IT301 Assignment 4

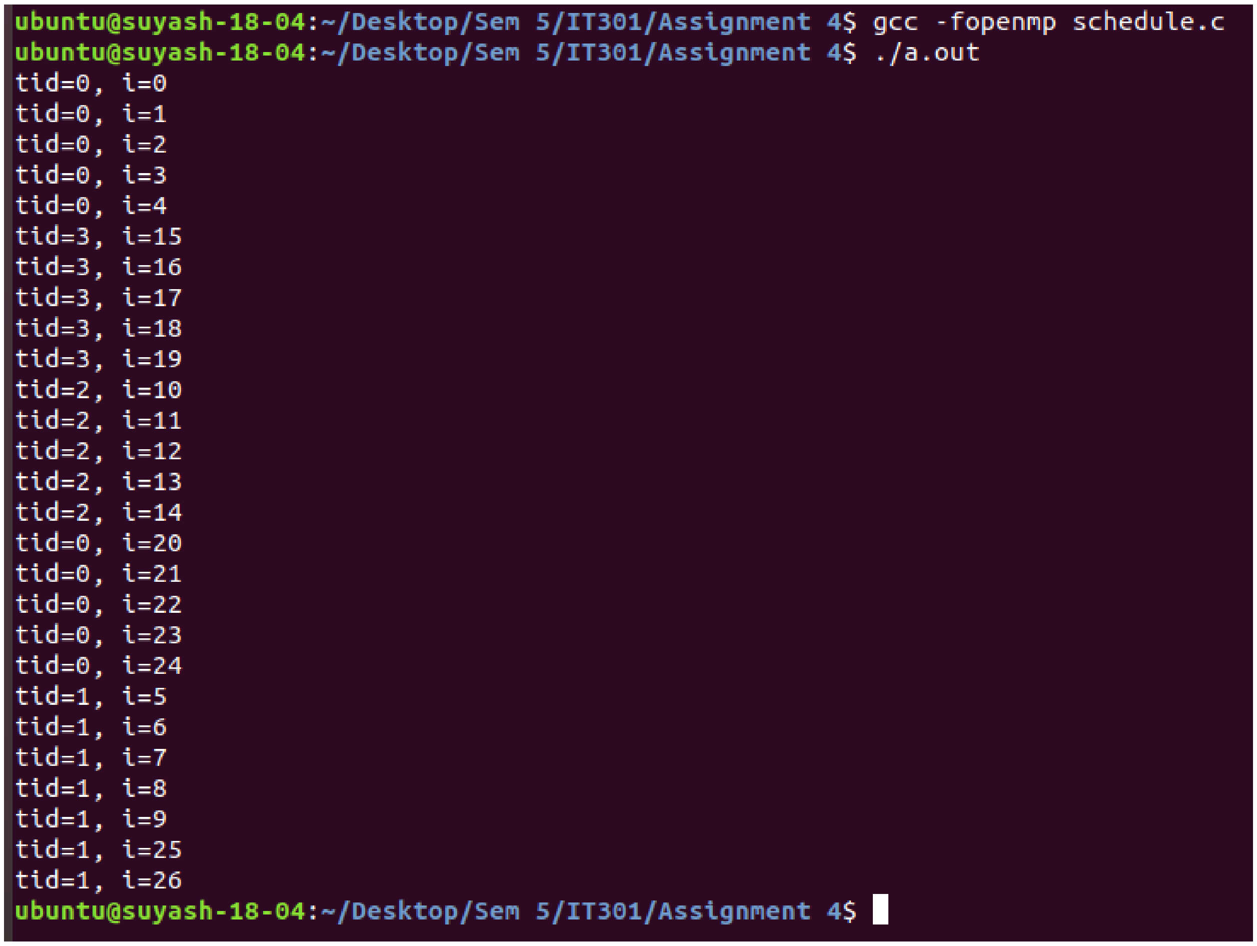
NAME: SUYASH CHINTAWAR
ROLL NO.: 191IT109
TOPIC: LAB 4

Note: The codes for problems have not been attached as they were provided in the problem statement itself.

Q1. Understanding the concept of schedule. Write the observation using schedule (static, 5), schedule (dynamic, 5) and schedule (guided, 5)
[Marks: 1+1+1=3]

SOLUTION:

(a) schedule(static,5)



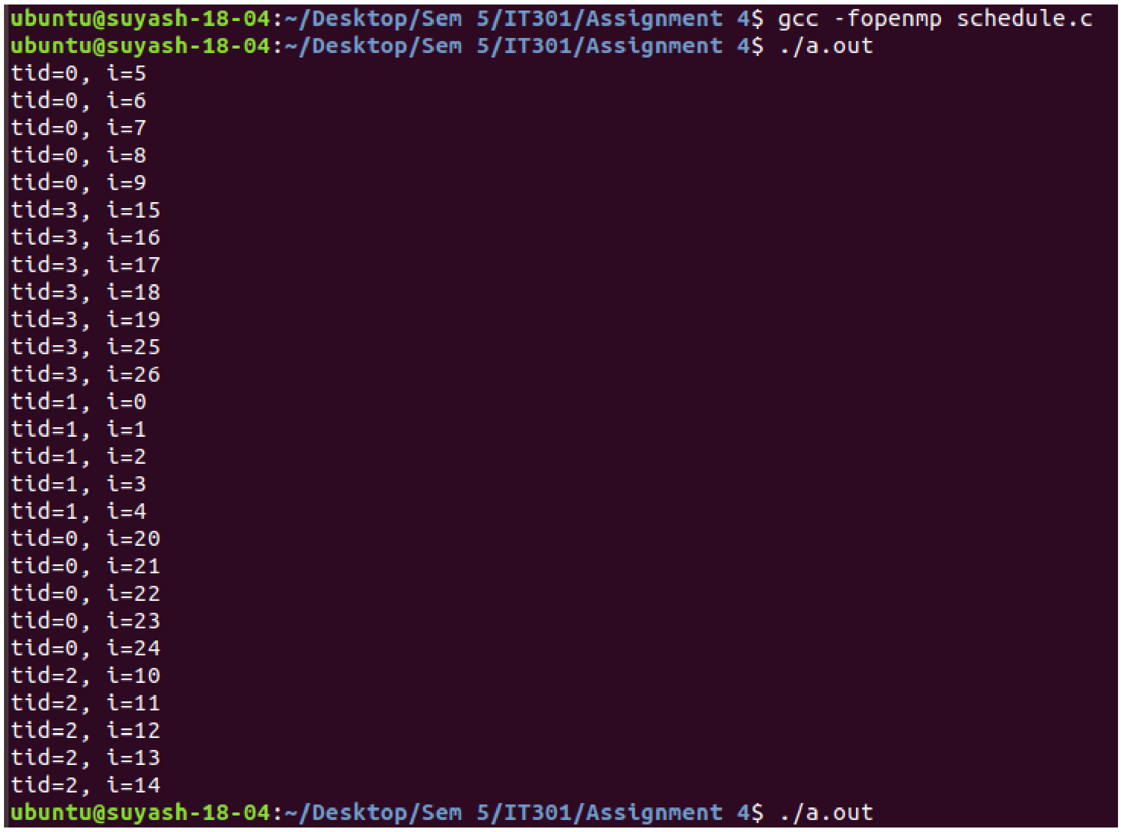
```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ gcc -fopenmp schedule.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ ./a.out
tid=0, i=0
tid=0, i=1
tid=0, i=2
tid=0, i=3
tid=0, i=4
tid=3, i=15
tid=3, i=16
tid=3, i=17
tid=3, i=18
tid=3, i=19
tid=2, i=10
tid=2, i=11
tid=2, i=12
tid=2, i=13
tid=2, i=14
tid=0, i=20
tid=0, i=21
tid=0, i=22
tid=0, i=23
tid=0, i=24
tid=1, i=5
tid=1, i=6
tid=1, i=7
tid=1, i=8
tid=1, i=9
tid=1, i=25
tid=1, i=26
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$
```

