IT301 – Parallel Computing

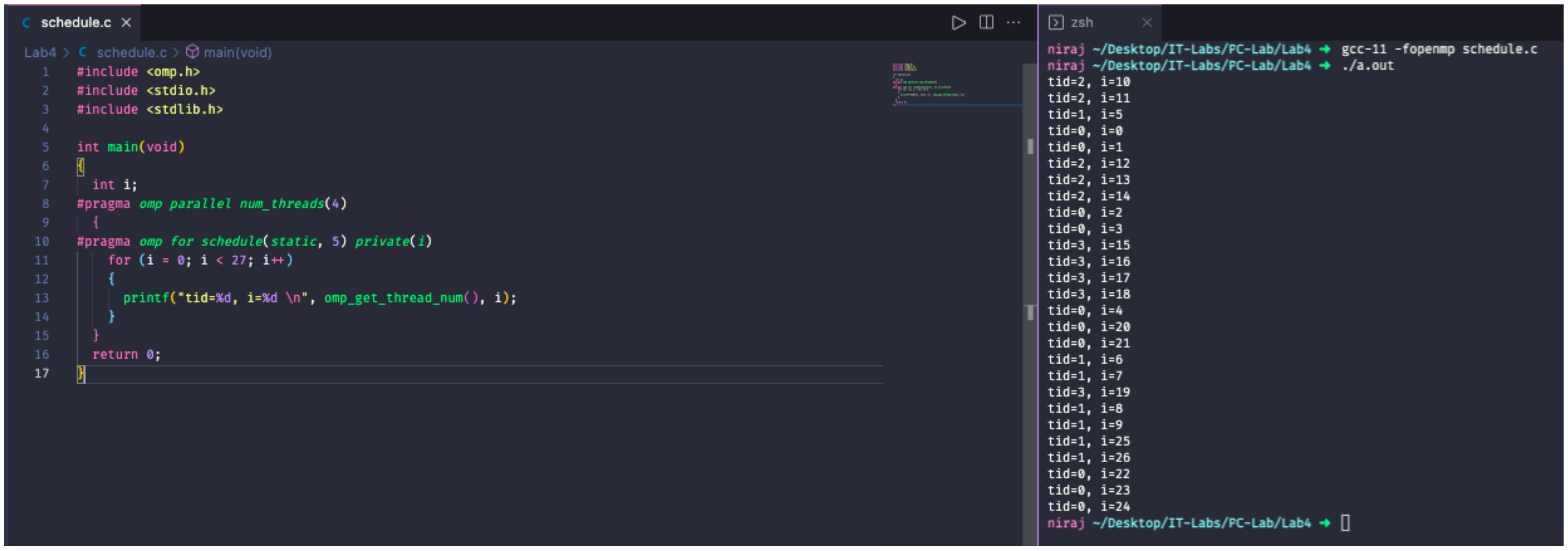
Assignment – 4

Name: Niraj Nandish

Roll No: 191IT234

1. Program 1
    a. Observation – Since we used **static** kind of scheduling here, the iterations are divided into chunks of size 5. The chunks are then assigned to threads in round robin order of thread number. Each chunk is of the given chunk size except for the last chunk which may be less than the given chunk size (5).

