Nithin S
221IT085

IT301 Lab Assignment 4

Q1.To understand the concept of schedule. Write the observation using schedule (static, 5), schedule(dynamic,5) and schedule (guided,5)

Using (static,5)

Analysis

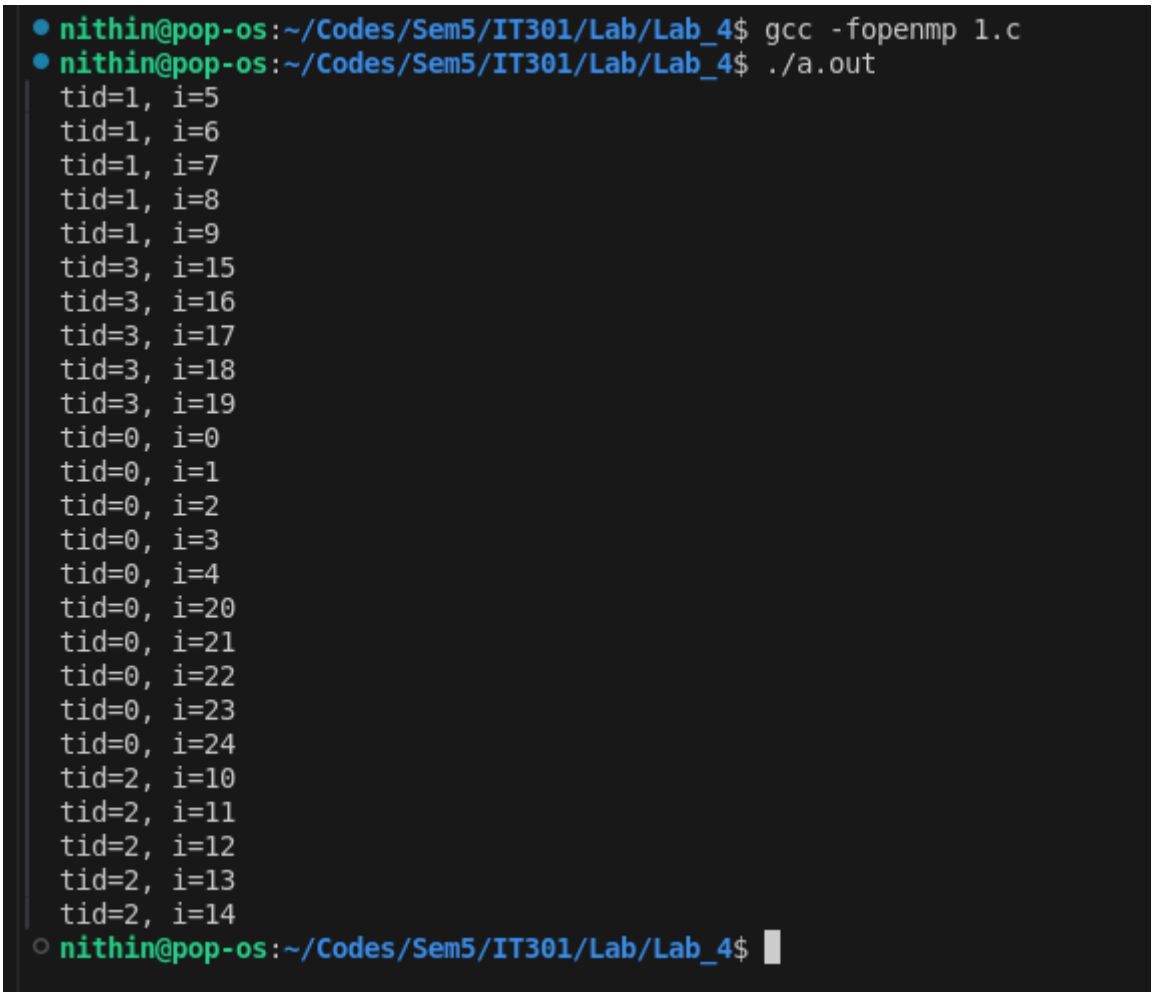
If we had used schedule(static, 5), it would divide the iterations into chunks of 5 and distribute them in a round-robin manner among the threads. Each thread would be assigned a chunk of 5 iterations before moving on to the next thread. We the round robin allocation of iterations in the screenshot attached.

