IT301 Assignment 2

NAME: SUYASH CHINTAWAR
ROLL NO.: 191IT109
TOPIC: LAB-2

1. Program 1 [2 Marks]
Aim: To understand and analyze shared clause in parallel directive.
Execute the program and write your observation. Change number of threads and write your observation. (filename as given in question: 'shared.c')

SOLUTION:

```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 2$ gcc -fopenmp shared.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 2$ ./a.out
Thread [0] value of x is 22
Thread [3] value of x is 23
Thread [1] value of x is 21
Thread [2] value of x is 22
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 2$ ./a.out
Thread [0] value of x is 22
Thread [1] value of x is 21
Thread [3] value of x is 23
Thread [2] value of x is 21
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 2$ ./a.out
Thread [0] value of x is 22
Thread [3] value of x is 24
Thread [1] value of x is 21
Thread [2] value of x is 23
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 2$
```

Fig 1. Number of threads= 4

```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 2$ gcc -fopenmp shared.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 2$ ./a.out
Thread [7] value of x is 24
Thread [0] value of x is 23
Thread [1] value of x is 21
Thread [3] value of x is 21
Thread [4] value of x is 21
Thread [2] value of x is 22
Thread [5] value of x is 22
Thread [6] value of x is 25
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 2$ ./a.out
Thread [0] value of x is 21
Thread [4] value of x is 21
Thread [3] value of x is 21
Thread [6] value of x is 22
Thread [5] value of x is 21
Thread [2] value of x is 21
Thread [7] value of x is 21
Thread [1] value of x is 21
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 2$
```

Fig 2: Number of threads = 8

Observation: The threads can execute in any order. We see that as the variable 'x' is shared amongst the threads, so the changes made by a thread to 'x' get reflected for all the other threads as well. In fig 1. we see that we get different results in every execution. The value of 'x' after 2 threads execute the increment operation may or may not be same. The value of 'x' remains the same if two threads increment the previous value of 'x' at the same time.

2. Program 2 (filename as given in question: 'learn.c') [2 Marks]
Learn the concept of private(), firstprivate()
(a) First execute the program with declaring i as *private(i)*. Along with results , write your observation
(b) Then execute the same program with *firstprivate(i)*. Observe the results and write your observation.

SOLUTION:

(a) private(i)

```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 2$ gcc -fopenmp learn.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 2$ ./a.out
Value of i before pragma i=20
Value after entering pragma i=0 tid=0
Value after changing value i=0 tid=0
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 entering pragma i=0 tid=2
Value after changing value i=2 tid=2
Value after having pragma i=20 tid=0
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 2$ ./a.out
Value of i before pragma i=20
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=20 tid=0
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 2$
```

Fig 3. Declaring 'i' as private

Observation: The threads can execute in any order. Even though the value of variable 'i' has been initialized to 20 before the pragma loop, but as 'i' is declared private, the value of 'i' for each thread at the start gets initialized to zero. After the pragma loop, the value of 'i' is again only for the master thread and hence its value is 20 (tid=0 after pragma loop guarantees that its the master thread).

(b) firstprivate(i)

(continued on next page...)