Fig 1. schedule(static,5)

Observations: As the scheduling is static in nature, and the chunk size is 5, every thread, in the order of their thread ids will get 5 iterations each in round robin fashion (except for the last chunk - 2 iterations remaining). The iterations assigned are as follows,

Thread ID	0	1	2	3
Iterations	0-4, 20-24	5-9, 25-26	10-14	15-19

(b) schedule(dynamic,5)



```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ gcc -fopenmp schedule.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ ./a.out
tid=0, i=5
tid=0, i=6
tid=0, i=7
tid=0, i=8
tid=0, i=9
tid=3, i=15
tid=3, i=16
tid=3, i=17
tid=3, i=18
tid=3, i=19
tid=3, i=25
tid=3, i=26
tid=1, i=0
tid=1, i=1
tid=1, i=2
tid=1, i=3
tid=1, i=4
tid=0, i=20
tid=0, i=21
tid=0, i=22
tid=0, i=23
tid=0, i=24
tid=2, i=10
tid=2, i=11
tid=2, i=12
tid=2, i=13
tid=2, i=14
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ ./a.out
```

Fig 2. schedule(dynamic,5) execution 1

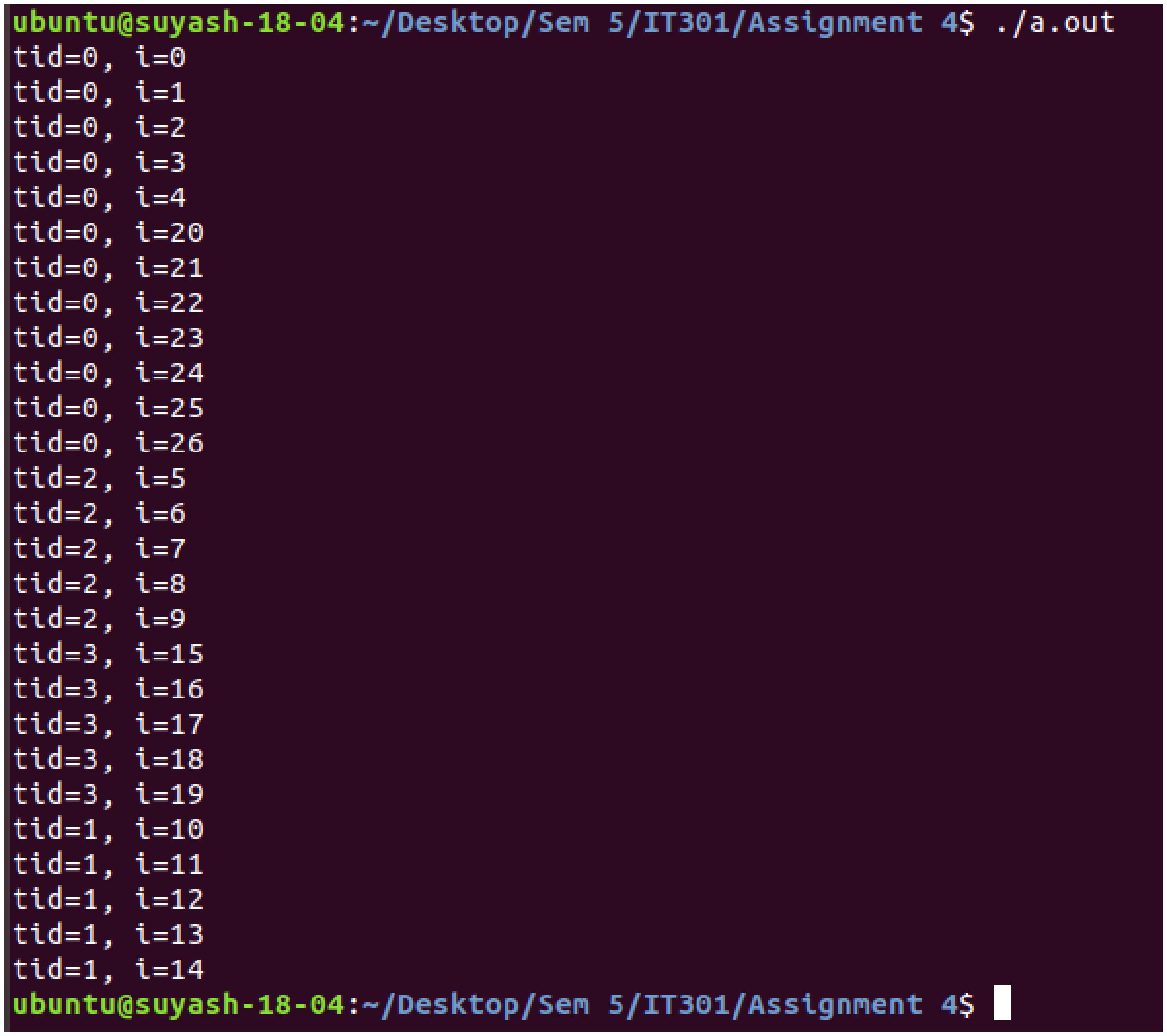
Observations: In dynamic scheduling, iterations are assigned at runtime in the order in which the thread requests. Also, as the chunk size is 5, every thread got 5 iterations each (except for the last chunk). After executing twice, (fig 2, and fig 3.) different results were obtained. In case of first execution,

