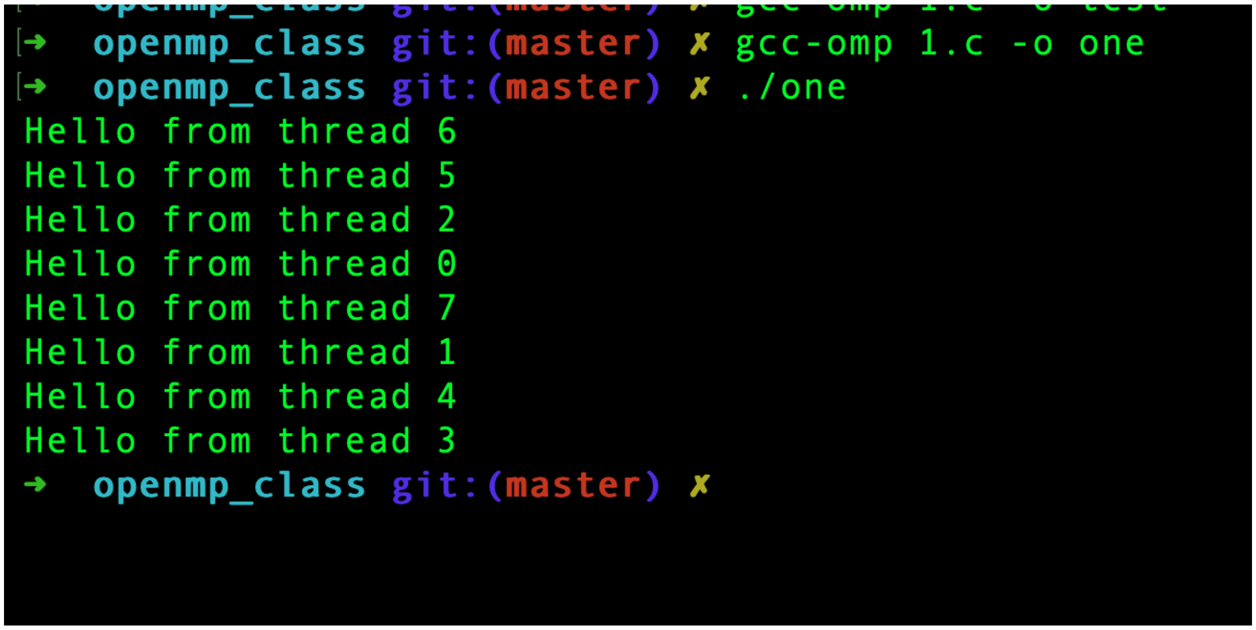
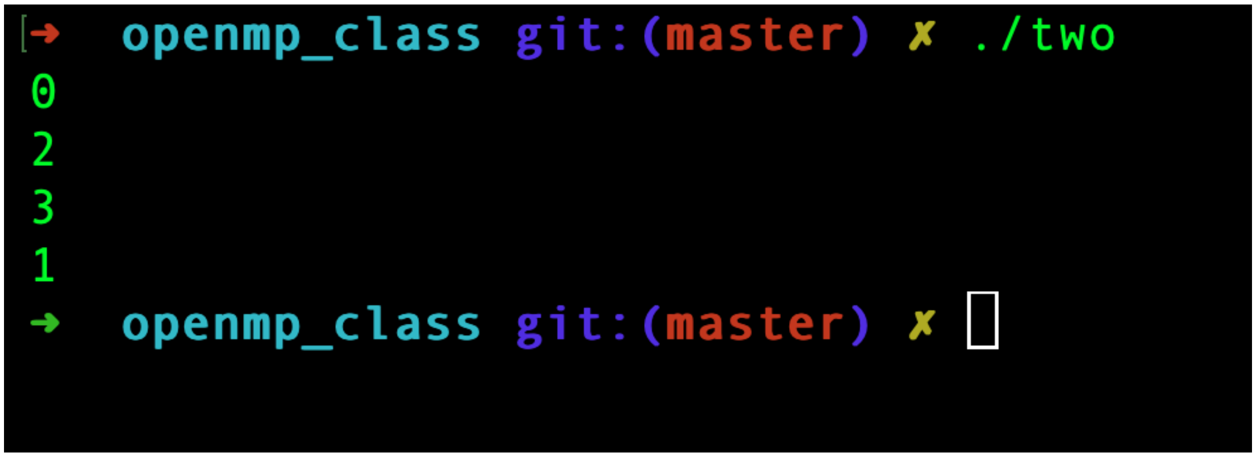
**Pratik Dhimal**
**2407779**