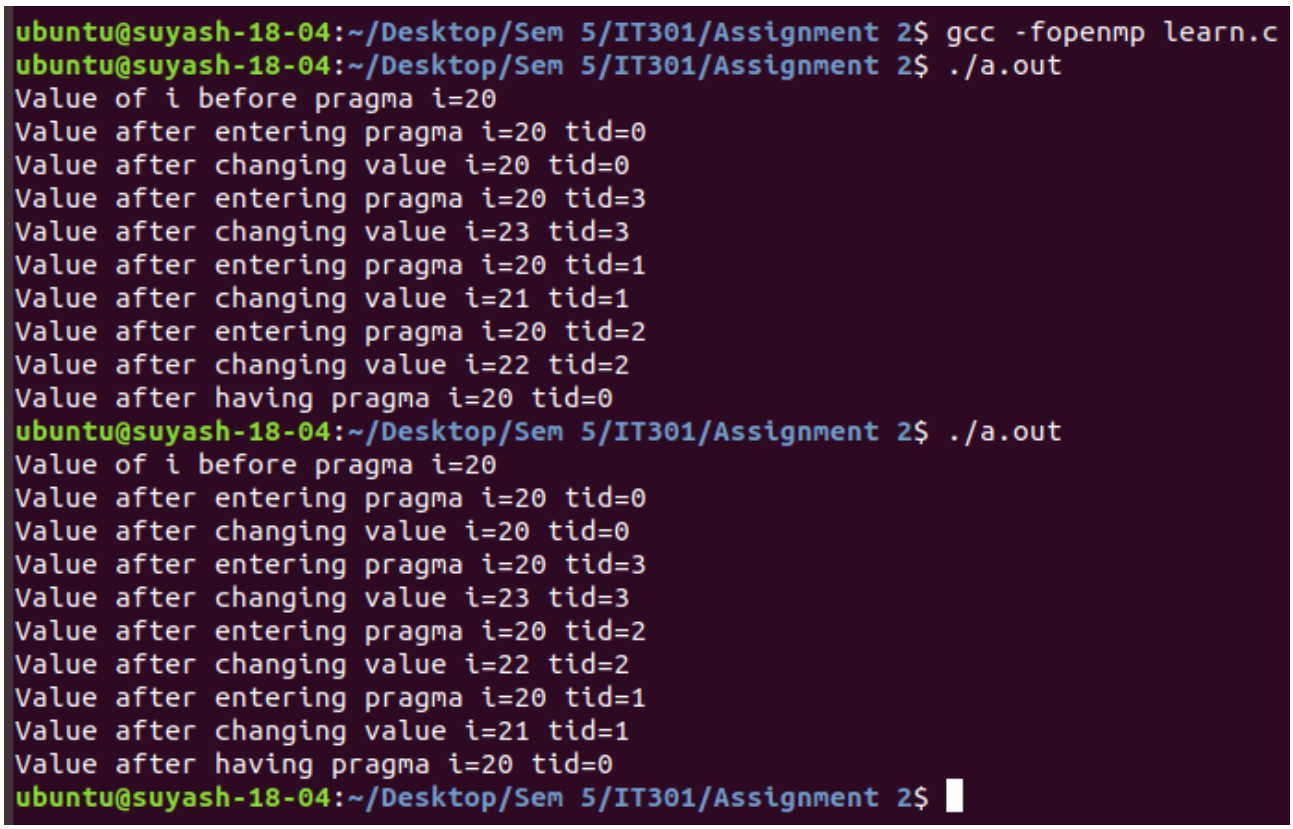

Fig 4. Declaring ‘i’ as firstprivate

Observation: Using firstprivate initializes ‘i’ with its value which was just before the pragma loop occured. In this case it was 20. So for each thread ‘i’ get initialized to 20. After coming out of the pragma loop the value of ‘i’ gets restored to 20 as the ‘i’ in the pragma was private to each of the threads which doesn’t affect the original value of the variable outside of the pragma loop.

3. Programming exercise [6 Marks]
Write a parallel program to perform c[i]=a[i]+b[i] where i=0,1,2……N. Execute the program by varying number of elements and number of threads. Check the computation done by each thread.
Write code, execution results and your observation.

SOLUTION:

(continued on next page...)

Code:

```c
/* Performs c[i]=a[i]+b[i] */
#include<stdio.h>
#include<omp.h>
int main()
{
    int threads,n;

    //Take input from user
    printf("Enter size of arrays a,b,c: ");
    scanf("%d",&n);
    printf("Enter number of threads: ");
    scanf("%d",&threads);
    omp_set_num_threads(threads); //set number of threads
    int a[n],b[n],c[n];
    printf("Enter elements of array a (%d numbers): ",n);
    for(int i=0;i<n;i++)
    {
        scanf("%d",&a[i]);
    }
    printf("Enter elements of array b (%d numbers): ",n);
    for(int i=0;i<n;i++)
    {
        scanf("%d",&b[i]);
    }

    int tid,low,high,i;
    #pragma omp parallel default(shared) private(tid,low,high,i)
    {
        tid = omp_get_thread_num();
        low = n*tid/threads;
        high = n*(tid+1)/threads;

        //check computation of each thread
        printf("tid:%d, low:%d, high-1:%d",tid,low,high-1);

        for(i=low; i<high; i++) c[i]=b[i]+a[i];
    }

    printf("Final array after addition:\n");
    for(i=0;i<n;i++) printf("%d ",c[i]);
    printf("\n");
}
```

Fig 5. Code

Algorithm:

1. Take input from the user. (number of threads, array size, array elements)
2. Declare pragma loop with arrays a,b,c and size of array as the shared elements and tid (i.e. Thread ID), low (starting array index for a thread), high(ending array index for a thread) as the private elements to the thread.
3. Calculate low, high values for each thread and perform addition from array indices low to high for respective threads. Check the computation of each thread by printing the low and high-1 of each thread which tells us the amount of work done by each thread.
4. Display the resulting array.

Results:

(a) Varying number of threads and keeping array size constant

```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 2$ gcc -fopenmp addition.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 2$ ./a.out
Enter size of arrays a,b,c: 4
Enter number of threads: 2
Enter elements of array a (4 numbers): 1 2 3 4
Enter elements of array b (4 numbers): 5 6 7 8
tid:0, low:0, high-1:1
tid:1, low:2, high-1:3
Final array after addition:
6 8 10 12
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 2$ ./a.out
Enter size of arrays a,b,c: 4
Enter number of threads: 3
Enter elements of array a (4 numbers): 1 2 3 4
Enter elements of array b (4 numbers): 5 6 7 8
tid:0, low:0, high-1:0
tid:1, low:1, high-1:1
tid:2, low:2, high-1:3
Final array after addition:
6 8 10 12
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 2$ ./a.out
Enter size of arrays a,b,c: 4
Enter number of threads: 8
Enter elements of array a (4 numbers): 1 2 3 4
Enter elements of array b (4 numbers): 5 6 7 8
tid:0, low:0, high-1:-1
tid:2, low:1, high-1:0
tid:7, low:3, high-1:3
tid:1, low:0, high-1:0
tid:5, low:2, high-1:2
tid:4, low:2, high-1:1
tid:3, low:1, high-1:1
tid:6, low:3, high-1:2
Final array after addition:
6 8 10 12
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 2$
```