Thread ID	0	1	2	3
Iterations	5-9, 20-24	0-4	10-14	15-19,25-26

In case of second execution,

Thread ID	0	1	2	3
Iterations	0-4,20-24,25-26	10-14	5-9	15-19

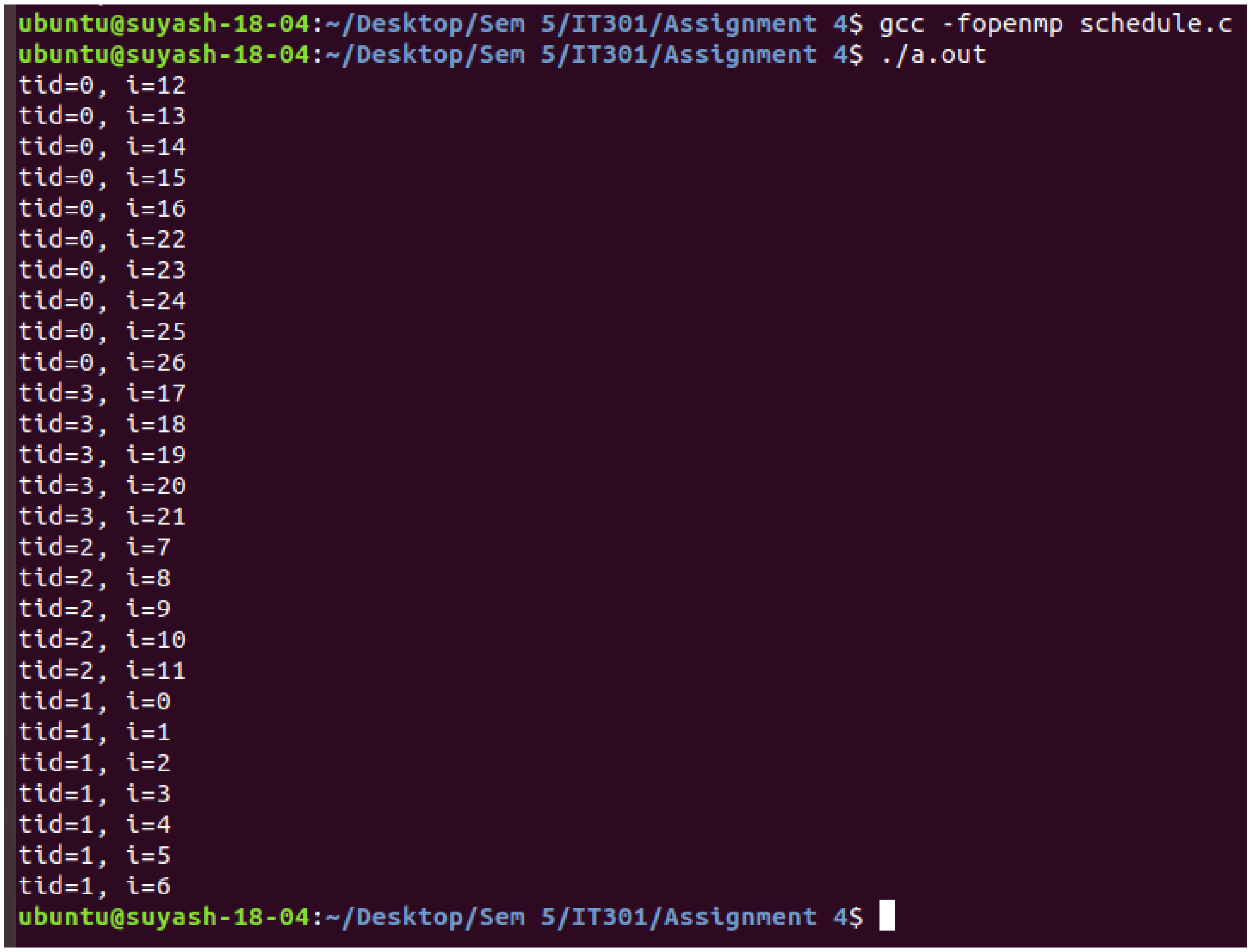
Hence, we see that the iterations assigned are totally dependent upon the requests made by the threads.



```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ ./a.out
tid=0, i=0
tid=0, i=1
tid=0, i=2
tid=0, i=3
tid=0, i=4
tid=0, i=20
tid=0, i=21
tid=0, i=22
tid=0, i=23
tid=0, i=24
tid=0, i=25
tid=0, i=26
tid=2, i=5
tid=2, i=6
tid=2, i=7
tid=2, i=8
tid=2, i=9
tid=3, i=15
tid=3, i=16
tid=3, i=17
tid=3, i=18
tid=3, i=19
tid=1, i=10
tid=1, i=11
tid=1, i=12
tid=1, i=13
tid=1, i=14
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$
```

Fig 3. schedule(dynamic,5) execution 2

(c) schedule(guided,5)



```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ gcc -fopenmp schedule.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ ./a.out
tid=0, i=12
tid=0, i=13
tid=0, i=14
tid=0, i=15
tid=0, i=16
tid=0, i=22
tid=0, i=23
tid=0, i=24
tid=0, i=25
tid=0, i=26
tid=3, i=17
tid=3, i=18
tid=3, i=19
tid=3, i=20
tid=3, i=21
tid=2, i=7
tid=2, i=8
tid=2, i=9
tid=2, i=10
tid=2, i=11
tid=1, i=0
tid=1, i=1
tid=1, i=2
tid=1, i=3
tid=1, i=4
tid=1, i=5
tid=1, i=6
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ 
```

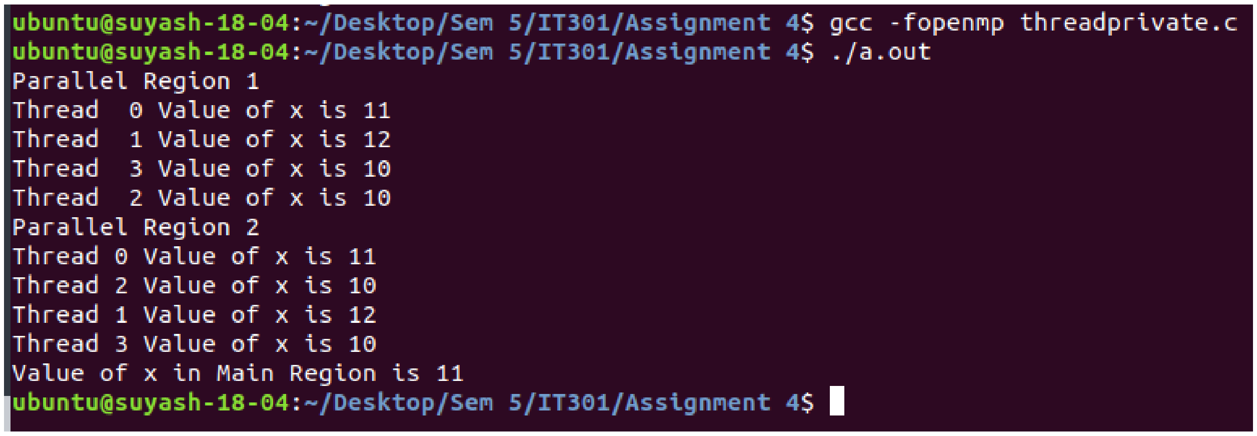
Fig 4. schedule(guided,5)

Observations: Guided scheduling divides and assigns iterations by calculating number of iterations remaining divided by the number of threads in every step. It is similar to dynamic scheduling in terms of runtime scheduling and the second parameter is the minimum number of iterations to be assigned (i.e. 5 here). If the number of remaining iterations becomes less than 5, all remaining iterations are assigned to the requesting thread. The iterations assigned in fig 4 are as follows,

Thread ID	0	1	2	3
Iterations	12-16,22-26	0-6	7-11	17-21

Q2. Execute following code and observe the working of threadprivate directive and copyin clause:

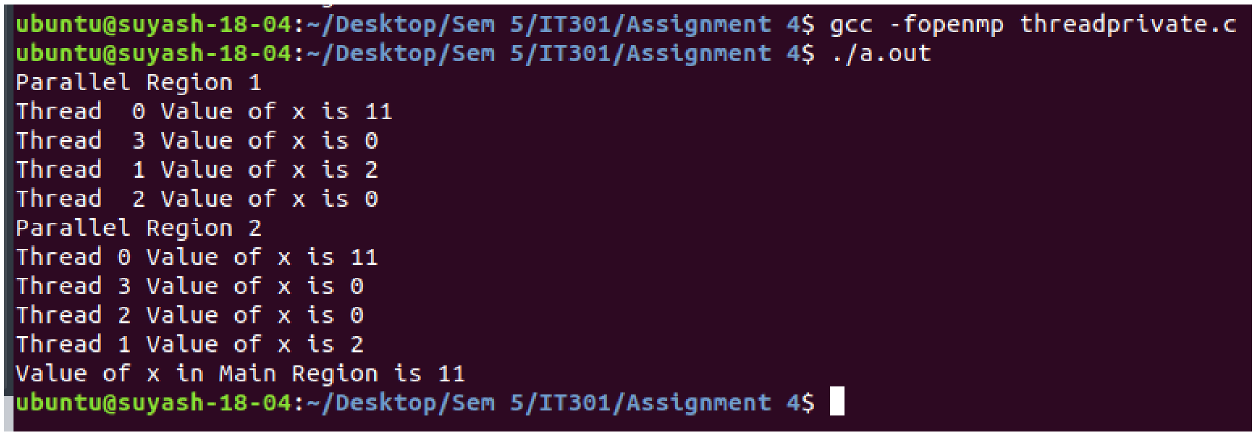
SOLUTION:



```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ gcc -fopenmp threadprivate.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ ./a.out
Parallel Region 1
Thread  0 Value of x is 11
Thread  1 Value of x is 12
Thread  3 Value of x is 10
Thread  2 Value of x is 10
Parallel Region 2
Thread 0 Value of x is 11
Thread 2 Value of x is 10
Thread 1 Value of x is 12
Thread 3 Value of x is 10
Value of x in Main Region is 11
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$
```

Fig 5. Original code execution

a. Remove copyin clause and check the output.