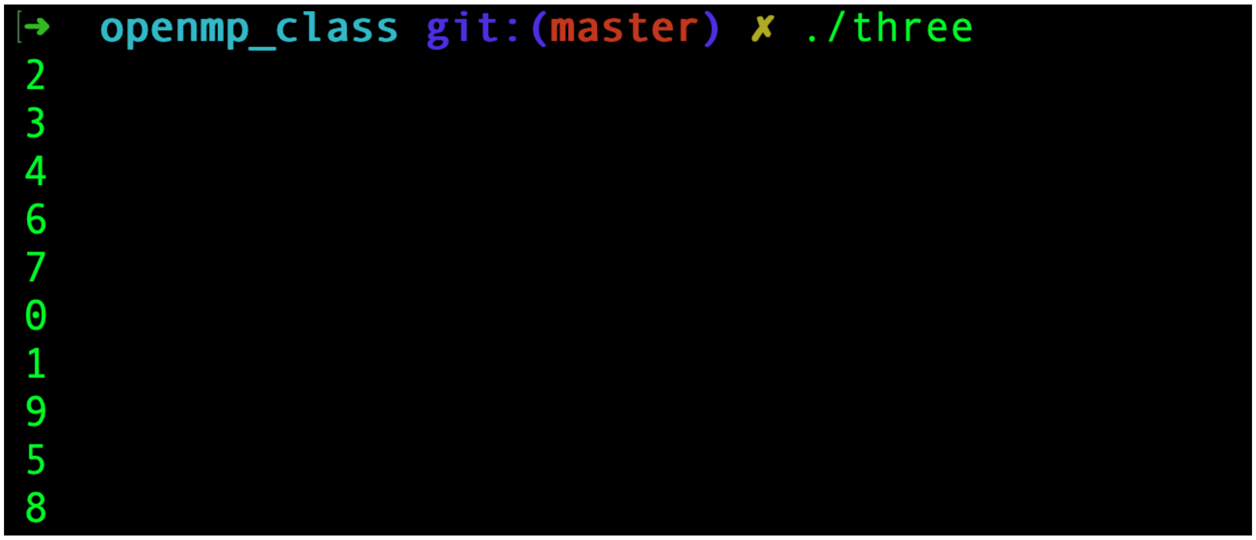
```
[➜  openmp_class git:(master) ✗ gcc-omp 1.c -o test
[➜  openmp_class git:(master) ✗ gcc-omp 1.c -o one
[➜  openmp_class git:(master) ✗ ./one
Hello from thread 6
Hello from thread 5
Hello from thread 2
Hello from thread 0
Hello from thread 7
Hello from thread 1
Hello from thread 4
Hello from thread 3
➜  openmp_class git:(master) ✗
```

=> Here we have used pragma omp parallel which makes the number of threads== no of cores of my local machine. We have user omp_get_thread_num() to get the thread id.



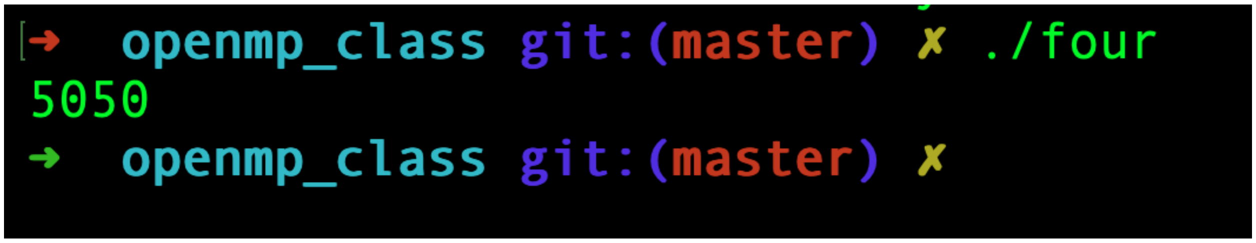
```
➜  openmp_class git:(master) ✗ ./two
0
2
3
1
➜  openmp_class git:(master) ✗ 
```

⇒ Here, the core logic remains the same as 1.c but here we have used **omp_set_num_threads(4)** to tell the compiler to only create 4 threads.