Code

```
#include <omp.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 < 25; i++)
      {
         printf("tid=%d, i=%d\n", omp_get_thread_num(), i);
      }
   }
```

```
    return 0;
}
```

Output



Using (dynamic,5)

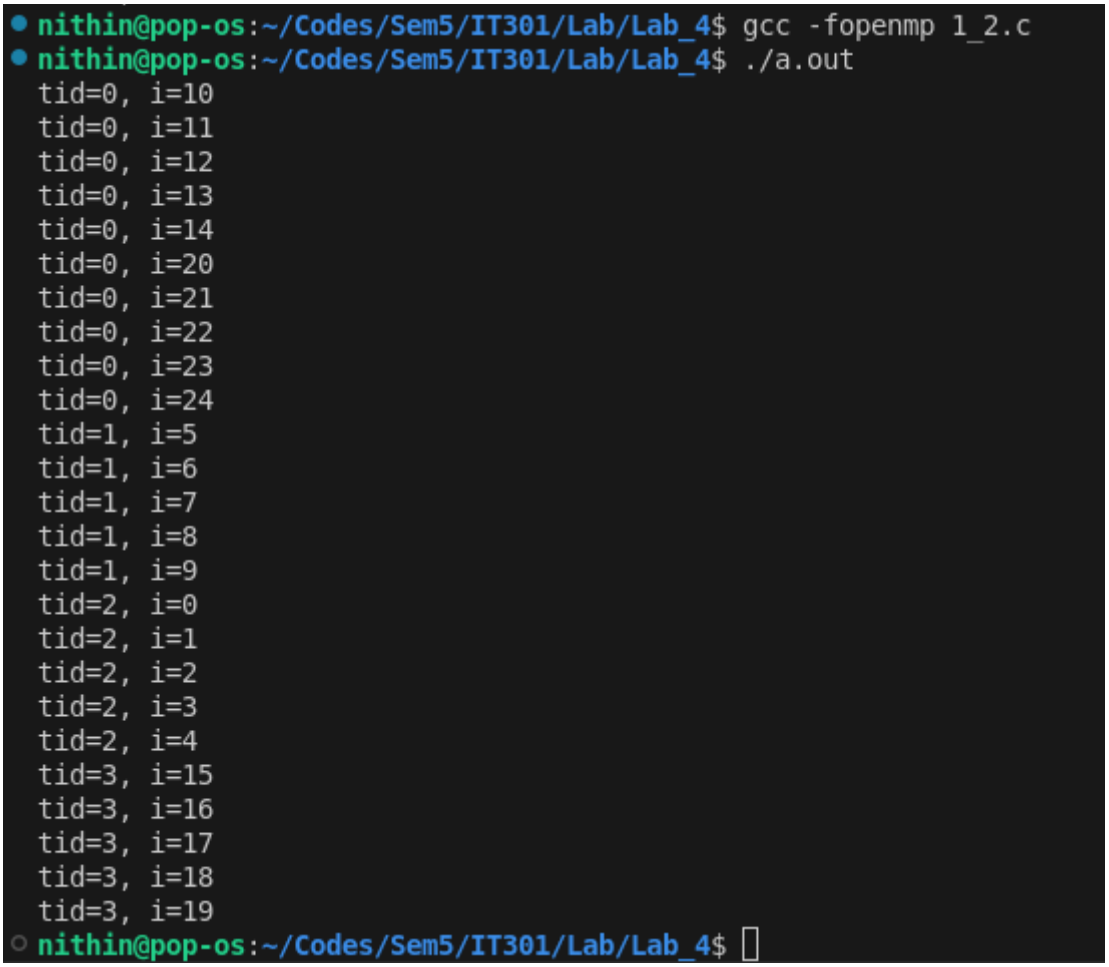
Analysis

With schedule(dynamic, 5), each thread is assigned an initial chunk of 5 iterations. When a thread finishes its assigned chunk, it dynamically grabs the next available chunk of 5 iterations. This can help balance the load if some iterations take longer than others. In the output screenshot thread 0 happened to be faster thread so it executed more iterations.