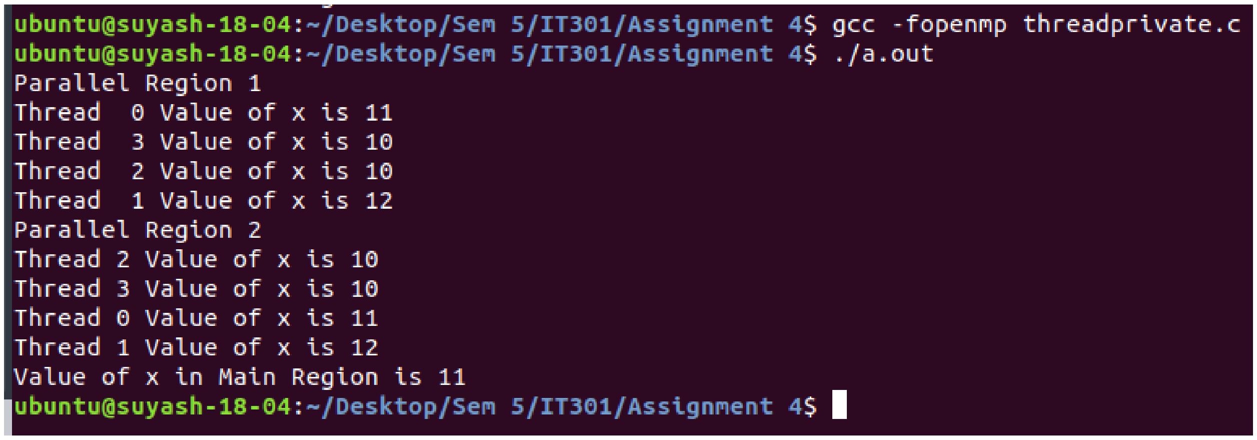


```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ gcc -fopenmp threadprivate.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ ./a.out
Parallel Region 1
Thread  0 Value of x is 11
Thread  3 Value of x is 0
Thread  1 Value of x is 2
Thread  2 Value of x is 0
Parallel Region 2
Thread 0 Value of x is 11
Thread 3 Value of x is 0
Thread 2 Value of x is 0
Thread 1 Value of x is 2
Value of x in Main Region is 11
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$
```

Fig 6. Removing copyin clause

Observations: The scope of threadprivate variables are particular to every thread even across parallel regions unlike the normal private variables. The copyin clause for threadprivate variables acts similar to the firstprivate clause for private variables. The copyin clause will copy the value of a threadprivate variable of the master thread to the threadprivate variable of each other member of the team that is executing the parallel region. In this case, when the original code is executed the value of x=10 in master is copied to all other threads. When the copyin clause is removed, only the master thread has the value of x=10 and other threads' x values get initialized with zero (as in private clause). After execution of both parallel regions, only the master threads continues the sequential execution so the value of x in the master thread is retained in the main program.

a. Remove copyin clause and initialize x globally.



```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ gcc -fopenmp threadprivate.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ ./a.out
Parallel Region 1
Thread  0 Value of x is 11
Thread  3 Value of x is 10
Thread  2 Value of x is 10
Thread  1 Value of x is 12
Parallel Region 2
Thread 2 Value of x is 10
Thread 3 Value of x is 10
Thread 0 Value of x is 11
Thread 1 Value of x is 12
Value of x in Main Region is 11
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$
```

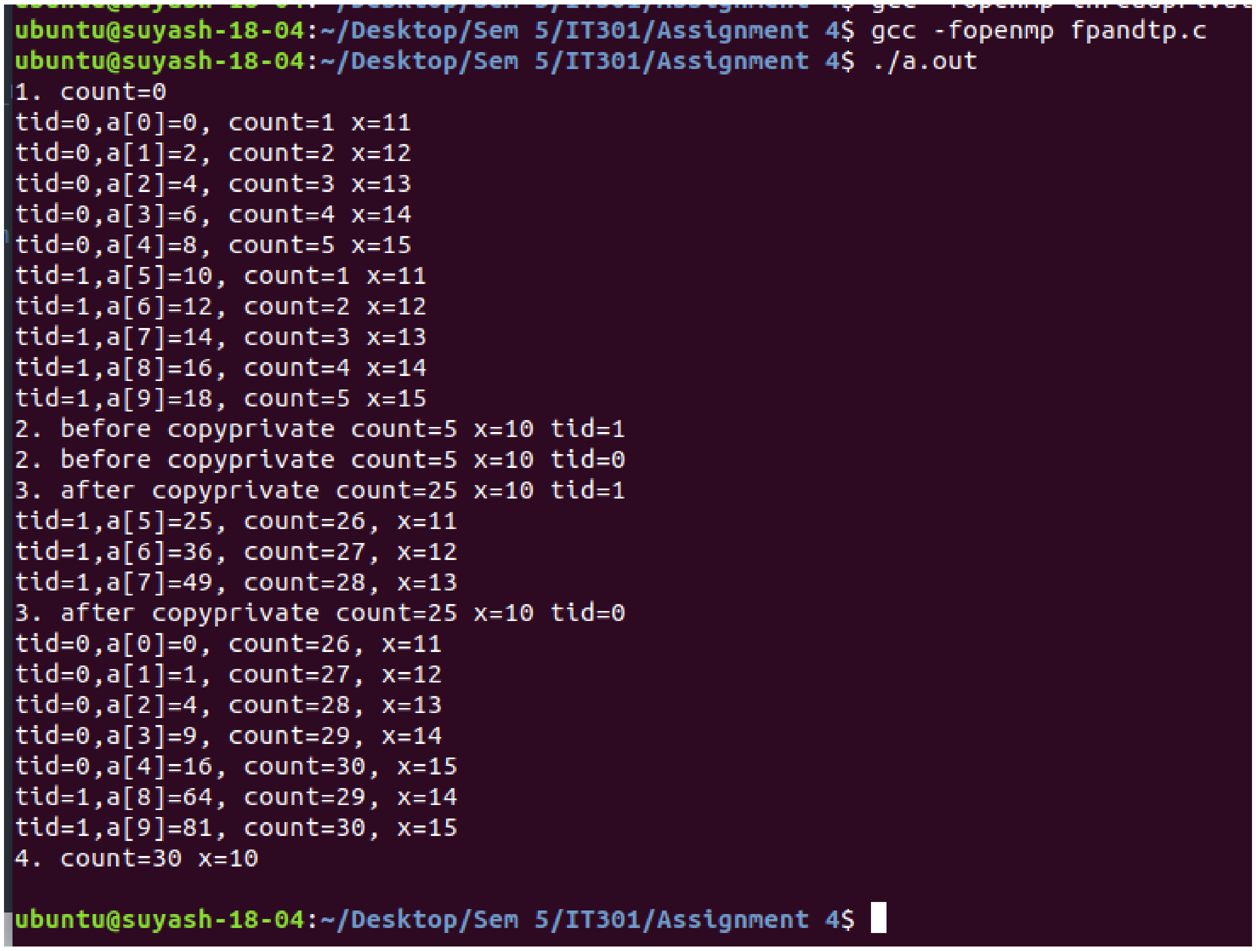
Fig 7. Removing copyin and initialization of x globally

Observations: We can see that the output in fig. 7 is exactly the same as that when the original code is executed in fig. 5. This happens because the global initialization of x is reflected for all threads. Hence, the value x=10 is initialized for all threads even though copyin is not used.

Q3. Learn the concept of firstprivate() and threadprivate(). Analyse the results for variable count and x. write your observation.