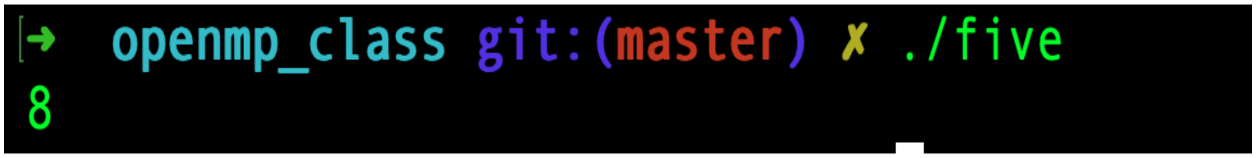
```
➜  openmp_class git:(master) ✗ ./three
2
3
4
6
7
0
1
9
5
8
```

⇒ Here, we have used **pragma omp parallel** for this is used to create thread and run the task for each iteration of the for loop. So, each threads divide the task for each iteration into different threads available.



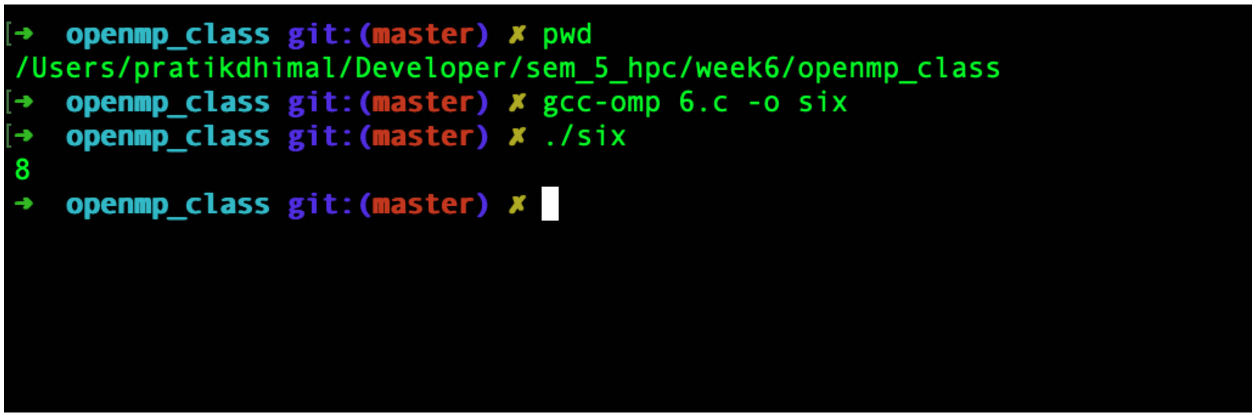
```
➜  openmp_class git:(master) ✗ ./four
5050
➜  openmp_class git:(master) ✗
```

=> Here, we have used reduction(+:s) in the omp parallel for. So, the each iteration f loop is divided into  number of threads and each thread add ups into the local copy and at last the reduction(+:s) reduces the overall value and adds to the global variable s