Code

```
#include <stdio.h>
#include <omp.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 < 25; i++)
        {
            printf("tid=%d, i=%d\n", omp_get_thread_num(), i);
        }
    }
    return 0;
}
```

Output



Using (guided,5)

Analysis

schedule(guided, 5) starts by assigning larger chunks to threads, and as the loop progresses, the chunk size decreases. The minimum chunk size in this case is 5. This approach tries to balance the workload by giving more iterations to faster threads initially and reducing the load as the loop continues, which can be clearly seen in the screenshot provided.

Code

```
#include <stdio.h>
#include <omp.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 < 25; i++)
        {
            printf("tid=%d, i=%d\n", omp_get_thread_num(), i);
        }
    }
    return 0;
}
```

Output

```
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_4$ gcc 1_3.c -fopenmp
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_4$ ./a.out
tid=2, i=12
tid=2, i=13
tid=2, i=14
tid=2, i=15
tid=2, i=16
tid=2, i=22
tid=2, i=23
tid=2, i=24
tid=1, i=7
tid=1, i=8
tid=1, i=9
tid=1, i=10
tid=1, i=11
tid=0, i=0
tid=0, i=1
tid=3, i=17
tid=3, i=18
tid=3, i=19
tid=3, i=20
tid=3, i=21
tid=0, i=2
tid=0, i=3
tid=0, i=4
tid=0, i=5
tid=0, i=6
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_4$
```

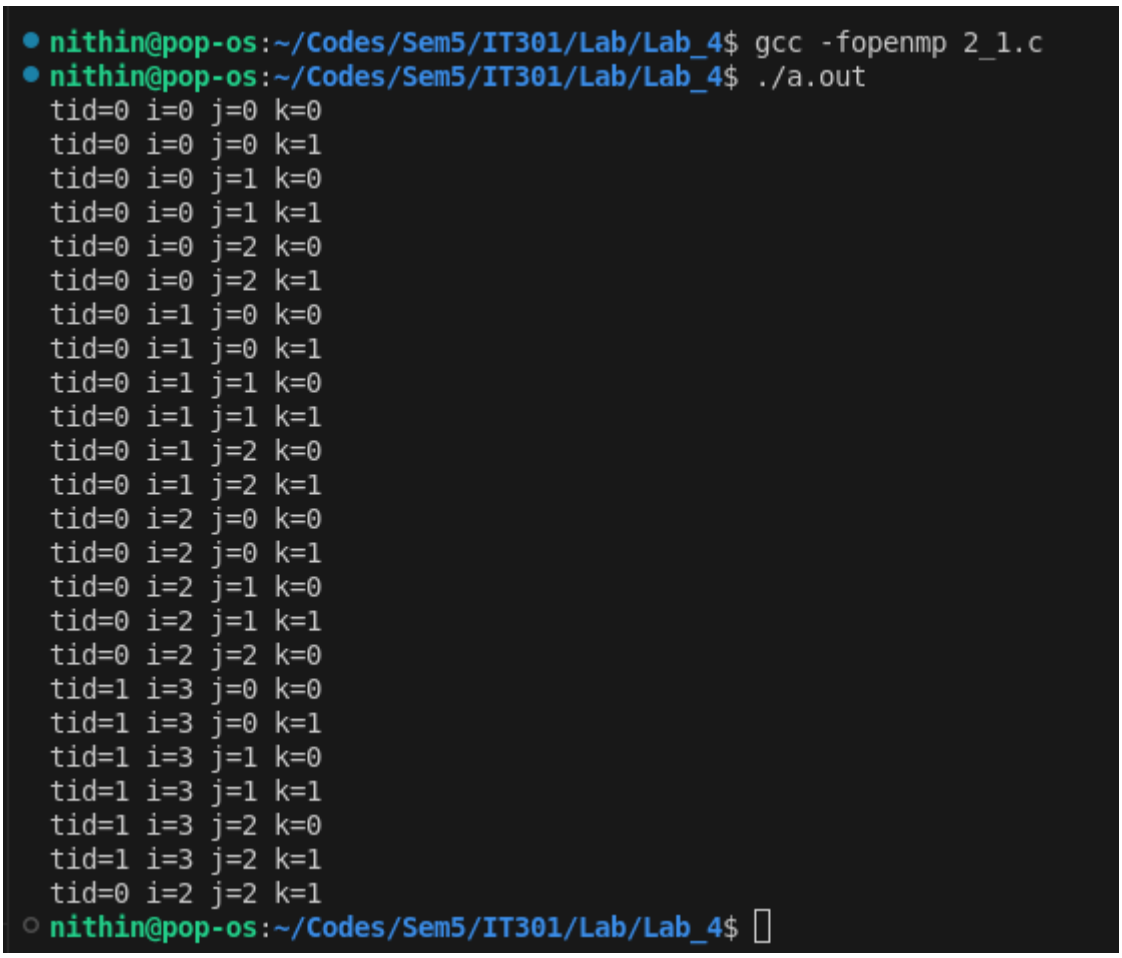
Q2. Execute the following code to understand the concept of collapse(). Consider three for loops and check the result with no collapse(), collapse(2), and collapse(3)