SOLUTION:

(continued on next page...)



```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ gcc -fopenmp fpandtp.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ ./a.out
1. count=0
tid=0,a[0]=0, count=1 x=11
tid=0,a[1]=2, count=2 x=12
tid=0,a[2]=4, count=3 x=13
tid=0,a[3]=6, count=4 x=14
tid=0,a[4]=8, count=5 x=15
tid=1,a[5]=10, count=1 x=11
tid=1,a[6]=12, count=2 x=12
tid=1,a[7]=14, count=3 x=13
tid=1,a[8]=16, count=4 x=14
tid=1,a[9]=18, count=5 x=15
2. before copyprivate count=5 x=10 tid=1
2. before copyprivate count=5 x=10 tid=0
3. after copyprivate count=25 x=10 tid=1
tid=1,a[5]=25, count=26, x=11
tid=1,a[6]=36, count=27, x=12
tid=1,a[7]=49, count=28, x=13
3. after copyprivate count=25 x=10 tid=0
tid=0,a[0]=0, count=26, x=11
tid=0,a[1]=1, count=27, x=12
tid=0,a[2]=4, count=28, x=13
tid=0,a[3]=9, count=29, x=14
tid=0,a[4]=16, count=30, x=15
tid=1,a[8]=64, count=29, x=14
tid=1,a[9]=81, count=30, x=15
4. count=30 x=10

ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$
```

Fig 8. Output of Q3

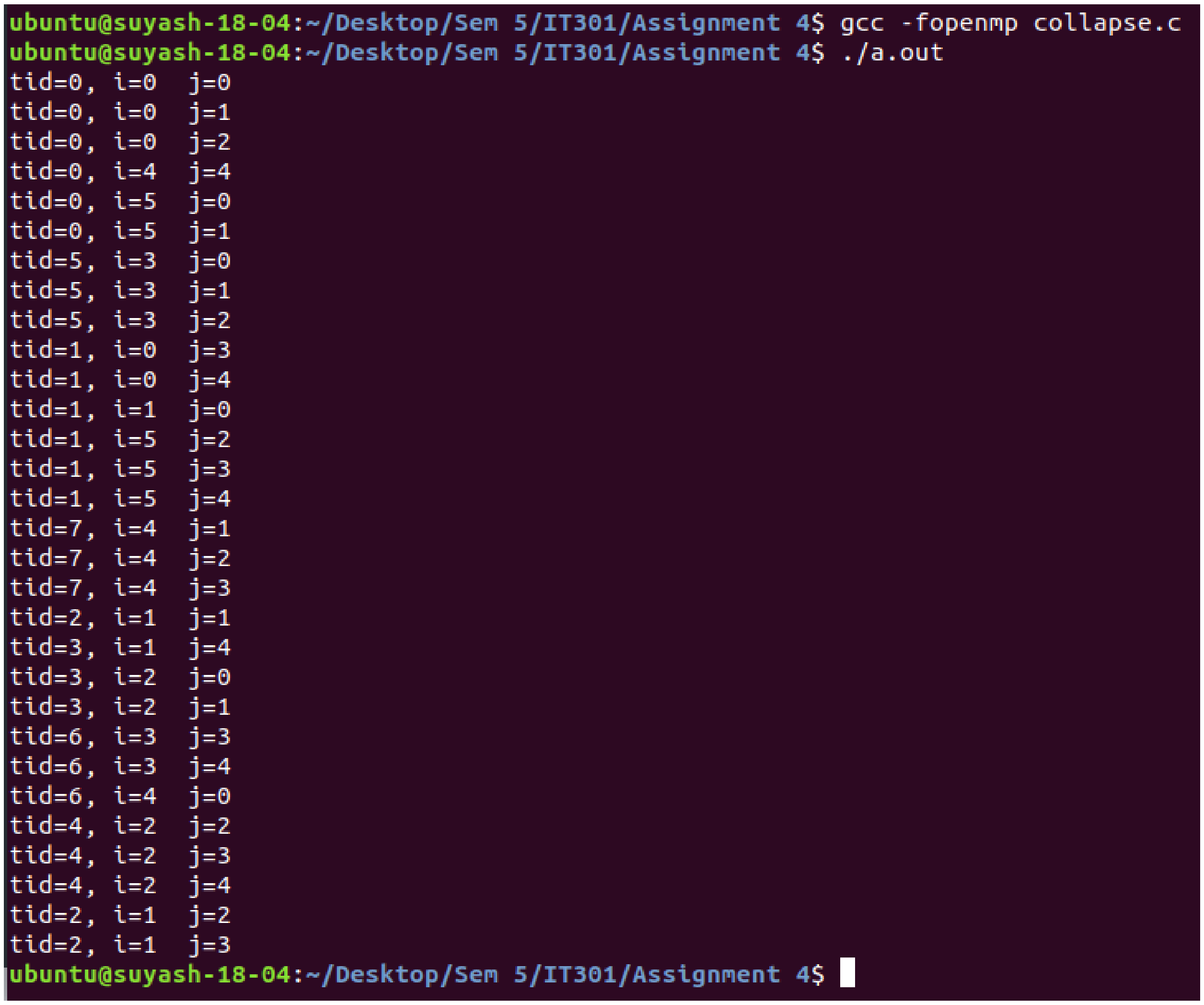
<u>Observations for variable ‘count’:</u>

There are two parallel regions in our program. The variable count is threadprivate. ‘count’ is globally initialized as well as copyin is used so the value count=0 is initialized by all the threads (here only 2 as num_threads=2). 10 iterations are to be performed using static scheduling among 2 threads. This implies that 5 iterations need to be executed by each thread. During the first parallel region, the value of count gets incremented by 5 for each thread as five iterations per thread is assigned. The value of count after the first parallel region is retained as 5 as the value of count in master thread is 5. The copyprivate clause between the two parallel regions increments the value of count by 20 for both the threads (as copyprivate broadcasts its value). After the second parallel region is executed, the value of count=30 is retained in the main program due to similar reasons as explained above.

Observations for variable ‘x’:

The value x=10 is initialized in the main program. ‘x’ is then firstprivate in the first parallel region which initializes the value x=10 for all threads only for that particular parallel region. Value of x gets incremented by 5 for each thread but its final value is not retained and is again rolled back to 10 as x is firstprivate. Continuing the same procedure, the same happens for the other parallel region as well.

Q4. Program to understand the concept of collapse(). Consider three for loops and check the result with no collapse(), collapse(2) and collapse(3).
SOLUTION:



