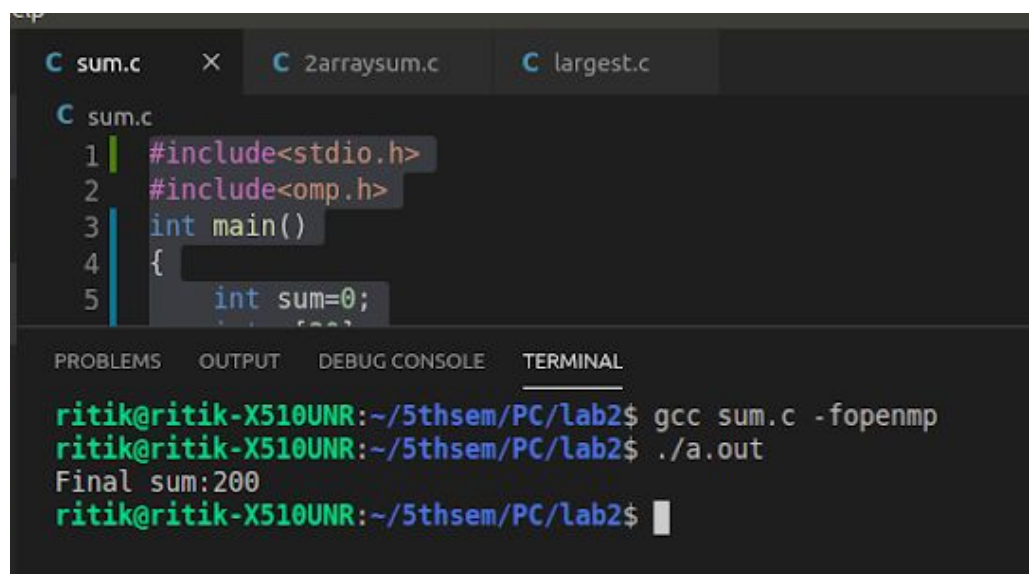
```
#include<stdio.h>
#include<omp.h>
void main()
{
    int x=0;
    #pragma omp parallel num_threads(6) reduction(+:x)
    {
        int id=omp_get_thread_num();
        int threads=omp_get_num_threads();
        x=x+1;
        printf("Hi from %d value of x : %d\n",id,x);
    }
    printf("Final x:%d\n",x);
}
```

The screenshot shows a code editor with a file named `reduction.c` open. The code is identical to the one in the previous block. Below the editor, a terminal window shows the compilation and execution of the program. The terminal output displays the thread IDs and the value of `x` for each of the 6 threads, followed by the final value of `x` after the reduction.

```
ritik@ritik-X510UNR:~/5thsem/PC/lab2$ gcc reduction.c -fopenmp
ritik@ritik-X510UNR:~/5thsem/PC/lab2$ ./a.out
Hi from 0 value of x : 1
Hi from 3 value of x : 1
Hi from 5 value of x : 1
Hi from 1 value of x : 1
Hi from 2 value of x : 1
Hi from 4 value of x : 1
Final x:6
ritik@ritik-X510UNR:~/5thsem/PC/lab2$
```

1. Write a parallel program to calculate the sum of elements in an array
Here i used reduction to calculate the sum because each thread in its iteration Will add corresponding array elements(defined by start and end) and atlast final addition of all partial sums will be done automatically due to the reduction clause. I took size of the array as 20 here which can change accordingly.

```
#include<stdio.h>
#include<omp.h>
int main()
{
    int sum=0;
    int a[20];
    for(int i=0;i<20;i++) a[i]=10;
    #pragma omp parallel reduction(+:sum)
    {
        int id=omp_get_thread_num();
        int threads=omp_get_num_threads();
        int start=20*id/threads;
        int end=20*(id+1)/threads;
        for(int i=start; i<end; i++) sum=sum+a[i];
    }
    printf("Final sum:%d\n",sum);
    return 0;
}
```



The screenshot shows a code editor with three tabs: 'sum.c', '2arraysum.c', and 'largest.c'. The 'sum.c' tab is active, displaying the C code from the previous block. Below the editor is a terminal window with the following output:

```
ritik@ritik-X510UNR:~/5thsem/PC/lab2$ gcc sum.c -fopenmp
ritik@ritik-X510UNR:~/5thsem/PC/lab2$ ./a.out
Final sum:200
ritik@ritik-X510UNR:~/5thsem/PC/lab2$
```

2. Write a parallel program to calculate the $a[i]=b[i]+c[i]$, for all elements in array $b[]$ and $c[]$.

Here I used normal schedule with static and chunk size 4, with number of threads As 8. It will be statically allocated the iterations to each thread in a round robin Pattern. Thus each index is allocated to one of the threads and they will calculate the sum and store it in the array "a".

```
#include<omp.h>

#include<stdio.h>

int main(){

    int i,n=40;

    int a[40],b[40],c[40];

    for(i=0;i<n;i++){

        b[i]=i*4;

        c[i]=i*6;

    }

    #pragma omp parallel for schedule(static,4)

    for(i=0;i<n;i++)

    {

        a[i]=b[i]+c[i];

        printf("Thread id= %d i=%d,a[%d]=%d\n",

omp_get_thread_num(),i,i,a[i]);

    }

}
```

3. Write a parallel program to find the largest among all elements in an array. Here I have a `cur_max` variable which store max element till now and will check If the current value is greater than max then we will update `cur_max` variable. Here critical construct restricts execution of the associated structured block to a single thread at a time. Because for last element to check we don't want That all the thread do that if we don't include the critical construct then it will Miss the last element if greatest.

```
#include <stdio.h>
#include <omp.h>
int main() {
    int array[]={12,15,8,3,24,56,65,1,67,44},i,cur_max;
    cur_max=0;
    #pragma omp parallel for
    for(i=0; i<10; i++){
        if(array[i]>cur_max)
            #pragma omp critical
            if(array[i]>cur_max)
                cur_max=array[i];
    }
    printf("The largest number in given array is %d\n",cur_max);
    return 0;
}
```


Help

C largest.c ×

C largest.c

```
1  #include <stdio.h>
2  #include <omp.h>
3  int main(){
4      int array[]={12,15,8,3,24,56,65,1,67,44},i,cur_max;
5      cur_max=0;
6      #pragma omp parallel for
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL

```
ritik@ritik-X510UNR:~/5thsem/PC/lab2$ gcc largest.c -fopenmp
```

```
ritik@ritik-X510UNR:~/5thsem/PC/lab2$ ./a.out
```

```
The largest number in given array is 67
```

```
ritik@ritik-X510UNR:~/5thsem/PC/lab2$
```