no collapse()

Code

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



Analysis

Without the collapse clause, only the outer loop (i) is parallelized. Each thread executes an entire iteration of the outer loop, which includes all iterations of the inner loops (j and k).
The total number of iterations that can be divided among threads is determined by the outermost loop (i), which has 4 iterations. As seen in the screenshot only two threads are doing all the work.
Poor parallization

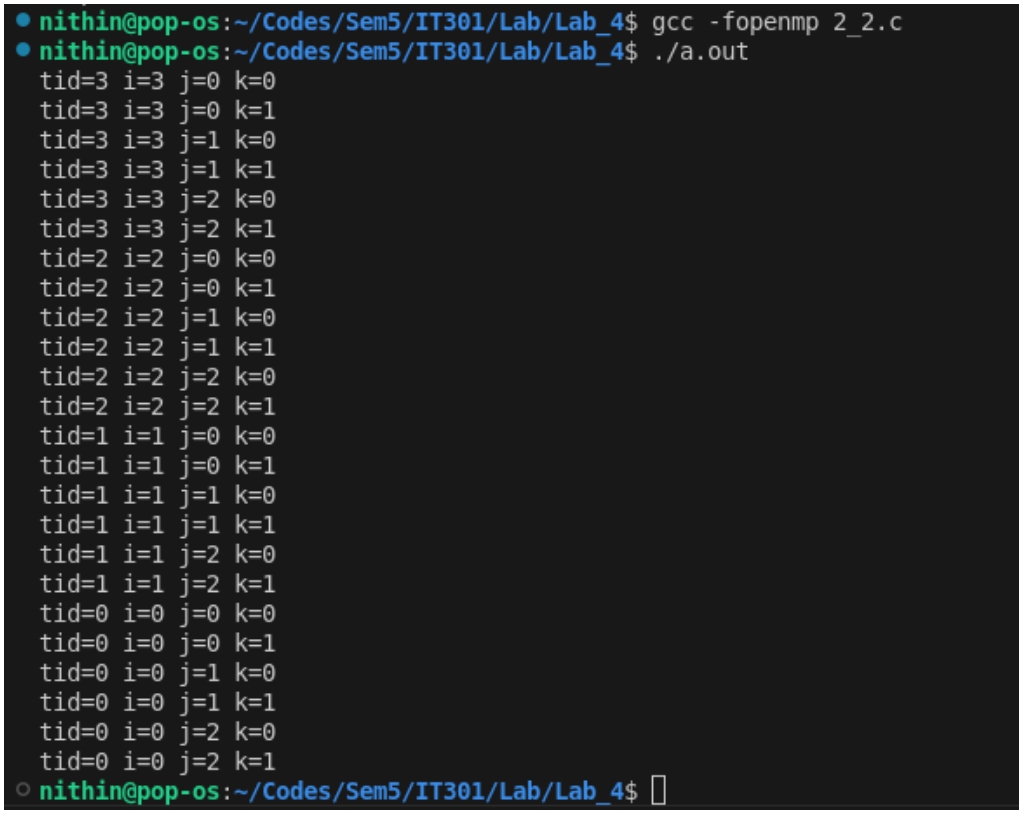
Collapse(2)