T0	T1	T2	T3	T0	T1
0-4	5-9	10-14	15-19	20-24	25-26



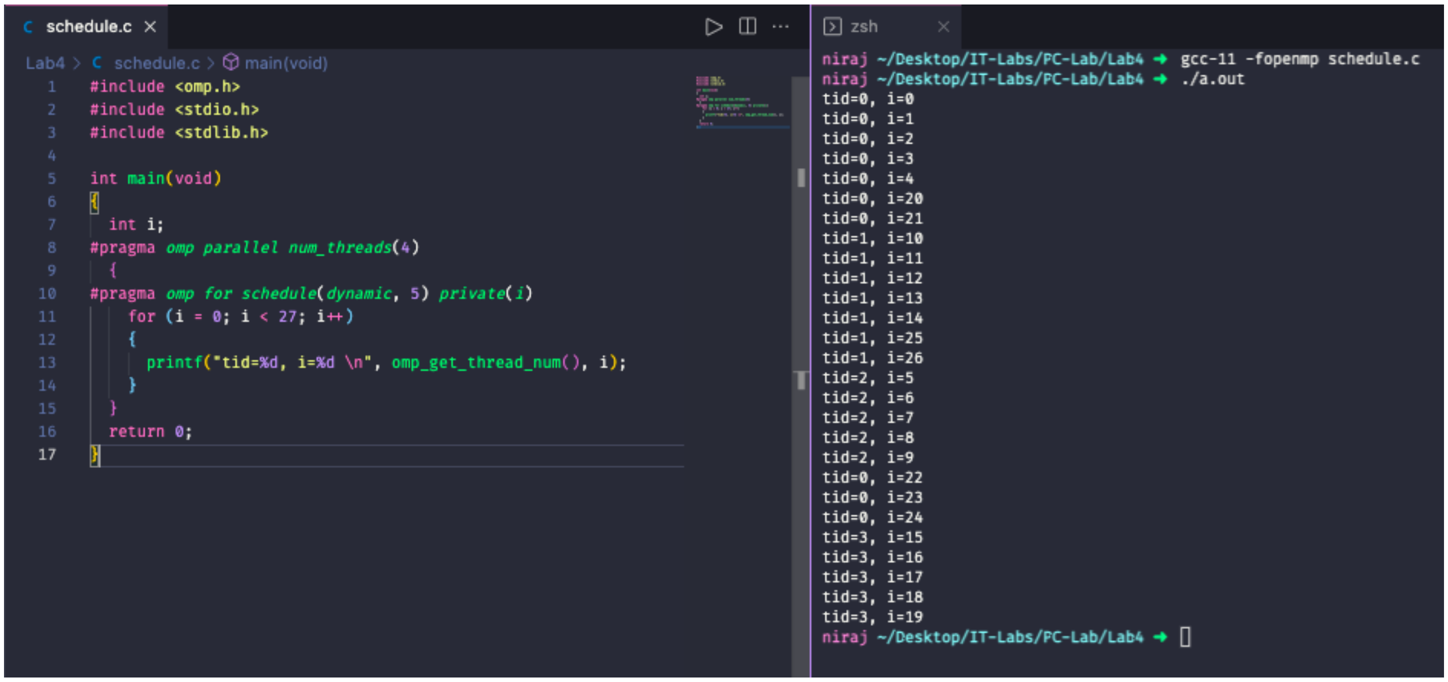
```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int i;
#pragma omp parallel num_threads(4)
  {
#pragma omp for schedule(static, 5) private(i)
    for (i = 0; i < 27; i++)
    {
      printf("tid=%d, i=%d \n", omp_get_thread_num(), i);
    }
  }
  return 0;
}
```

```
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜ gcc-11 -fopenmp schedule.c
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜ ./a.out
tid=2, i=10
tid=2, i=11
tid=1, i=5
tid=0, i=0
tid=0, i=1
tid=2, i=12
tid=2, i=13
tid=2, i=14
tid=0, i=2
tid=0, i=3
tid=3, i=15
tid=3, i=16
tid=3, i=17
tid=3, i=18
tid=0, i=4
tid=0, i=20
tid=0, i=21
tid=1, i=6
tid=1, i=7
tid=3, i=19
tid=1, i=8
tid=1, i=9
tid=1, i=25
tid=1, i=26
tid=0, i=22
tid=0, i=23
tid=0, i=24
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜ 
```