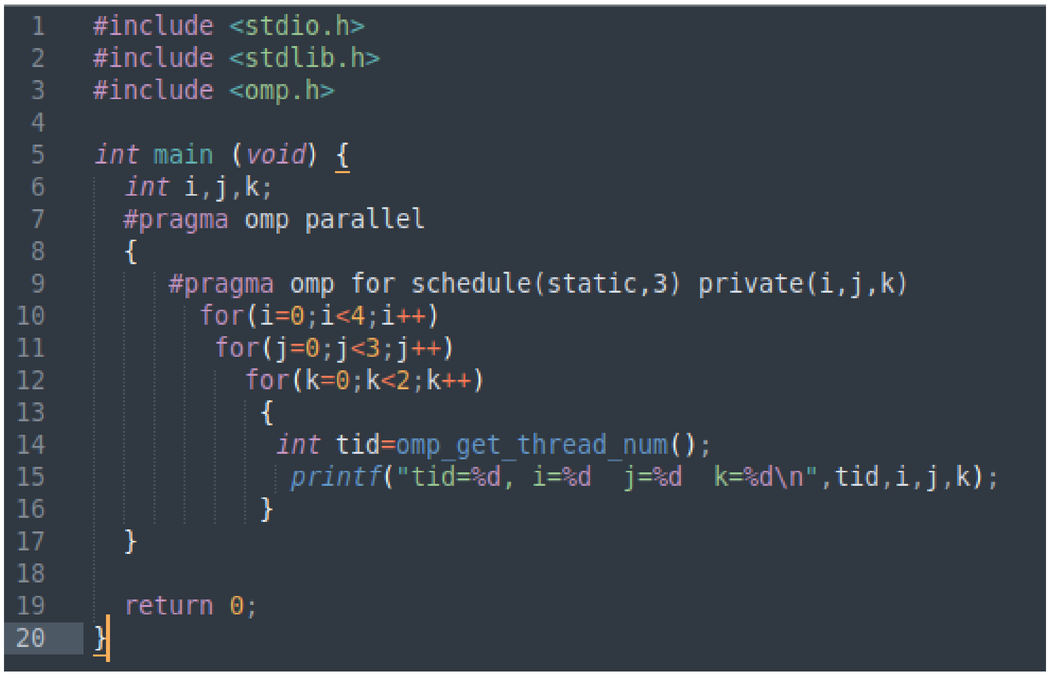
```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ gcc -fopenmp collapse.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ ./a.out
tid=0, i=0  j=0
tid=0, i=0  j=1
tid=0, i=0  j=2
tid=0, i=4  j=4
tid=0, i=5  j=0
tid=0, i=5  j=1
tid=5, i=3  j=0
tid=5, i=3  j=1
tid=5, i=3  j=2
tid=1, i=0  j=3
tid=1, i=0  j=4
tid=1, i=1  j=0
tid=1, i=5  j=2
tid=1, i=5  j=3
tid=1, i=5  j=4
tid=7, i=4  j=1
tid=7, i=4  j=2
tid=7, i=4  j=3
tid=2, i=1  j=1
tid=3, i=1  j=4
tid=3, i=2  j=0
tid=3, i=2  j=1
tid=6, i=3  j=3
tid=6, i=3  j=4
tid=6, i=4  j=0
tid=4, i=2  j=2
tid=4, i=2  j=3
tid=4, i=2  j=4
tid=2, i=1  j=2
tid=2, i=1  j=3
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ 
```

Fig 9. Output of original program (2 for loops)

NOTE: As 3 for loops are needed, the number of iteration for the for loops have been changed. The code for every sub problem has been attached. For the corresponding observations, please refer to the code attached .

(a) no collapse() in 3 for loops

Code:

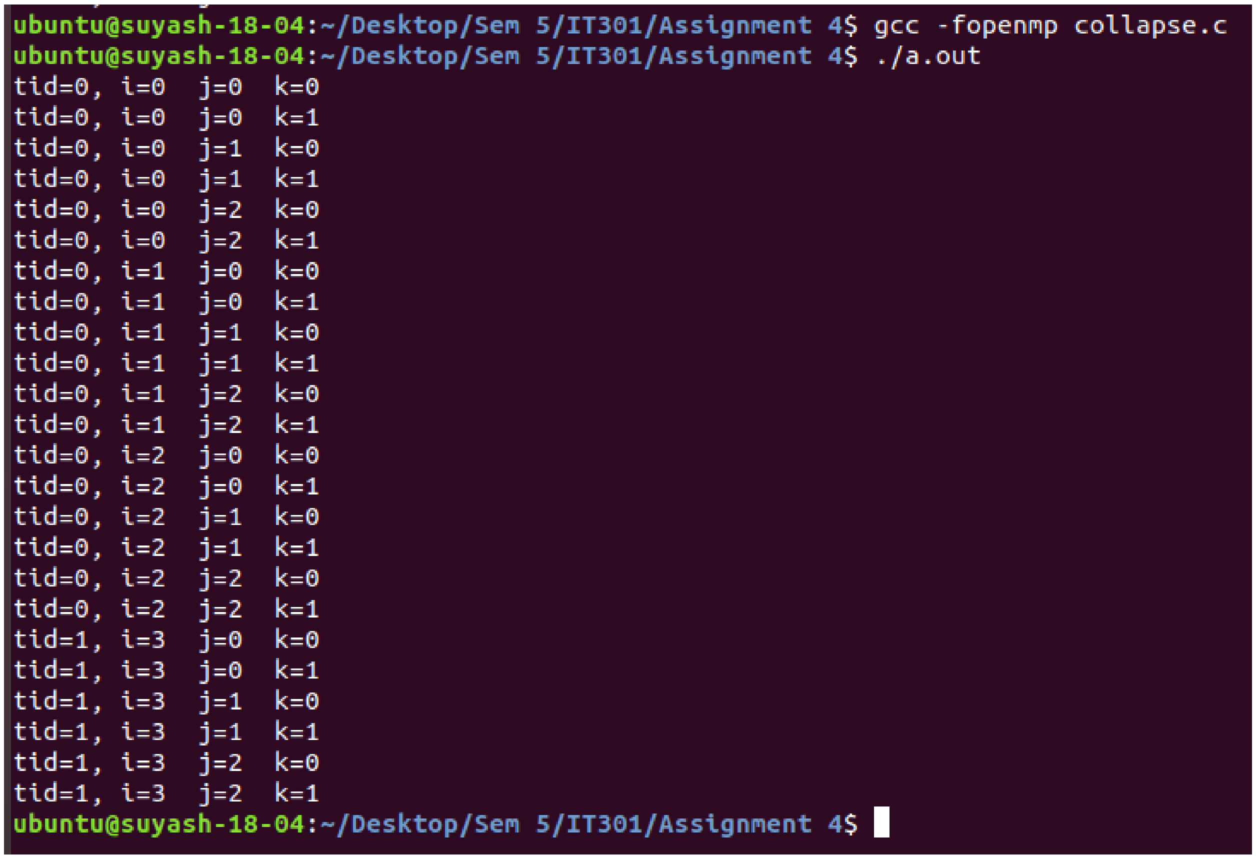


```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

int main (void) {
  int i,j,k;
  #pragma omp parallel
  {
     #pragma omp for schedule(static,3) private(i,j,k)
       for(i=0;i<4;i++)
        for(j=0;j<3;j++)
          for(k=0;k<2;k++)
           {
            int tid=omp_get_thread_num();
             printf("tid=%d, i=%d  j=%d  k=%d\n",tid,i,j,k);
           }
  }

  return 0;
}
```

Fig. 10

Output:

(continued on next page...)