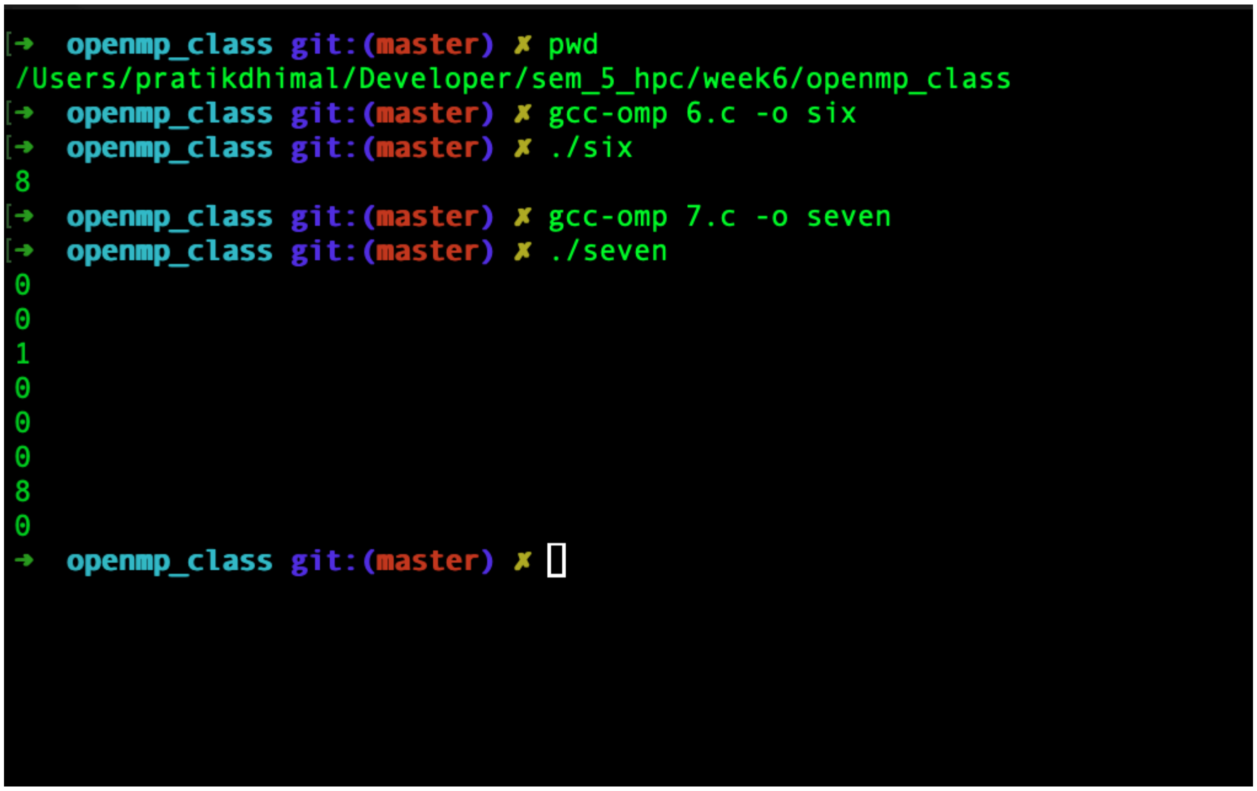
```
➜  openmp_class git:(master) ✗ ./five
8
```

Here we have used omp parallel to run threads here 8 as per my cpu cores. And we have used pragma omp critical to prevent race condition while doing x++.