b. Observation – Since we used **dynamic** kind of scheduling here, the iterations are divided into chunks of size 5. The chunks are then assigned to the threads as and when the thread requests for a chunk. Each chunk is of the given chunk size except for the last chunk which may be less than the given chunk size (5).

T0	T2	T1	T3	T0	T1
0-4	5-9	10-14	15-19	20-24	25-26



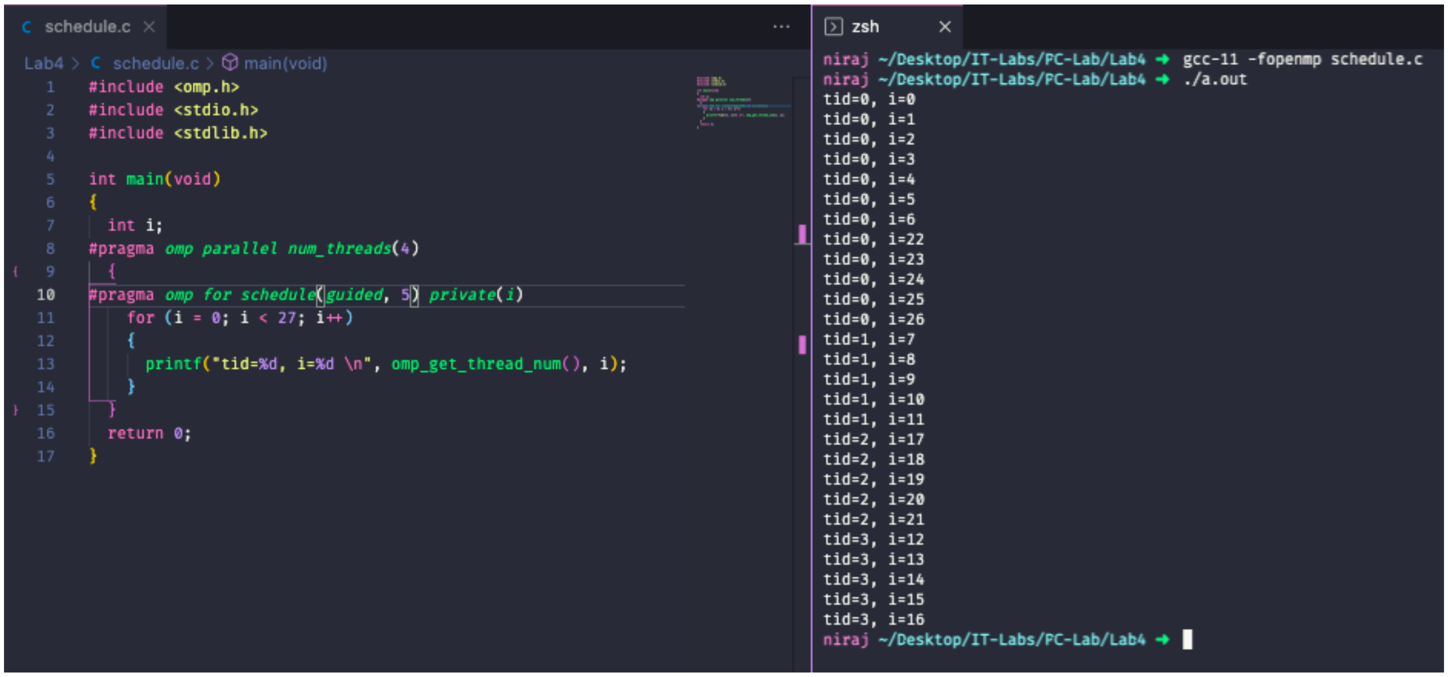
```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int i;
#pragma omp parallel num_threads(4)
  {
#pragma omp for schedule(dynamic, 5) private(i)
    for (i = 0; i < 27; i++)
    {
      printf("tid=%d, i=%d \n", omp_get_thread_num(), i);
    }
  }
  return 0;
}
```

```
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜ gcc-11 -fopenmp schedule.c
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜ ./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=1, i=10
tid=1, i=11
tid=1, i=12
tid=1, i=13
tid=1, i=14
tid=1, i=25
tid=1, i=26
tid=2, i=5
tid=2, i=6
tid=2, i=7
tid=2, i=8
tid=2, i=9
tid=0, i=22
tid=0, i=23
tid=0, i=24
tid=3, i=15
tid=3, i=16
tid=3, i=17
tid=3, i=18
tid=3, i=19
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜
```