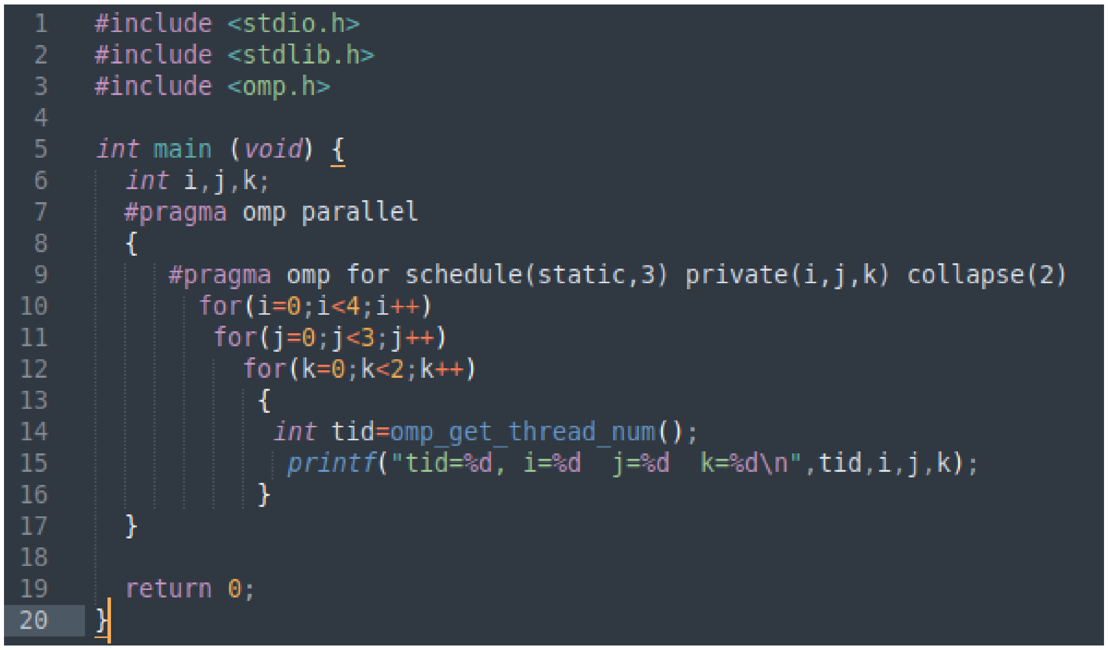
```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ gcc -fopenmp collapse.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ ./a.out
tid=0, i=0  j=0  k=0
tid=0, i=0  j=0  k=1
tid=0, i=0  j=1  k=0
tid=0, i=0  j=1  k=1
tid=0, i=0  j=2  k=0
tid=0, i=0  j=2  k=1
tid=0, i=1  j=0  k=0
tid=0, i=1  j=0  k=1
tid=0, i=1  j=1  k=0
tid=0, i=1  j=1  k=1
tid=0, i=1  j=2  k=0
tid=0, i=1  j=2  k=1
tid=0, i=2  j=0  k=0
tid=0, i=2  j=0  k=1
tid=0, i=2  j=1  k=0
tid=0, i=2  j=1  k=1
tid=0, i=2  j=2  k=0
tid=0, i=2  j=2  k=1
tid=1, i=3  j=0  k=0
tid=1, i=3  j=0  k=1
tid=1, i=3  j=1  k=0
tid=1, i=3  j=1  k=1
tid=1, i=3  j=2  k=0
tid=1, i=3  j=2  k=1
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$
```

Fig 11. No collapse()

Observations: As no collapse is applied, the static schedule gets applied only on one for loop, i.e. the outer for loop (i=0 to 3). As chunk size is 3, thread 0 gets 3 iterations, namely i=0,1,2 and thread 1 gets i=3. Hence, the output in fig 11 is verified.

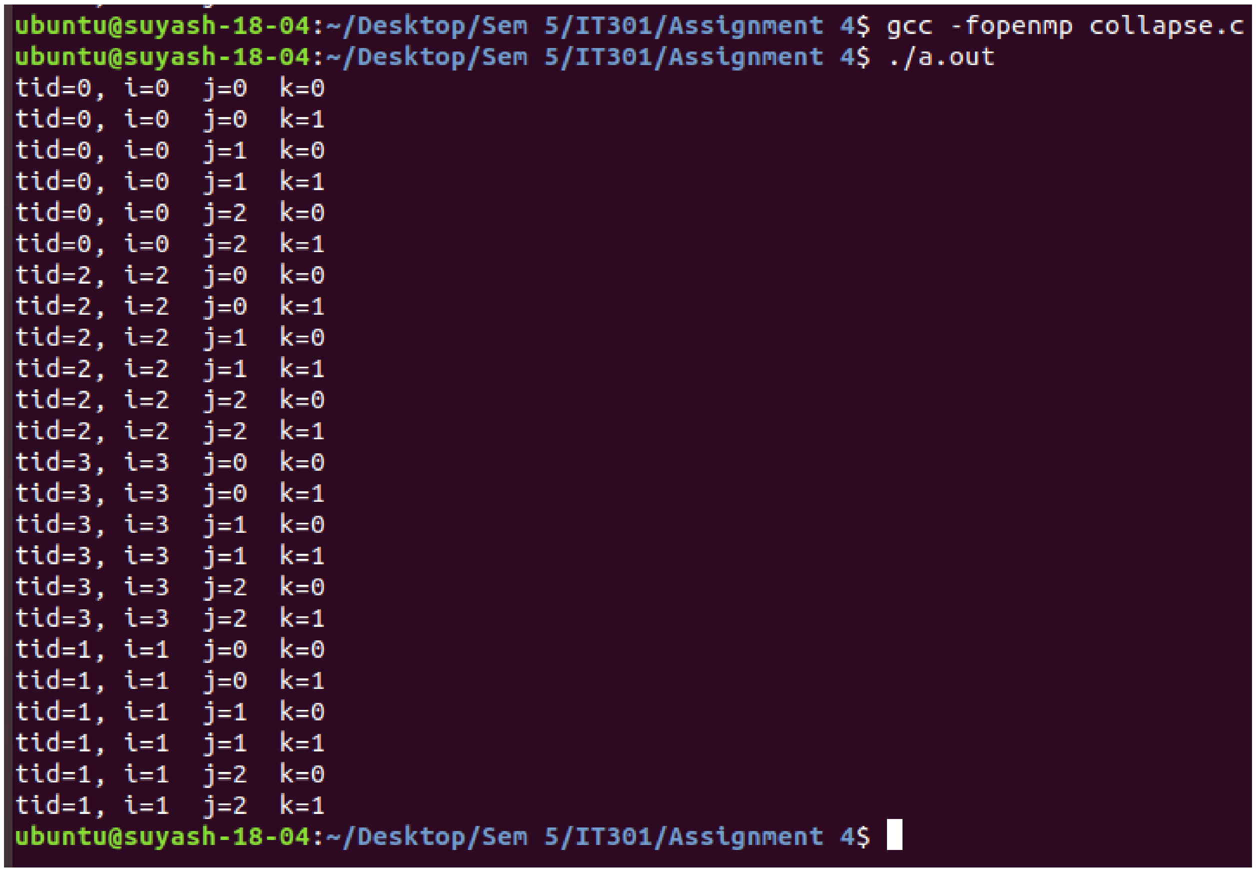
(b) collapse(2) in 3 for loops

Code: Fig 12.



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

int main (void) {
  int i,j,k;
  #pragma omp parallel
  {
     #pragma omp for schedule(static,3) private(i,j,k) collapse(2)
       for(i=0;i<4;i++)
        for(j=0;j<3;j++)
          for(k=0;k<2;k++)
          {
            int tid=omp_get_thread_num();
             printf("tid=%d, i=%d  j=%d  k=%d\n",tid,i,j,k);
          }
  }