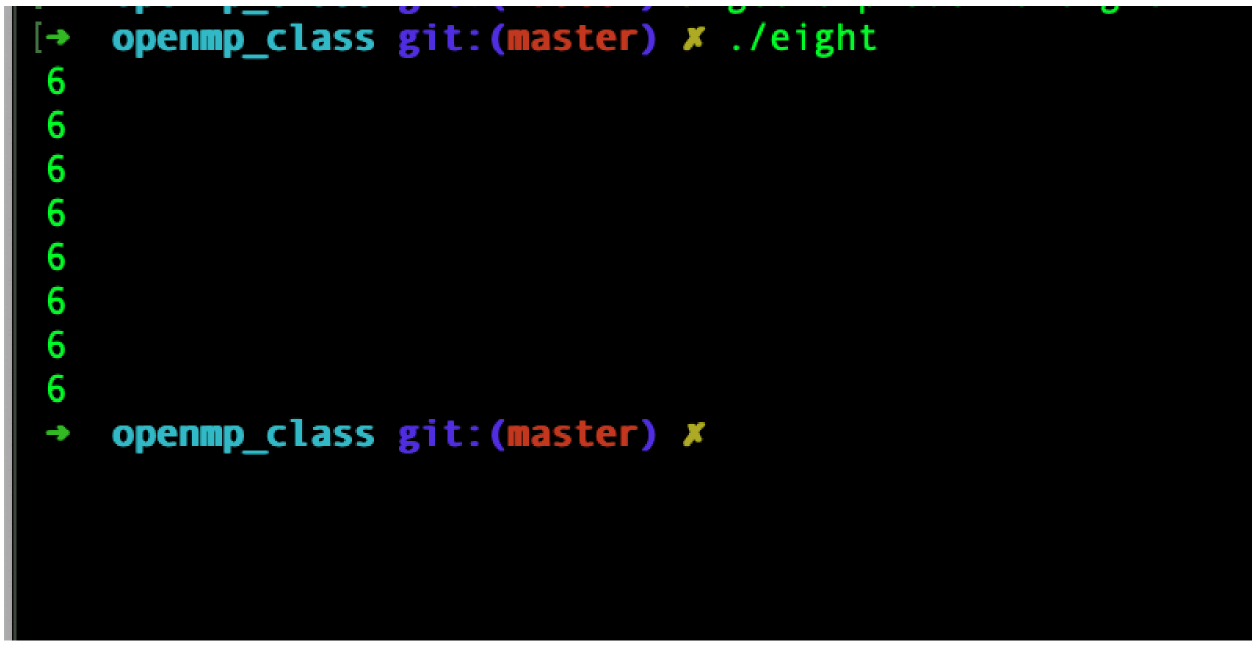
```
[➜  openmp_class git:(master) ✗ pwd
/Users/pratikdhimal/Developer/sem_5_hpc/week6/openmp_class
[➜  openmp_class git:(master) ✗ gcc-omp 6.c -o six
[➜  openmp_class git:(master) ✗ ./six
8
➜  openmp_class git:(master) ✗ 
```

With this program, there are several threads and they run x++. The #pragma omp atomic ensures that the increment is done safely to ensure that threads do not conflict during the updating of x. At last, x will be equal to the number of threads.



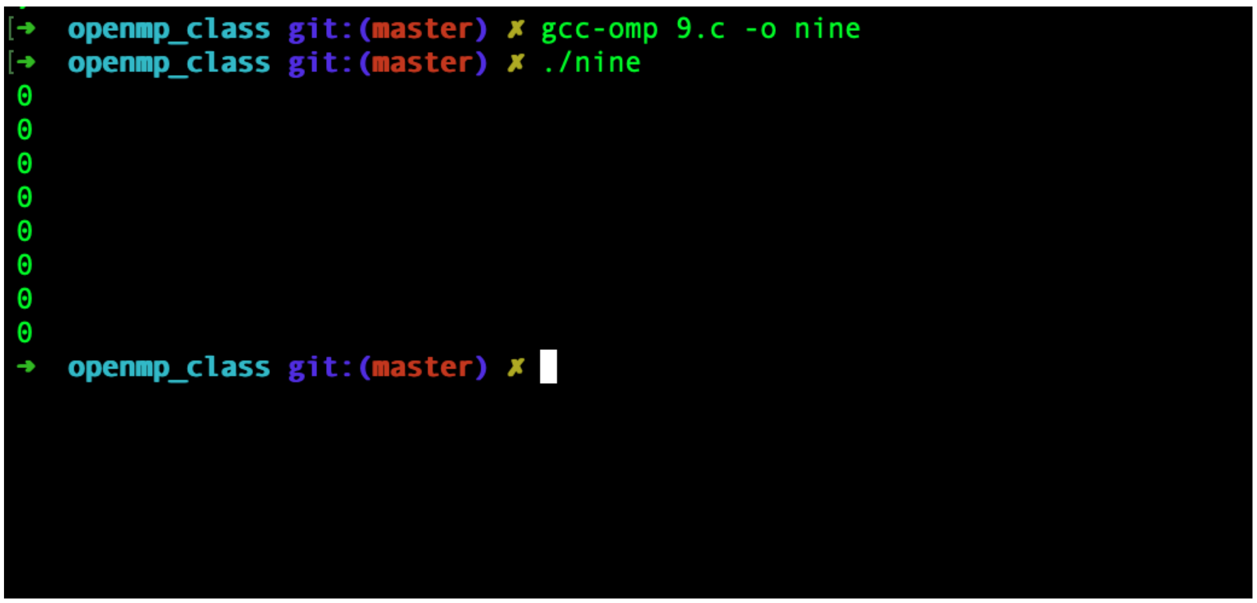
```
[➜  openmp_class git:(master) ✗ pwd
/Users/pratikdhimal/Developer/sem_5_hpc/week6/openmp_class
[➜  openmp_class git:(master) ✗ gcc-omp 6.c -o six
[➜  openmp_class git:(master) ✗ ./six
8
[➜  openmp_class git:(master) ✗ gcc-omp 7.c -o seven
[➜  openmp_class git:(master) ✗ ./seven
0
0
1
0
0
0
8
0
➜  openmp_class git:(master) ✗ ▯
```