c. Observation – Since we used **guided** kind of scheduling here, the iterations are divided into chunks of size 5. The chunks are then assigned to the threads as and when the thread requests for a chunk. The chunk size of each chunk is calculated by using the formula, *chunk = remaining iterations/no of threads.* But if the size of the chunk after calculating is below the given chunk size (5), then the size will the given chunk size (5). If the remaining iterations is less than or equal to the given chunk size (5), the size of the last chunk will be equal to the number of remaining iterations.

T0	T1	T3	T2	T0
0-6 (27/4=6.75 ≈ 7)	7-11 (20/4=5 ≈ 5)	12-16 (15/4=3.75 < 5 ≈ 5)	17-21 (10/4=2.5 < 5 ≈ 5)	22-26 (5 <= 5 ≈ 5)

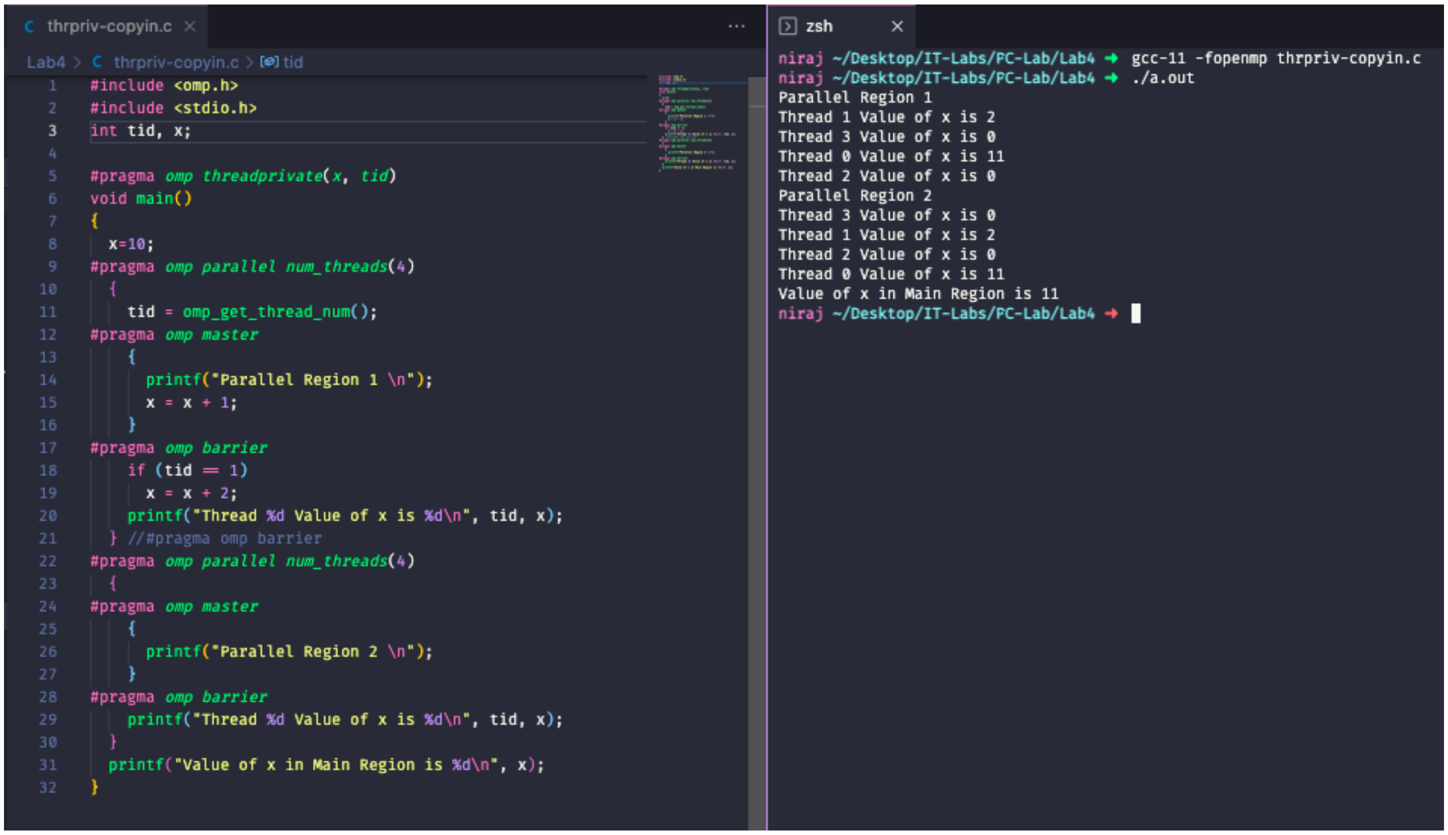


```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int i;
#pragma omp parallel num_threads(4)
  {
#pragma omp for schedule(guided, 5) private(i)
    for (i = 0; i < 27; i++)
    {
      printf("tid=%d, i=%d \n", omp_get_thread_num(), i);
    }
  }
  return 0;
}
```

```
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜ gcc-11 -fopenmp schedule.c
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜ ./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=5
tid=0, i=6
tid=0, i=22
tid=0, i=23
tid=0, i=24
tid=0, i=25
tid=0, i=26
tid=1, i=7
tid=1, i=8
tid=1, i=9
tid=1, i=10
tid=1, i=11
tid=2, i=17
tid=2, i=18
tid=2, i=19
tid=2, i=20
tid=2, i=21
tid=3, i=12
tid=3, i=13
tid=3, i=14
tid=3, i=15
tid=3, i=16
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜
```