  return 0;
}
```

Output:



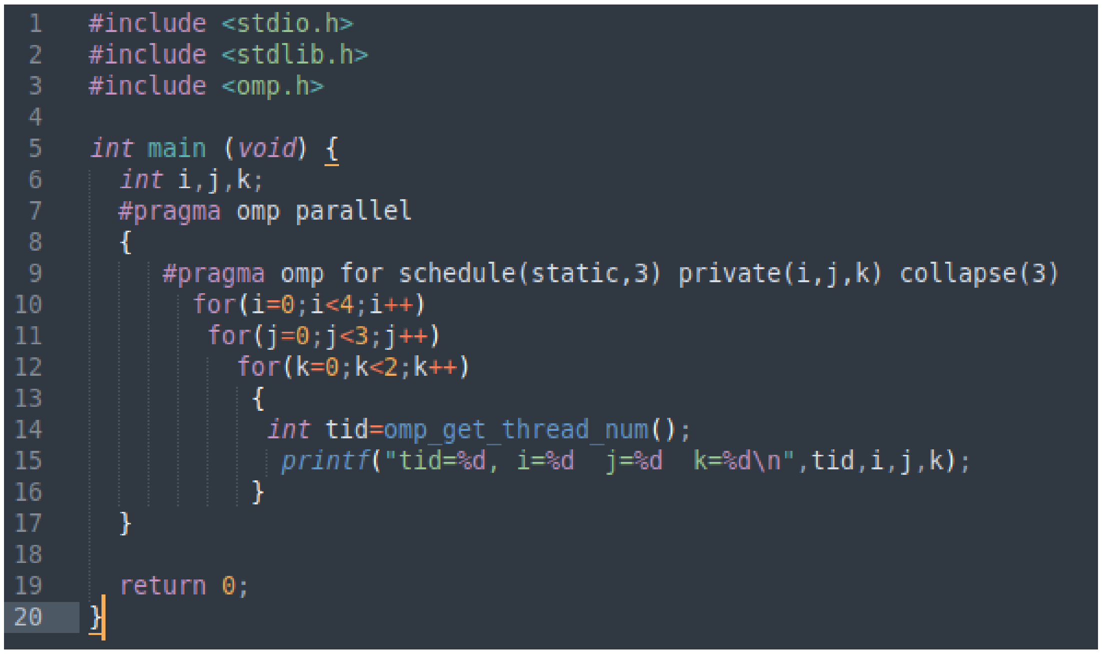
```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ gcc -fopenmp collapse.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ ./a.out
tid=0, i=0  j=0  k=0
tid=0, i=0  j=0  k=1
tid=0, i=0  j=1  k=0
tid=0, i=0  j=1  k=1
tid=0, i=0  j=2  k=0
tid=0, i=0  j=2  k=1
tid=2, i=2  j=0  k=0
tid=2, i=2  j=0  k=1
tid=2, i=2  j=1  k=0
tid=2, i=2  j=1  k=1
tid=2, i=2  j=2  k=0
tid=2, i=2  j=2  k=1
tid=3, i=3  j=0  k=0
tid=3, i=3  j=0  k=1
tid=3, i=3  j=1  k=0
tid=3, i=3  j=1  k=1
tid=3, i=3  j=2  k=0
tid=3, i=3  j=2  k=1
tid=1, i=1  j=0  k=0
tid=1, i=1  j=0  k=1
tid=1, i=1  j=1  k=0
tid=1, i=1  j=1  k=1
tid=1, i=1  j=2  k=0
tid=1, i=1  j=2  k=1
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$
```

Fig 13. collapse(2)

Observations: As two for loops will be considered this time, i.e. 4*3=12 iterations based on the second for loop. Hence, 4 threads with 3 iterations of ‘j’ are distributed.

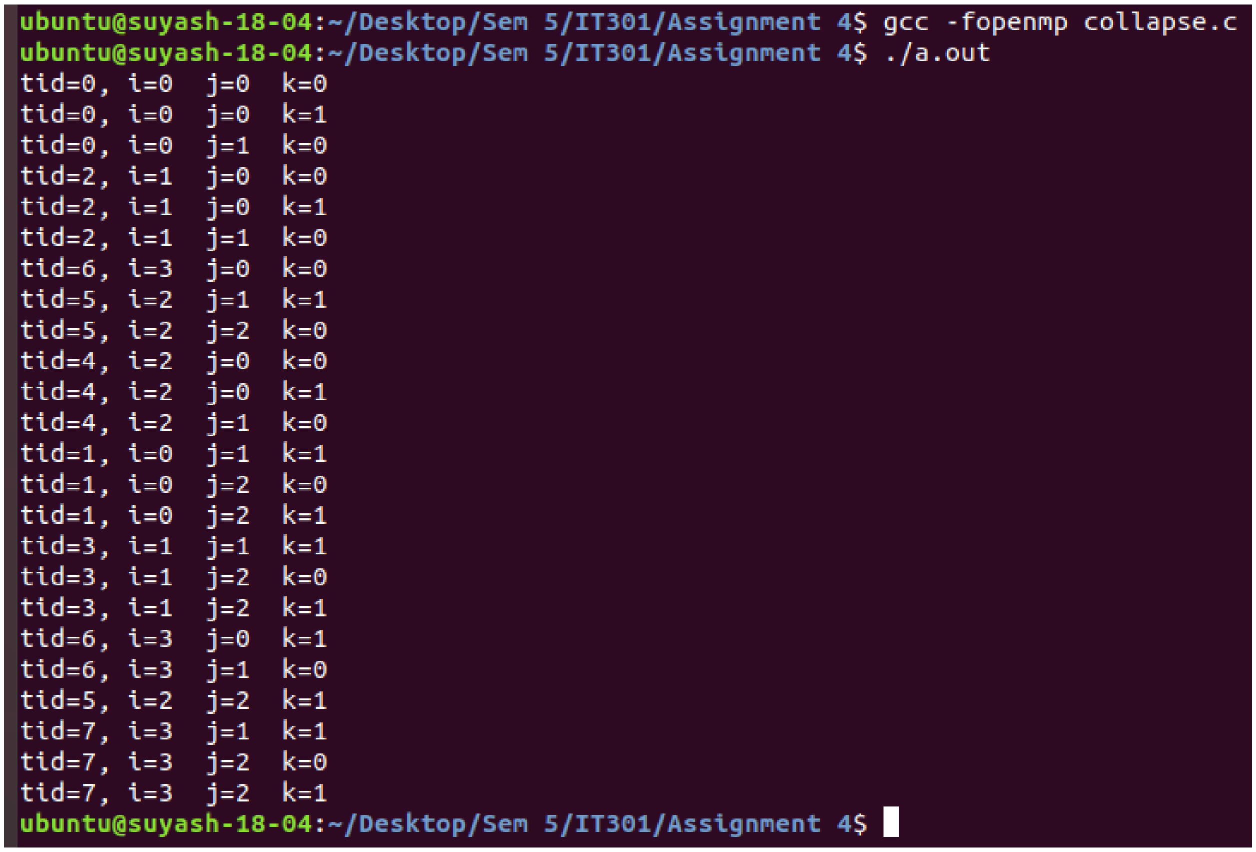
(c) collapse(3) in 3 for loops

Code:



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

int main (void) {
  int i,j,k;
  #pragma omp parallel
  {
     #pragma omp for schedule(static,3) private(i,j,k) collapse(3)
       for(i=0;i<4;i++)
        for(j=0;j<3;j++)
          for(k=0;k<2;k++)
           {
            int tid=omp_get_thread_num();
             printf("tid=%d, i=%d  j=%d  k=%d\n",tid,i,j,k);
           }
  }

  return 0;
}
```

Output:



```
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ gcc -fopenmp collapse.c
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ ./a.out
tid=0, i=0  j=0  k=0
tid=0, i=0  j=0  k=1
tid=0, i=0  j=1  k=0
tid=2, i=1  j=0  k=0
tid=2, i=1  j=0  k=1
tid=2, i=1  j=1  k=0
tid=6, i=3  j=0  k=0
tid=5, i=2  j=1  k=1
tid=5, i=2  j=2  k=0
tid=4, i=2  j=0  k=0
tid=4, i=2  j=0  k=1
tid=4, i=2  j=1  k=0
tid=1, i=0  j=1  k=1
tid=1, i=0  j=2  k=0
tid=1, i=0  j=2  k=1
tid=3, i=1  j=1  k=1
tid=3, i=1  j=2  k=0
tid=3, i=1  j=2  k=1
tid=6, i=3  j=0  k=1
tid=6, i=3  j=1  k=0
tid=5, i=2  j=2  k=1
tid=7, i=3  j=1  k=1
tid=7, i=3  j=2  k=0
tid=7, i=3  j=2  k=1
ubuntu@suyash-18-04:~/Desktop/Sem 5/IT301/Assignment 4$ 
```

Fig 15. collapse(3)

Observations: As there are total 4*3*2 iterations and we have 8 threads (default), every thread will do 3 iterations each.

THANK YOU