Code

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

```c
#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 < 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



Analysis

With collapse(2), OpenMP collapses the first two loops (i and j) into a single loop. This results in 4 * 3 = 12 iterations that can be distributed across threads.
The chunks of these 12 iterations are distributed according to the scheduling policy, allowing more work to be distributed among the threads compared to the non-collapsed version. 4 threads are doing all the work. More parallelization than previous method.

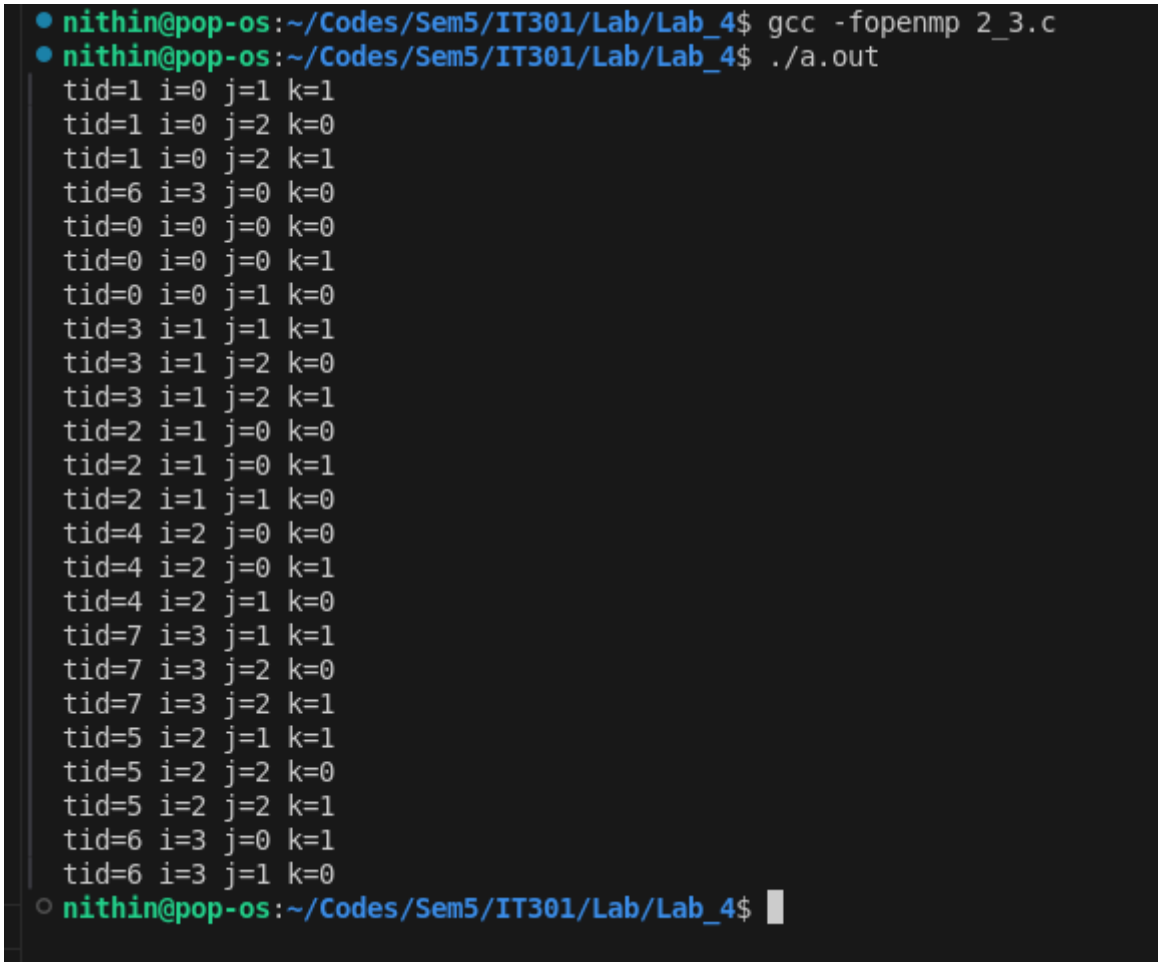
Collapse(3)

Code

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



Analysis

With collapse(3), OpenMP collapses all three loops (i, j, and k) into a single loop. This results in 4 * 3 * 2 = 24 iterations, which can be distributed across the threads.
This provides the most parallelization opportunities, as the work is divided into smaller chunks that can be executed by the threads concurrently. Better parallelization than the two methods before.

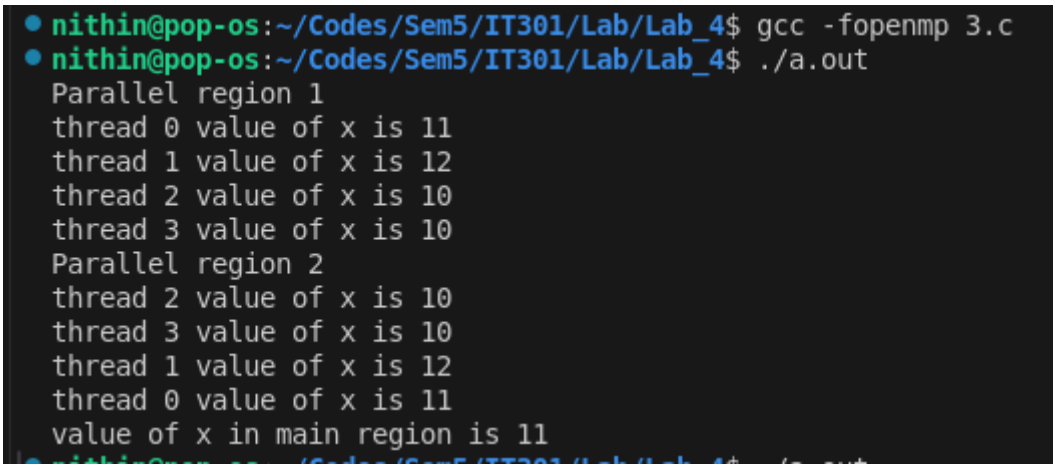
Q3. Execute the following code and observe the working of threadprivate directive and copyin clause.

Code