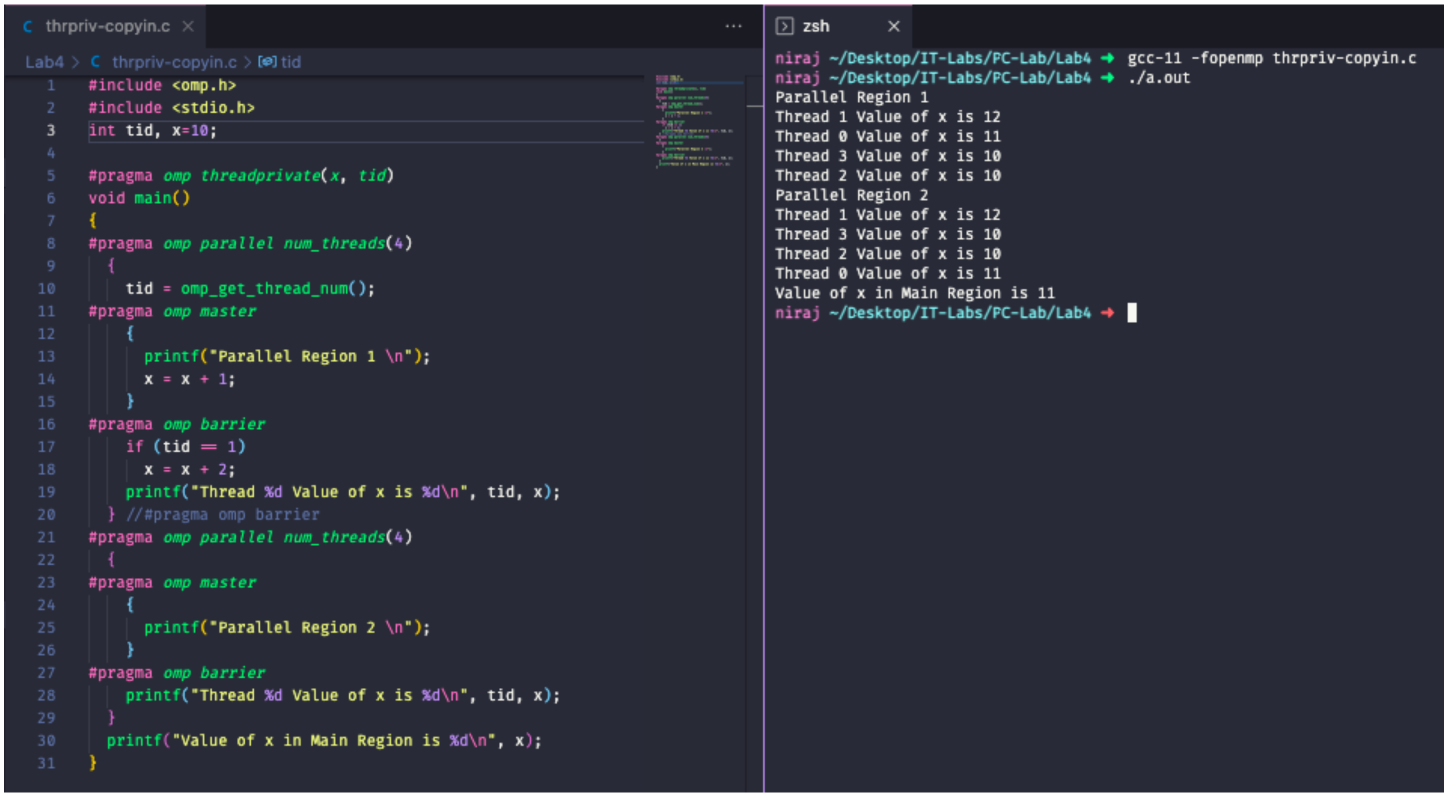
2. Program 2
    a. Observation – Here we have executed the code after removing the **copyin** clause in line 9. The **copyin** clause would copy the value of "x" from the master thread's threadprivate to the individual threads threadprivate. So, then each thread would have a value of 10 stored in the thread's private version of variable "x" before the execution of the parallel block. But as we've removed the clause here, the value of "x" is initialized to 0 for all threads except the master thread. Also, since we've used **threadprivate** directive, each thread gets its own copy of the given variables, and the value gets reflected across all parallel regions in the **threadprivate** block.



```c
1   #include <omp.h>
2   #include <stdio.h>
3   int tid, x;
4
5   #pragma omp threadprivate(x, tid)
6   void main()
7   {
8     x=10;
9   #pragma omp parallel num_threads(4)
10    {
11      tid = omp_get_thread_num();
12  #pragma omp master
13      {
14        printf("Parallel Region 1 \n");
15        x = x + 1;
16      }
17  #pragma omp barrier
18      if (tid == 1)
19        x = x + 2;
20      printf("Thread %d Value of x is %d\n", tid, x);
21    } //#pragma omp barrier
22  #pragma omp parallel num_threads(4)
23    {
24  #pragma omp master
25      {
26        printf("Parallel Region 2 \n");
27      }
28  #pragma omp barrier
29      printf("Thread %d Value of x is %d\n", tid, x);
30    }
31    printf("Value of x in Main Region is %d\n", x);
32  }
```

```
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜ gcc-11 -fopenmp thrpriv-copyin.c
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜ ./a.out
Parallel Region 1
Thread 1 Value of x is 2
Thread 3 Value of x is 0
Thread 0 Value of x is 11
Thread 2 Value of x is 0
Parallel Region 2
Thread 3 Value of x is 0
Thread 1 Value of x is 2
Thread 2 Value of x is 0
Thread 0 Value of x is 11
Value of x in Main Region is 11
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜
```