In this program, the threads have their own uninitialized copy of a, and therefore, they do not retain the value 5. All threads write some meaningless values generated by randomisation.



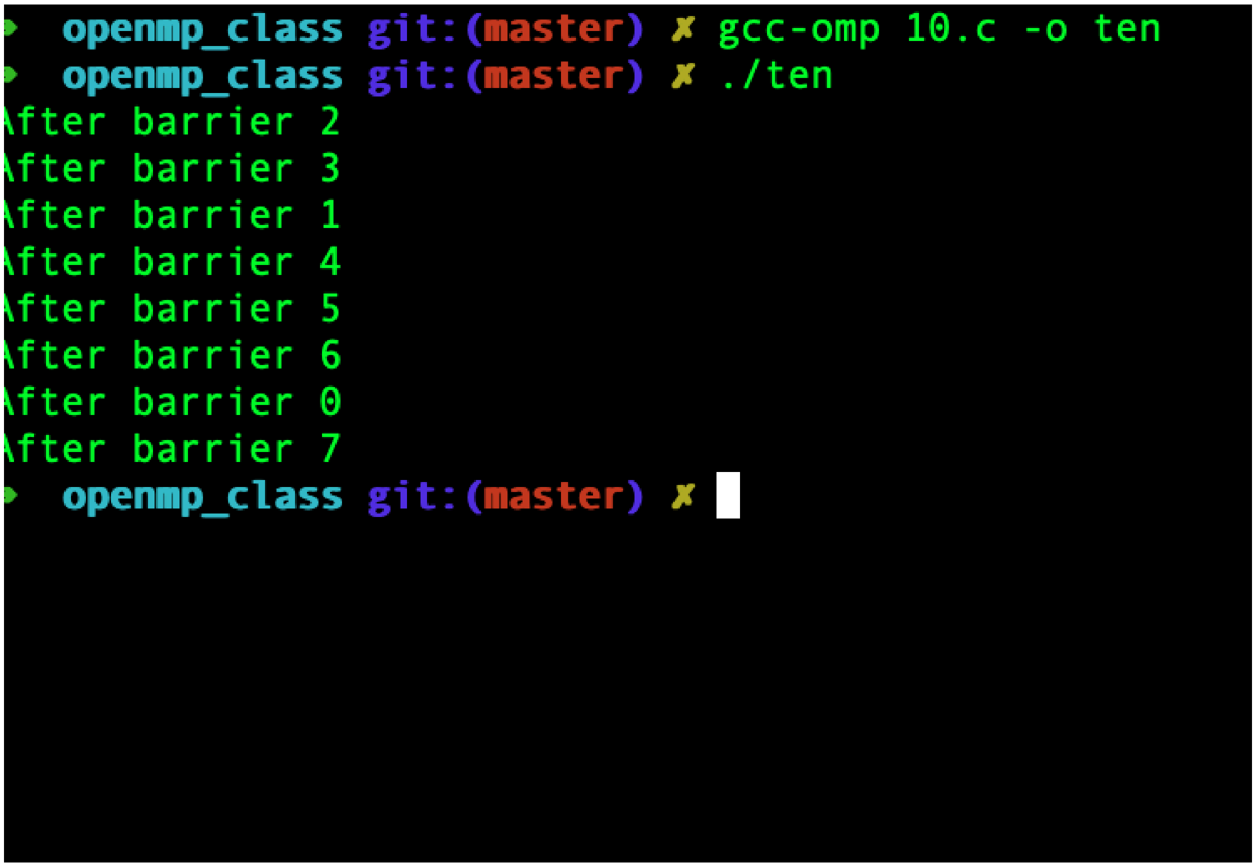
```
[→  openmp_class git:(master) ✗ ./eight
 6
 6
 6
 6
 6
 6
 6
 6
 →  openmp_class git:(master) ✗
```

Every thread is presented with its own distinct copy of a, by this program gives firstprivate(a) which is initialized at the initial value (5). So every thread safe copies its own copy and prints a+1= 6.