```
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
int tid, x;
#pragma omp threadprivate(x, tid)
void main()
{
   x = 10;
   #pragma omp parallel num_threads(4) copyin(x)
   {
      tid = omp_get_thread_num();
      #pragma omp master
      {
         printf("Parallel region 1 \n");
         x = x + 1;
      }
      #pragma omp barrier
      if (tid == 1)
         x = x + 2;
      printf("thread %d value of x is %d\n", tid, x);
   }
   #pragma omp parallel num_threads(4)
   {
   #pragma omp master
      {
         printf("Parallel region 2 \n");
      }
   #pragma omp barrier
      printf("thread %d value of x is %d\n", tid, x);
   }
```

```
    printf("value of x in main region is %d\n", x);
}
```

Output



Analysis

tid and x are two global variables. **#pragma omp threadprivate(x, tid) makes these variables private to each thread, meaning each thread has its own copy of tid and x**.

The main function begins by initializing x = 10.

First Parallel Region:

#pragma omp parallel num_threads(4) copyin(x): A parallel region is created with 4 threads.

The copyin(x) clause initializes each thread's private copy of x with the value of x from the main thread (which is 10).

Inside the parallel region, Each thread gets its own copy of tid and x. The master thread (thread 0) increments x by 1.
After the barrier, if the thread ID (tid) is 1, x is further incremented by 2. Each thread then prints its thread ID (tid) and

the value of x. The master thread 0 will print x value as 11 and the thread 1 will print x value as 12.

Second Parallel Region:

Another parallel region with 4 threads is created.
Each thread prints its thread ID (tid) and the value of x after the barrier. The values of x were set in the first parallel region.

The main thread's value of x is 11 because it was modified in the main thread , i.e incremented by 1 by the master thread itself.

This code snippet showcases how threadprivate allows each thread to maintain its own state, and how the copyin clause can be used to initialize those states based on a value from the master thread.

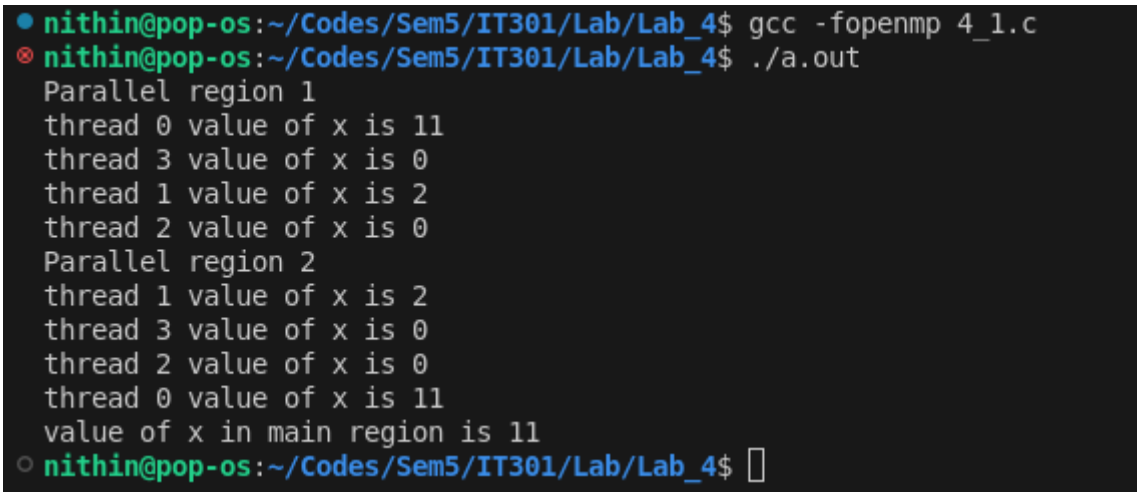
Q4. In the above program (Program No.3),
(i) remove the copyin clause and check the output.

Code

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

int tid, x;
#pragma omp threadprivate(x, tid)

void main() {
   x = 10;

   #pragma omp parallel num_threads(4)
   {
      tid = omp_get_thread_num();
```

```
        #pragma omp master
        {
            printf("Parallel region 1 \n");
            x = x + 1;
        }

        #pragma omp barrier

        if (tid == 1)
            x = x + 2;

        printf("thread %d value of x is %d\n", tid, x);