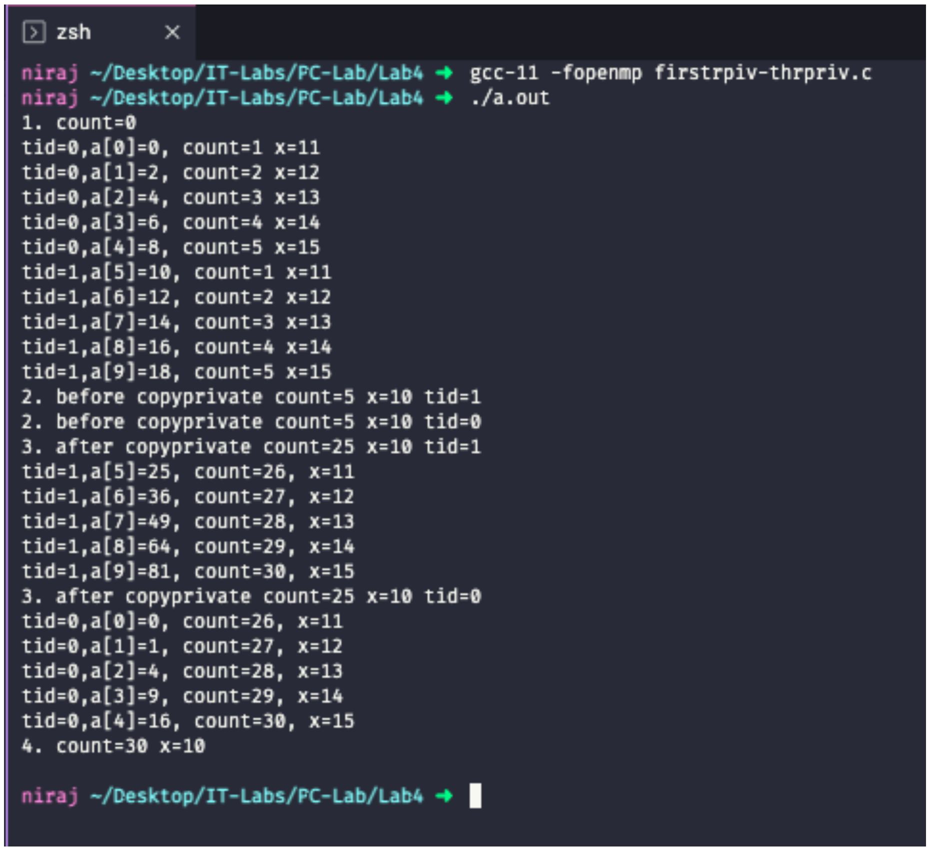
b. Observation – Here we have executed the code after removing the **copyin** clause in line 9 and globally initializing "x" to a value of 10 in line 3. Since "x" has a value when the **threadprivate** block is encountered, so each thread gets a copy of the variable "x" with the value of 10 unlike in the previous sub-question where the value of "x" was initialized only for the master thread.



```
1  #include <omp.h>
2  #include <stdio.h>
3  int tid, x=10;
4
5  #pragma omp threadprivate(x, tid)
6  void main()
7  {
8  #pragma omp parallel num_threads(4)
9      {
10         tid = omp_get_thread_num();
11 #pragma omp master
12         {
13             printf("Parallel Region 1 \n");
14             x = x + 1;
15         }
16 #pragma omp barrier
17         if (tid == 1)
18             x = x + 2;
19         printf("Thread %d Value of x is %d\n", tid, x);
20     } //#pragma omp barrier
21 #pragma omp parallel num_threads(4)
22     {
23 #pragma omp master
24         {
25             printf("Parallel Region 2 \n");
26         }
27 #pragma omp barrier
28         printf("Thread %d Value of x is %d\n", tid, x);
29     }
30     printf("Value of x in Main Region is %d\n", x);
31 }
```

```
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜ gcc-11 -fopenmp thrpriv-copyin.c
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜ ./a.out
Parallel Region 1
Thread 1 Value of x is 12
Thread 0 Value of x is 11
Thread 3 Value of x is 10
Thread 2 Value of x is 10
Parallel Region 2
Thread 1 Value of x is 12
Thread 3 Value of x is 10
Thread 2 Value of x is 10
Thread 0 Value of x is 11
Value of x in Main Region is 11
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜
```

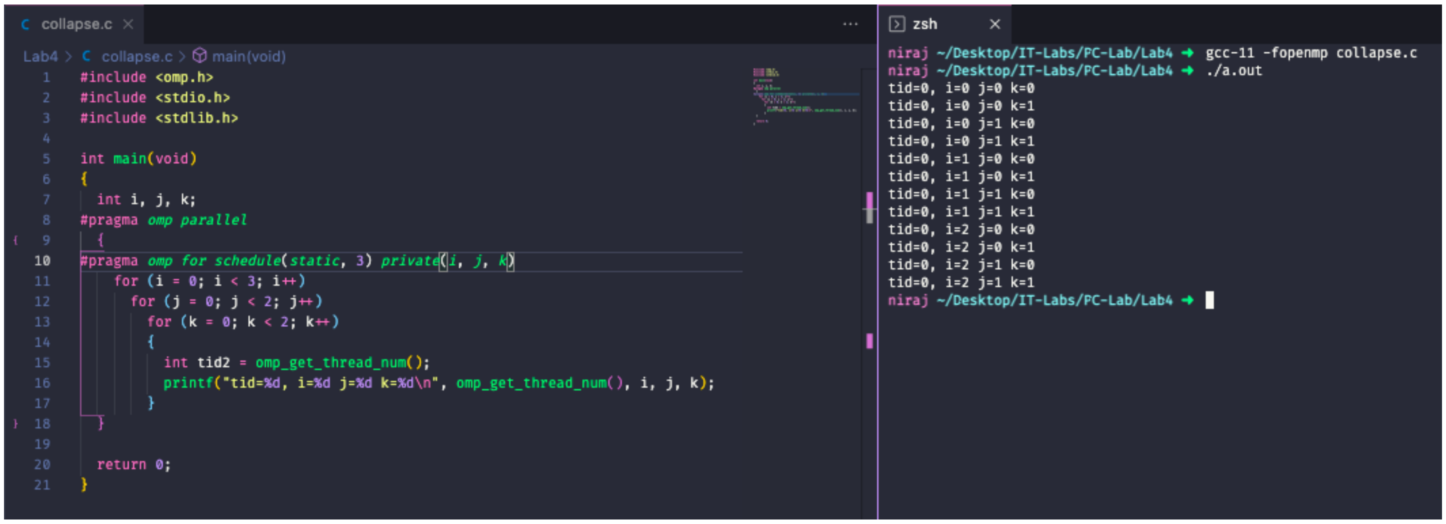
3. Program 3
    a. Observation – Here the variable "count" is globally initialized to the value 0. Next, we encounter the **threadprivate** directive which creates a copy of the "count" variable to both the threads. After that we encounter a for loop with the **firstprivate** clause on the variable "x". So, each thread here gets assigned a copy of the variable "x" with its initial value set to 10. The for loop increments both the "count" and "x" variable and we can see the output of that in the below image. After the for loop, we encounter the **copyprivate** clause where we are adding 20 the value of "count" to one thread only (due to the **single** directive) and broadcasting its final value to all threads. Now both threads have the value of 25 in "count" variable. Next, we again encounter a for loop with the **firstprivate** clause on the variable "x". So, each thread here gets assigned a copy of the variable "x" with its initial value set to 10. The for loop increments both the "count" and "x" variable and we can see the output of that in the below image. Finally, when we print both the values of "count" and "x" we can see that the value of "x" hasn't been changed as **firstprivate** clause copies the original variable to each thread and the changes are restricted to the respective parallel block. But the variable "count" has had its value changed which is due to the **threadprivate** directive which keeps a copy of the "count" variable in each thread and any change in the value gets reflected across all parallel regions as the variable is restricted to within a thread.



```
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜ gcc-11 -fopenmp firstrpiv-thrpriv.c
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜ ./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
tid=1,a[8]=64, count=29, x=14
tid=1,a[9]=81, count=30, x=15
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
4. count=30 x=10

niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜
```