Fig 6. num_threads -> different , size -> constant

Observation: We see that when the array size is divisble by the number of threads, least amount of overlapping is obtained and hence every thread performs a unique operation. We can also see in Fig 6. that when number of threads exceeds the size of the array, some threads do not perform any work. We see that when size=4 and num_threads=8, Threads 0,2,4,6 do not perform any work which is equivalent to using 4 threads (These threads do not perform any work as high<low). We can infer that we can use a maximum of ‘n’ threads for an ‘n’ sized array.

(continued on next page...)

(b) Varying array size and keeping number of threads constant

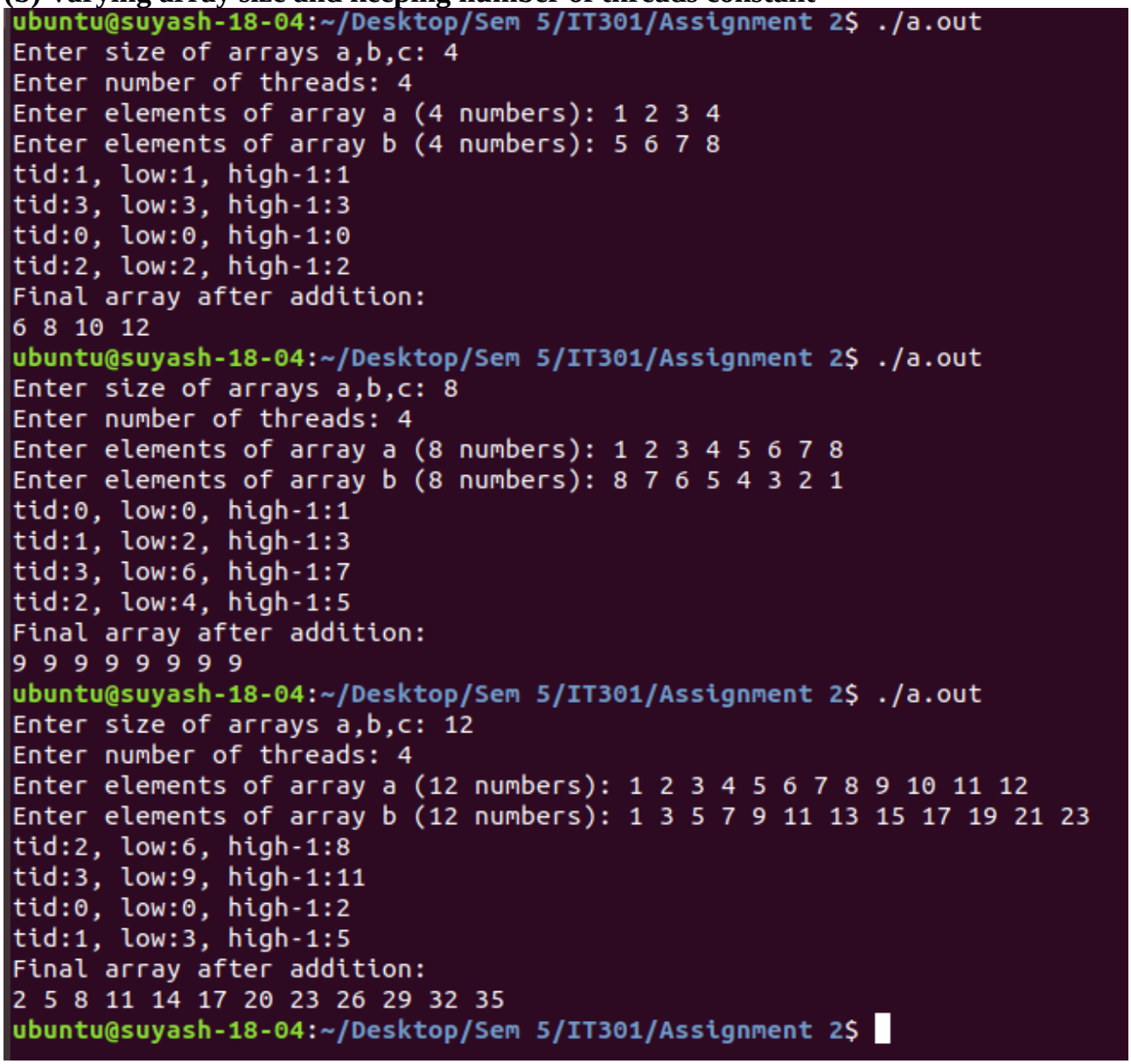

Fig 7. num_threads -> constant , size -> different

Observation: Here also we can see that as the array size increases, the division of work amongst the threads is better and vice versa.

THANK YOU