    }

    #pragma omp parallel num_threads(4)
    {
        #pragma omp master
        {
            printf("Parallel region 2 \n");
        }

        #pragma omp barrier

        printf("thread %d value of x is %d\n", tid, x);
    }

    printf("value of x in main region is %d\n", x);
}
```

Output



Analysis

When the copyin clause is removed, each thread's copy of x is uninitialized.
Only thread 0 (the master) initializes its x to 11 (10 + 1).
The other threads (tid = 1, 2, 3) have an uninitialized x, which behaves as 0 or could be arbitrary.

Master thread prints x = 11 after adding 1 to its initial value of 10.

For Thread 1, x starts at 0 (uninitialized), adds 2 in the if block, resulting in x = 2.

Threads 2 and 3, both print x = 0, as they do not modify x.

(ii) remove the copyin clause, initialize x globally and check the output.

Code

```
#include <stdio.h>
#include <omp.h>
#include <stdlib.h>
int tid, x = 10;
#pragma omp threadprivate(x, tid)
void main()
{
   #pragma omp parallel num_threads(4)
   {
      tid = omp_get_thread_num();
   #pragma omp master
      {
         printf("Parallel region 1 \n");
         x = x + 1;
      }
   #pragma omp barrier
      if (tid == 1)
         x = x + 2;
      printf("thread %d value of x is %d\n", tid, x);
   }
   #pragma omp parallel num_threads(4)
   {
   #pragma omp master
      {
         printf("Parallel region 2 \n");
      }
   #pragma omp barrier
      printf("thread %d value of x is %d\n", tid, x);
   }
   printf("value of x in main region is %d\n", x);
```

}

Output

```
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_4$ gcc -fopenmp 4_2.c
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_4$ ./a.out
Parallel region 1
thread 0 value of x is 11
thread 3 value of x is 10
thread 1 value of x is 12
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
nithin@pop-os:~/Codes/Sem5/IT301/Lab/Lab_4$
```

Analysis

int tid, x = 10; initializes the global variable x to 10.
The variables tid and x are marked as threadprivate.

The #pragma omp threadprivate(x, tid) directive ensures that each thread has its own copy of x and tid, separate from the main thread

The global initialization (x = 10) is reflected in each thread's local copy when the threads are first created. This happens because x is declared threadprivate, and its initial value in each thread is copied from the global variable.

As the master thread, thread 0 increments its x by 1, resulting in x = 11.

Thread 1, starts with x = 10, increments it by 2 in the if block, resulting in x = 12.

Threads 2 and 3, both start with x = 10 and do not modify it, so they print x = 10.