4. Program 4
   a. Observation – The **collapse** clause increases the total number of iterations so that the work will be partitioned among all the threads. This way we can reduce the amount of work to be done by each thread.
   b. In the below picture the **collapse** clause is not used, so none of the for loops are collapsed and the **static** scheduling will try to distribute the number of iterations of only the first for loop among the threads. Hence only one thread is used as the first for loop iterates only 3 times. This results in a waste of time as the available threads are not being used efficiently.



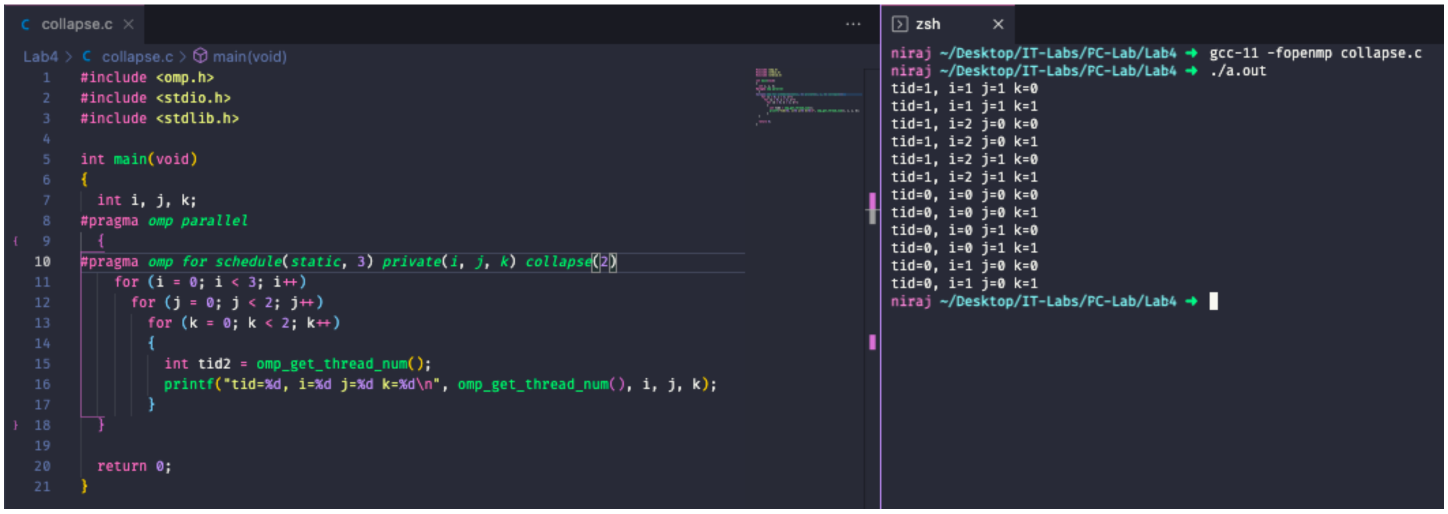
```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
  int i, j, k;
#pragma omp parallel
  {
#pragma omp for schedule(static, 3) private(i, j, k)
    for (i = 0; i < 3; i++)
      for (j = 0; j < 2; j++)
        for (k = 0; k < 2; k++)
        {
          int tid2 = omp_get_thread_num();
          printf("tid=%d, i=%d j=%d k=%d\n", omp_get_thread_num(), i, j, k);
        }
  }

  return 0;
}
```

```
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜ gcc-11 -fopenmp collapse.c
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜ ./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=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=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
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜
```

   c. In the below picture we've used **collapse** clause with argument 2, so the first two for loops will be collapsed into one for loop and the final code will look like it contains two for loops having 6 and 2 iterations respectively. Here the **static** scheduling will try to distribute the 6 iterations among the threads resulting in only 2 threads being used. This results in a waste of time as the available threads are not being used efficiently.



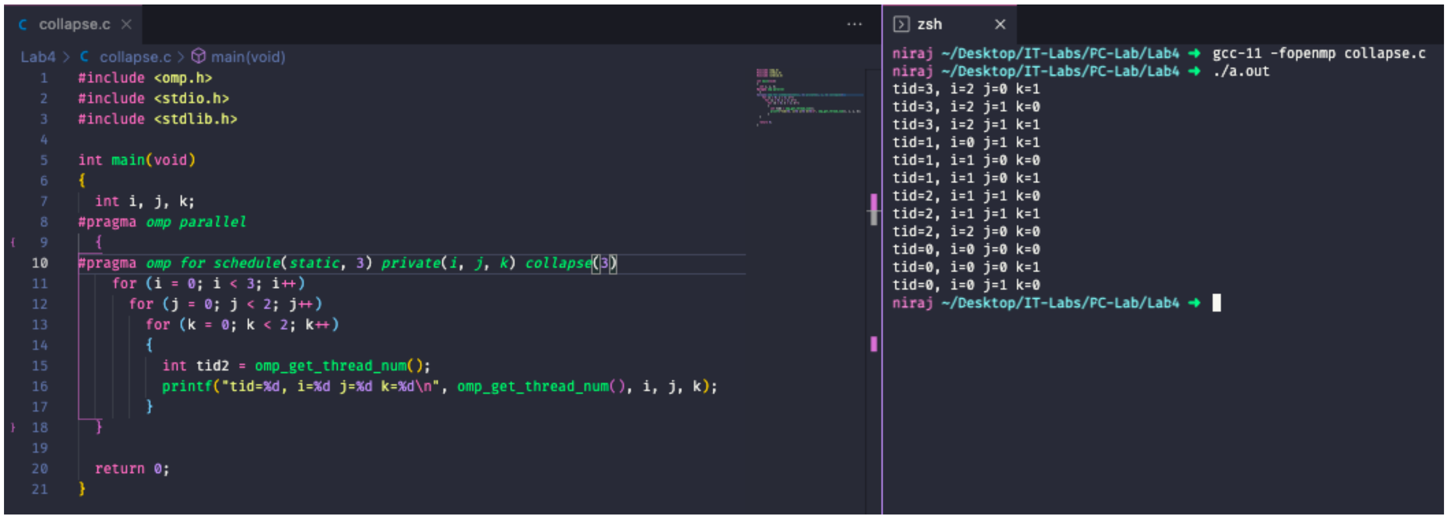
```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>

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 < 3; i++)
      for (j = 0; j < 2; j++)
        for (k = 0; k < 2; k++)
        {
          int tid2 = omp_get_thread_num();
          printf("tid=%d, i=%d j=%d k=%d\n", omp_get_thread_num(), i, j, k);
        }
  }

  return 0;
}
```

```
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜ gcc-11 -fopenmp collapse.c
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜ ./a.out
tid=1, i=1 j=1 k=0
tid=1, i=1 j=1 k=1
tid=1, i=2 j=0 k=0
tid=1, i=2 j=0 k=1
tid=1, i=2 j=1 k=0
tid=1, i=2 j=1 k=1
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=1 j=0 k=0
tid=0, i=1 j=0 k=1
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜
```

d. In the below picture we've used **collapse** clause with argument 3, so the first three for loops will be collapsed into one for loop and the final code will look like it contains one for loops having 12 iterations. Here the **static** scheduling will try to distribute the 12 iterations among the threads resulting in 4 threads being used. This results in the available threads being used efficiently by equally partitioning the total number of iterations and assigning them to the threads.



```c
#include <omp.h>
#include <stdio.h>
#include <stdlib.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 < 3; i++)
      for (j = 0; j < 2; j++)
        for (k = 0; k < 2; k++)
        {
          int tid2 = omp_get_thread_num();
          printf("tid=%d, i=%d j=%d k=%d\n", omp_get_thread_num(), i, j, k);
        }
  }

  return 0;
}
```

```
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜ gcc-11 -fopenmp collapse.c
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜ ./a.out
tid=3, i=2 j=0 k=1
tid=3, i=2 j=1 k=0
tid=3, i=2 j=1 k=1
tid=1, i=0 j=1 k=1
tid=1, i=1 j=0 k=0
tid=1, i=1 j=0 k=1
tid=2, i=1 j=1 k=0
tid=2, i=1 j=1 k=1
tid=2, i=2 j=0 k=0
tid=0, i=0 j=0 k=0
tid=0, i=0 j=0 k=1
tid=0, i=0 j=1 k=0
niraj ~/Desktop/IT-Labs/PC-Lab/Lab4 ➜
```