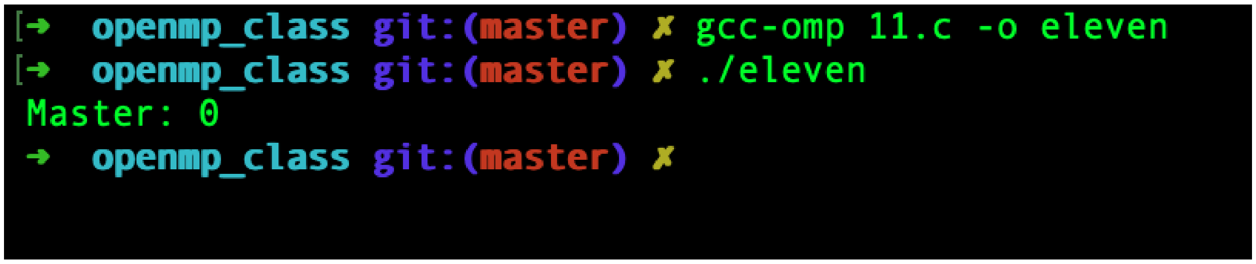
```
[➜  openmp_class git:(master) ✗ gcc-omp 9.c -o nine
[➜  openmp_class git:(master) ✗ ./nine
0
0
0
0
0
0
0
0
➜  openmp_class git:(master) ✗ 
```

In this program, all threads are given the same variable x, and the value of this is zero to all of them. Nothing can modify x, which means that each thread will print 0, yet the prints may be in any random order since they are running concurrently.



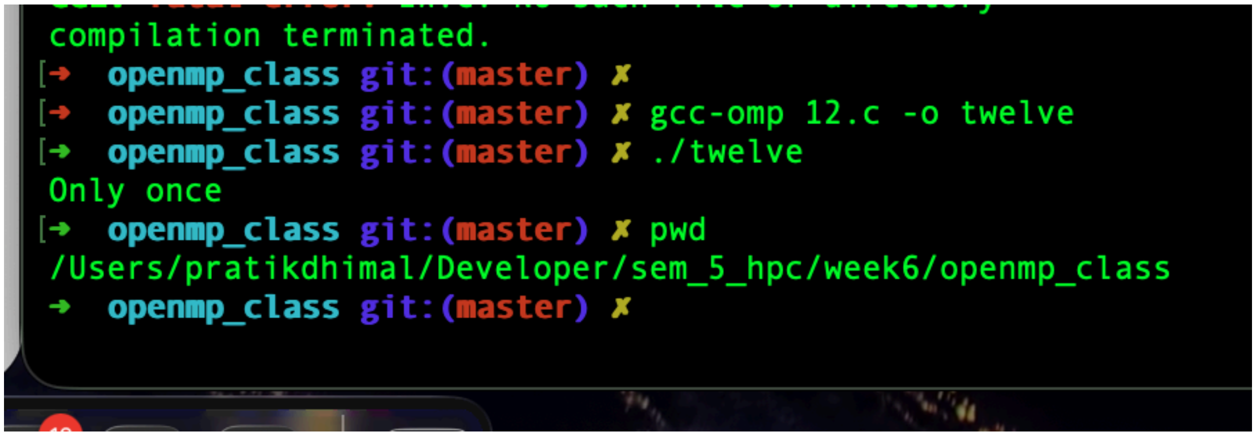
```
➜  openmp_class git:(master) ✗ gcc-omp 10.c -o ten
➜  openmp_class git:(master) ✗ ./ten
After barrier 2
After barrier 3
After barrier 1
After barrier 4
After barrier 5
After barrier 6
After barrier 0
After barrier 7
➜  openmp_class git:(master) ✗
```

This program barrier blocks until they have all
reached that point. Once all
threads come, they proceed singly, and every thread
writes After barrier and its.
thread ID.



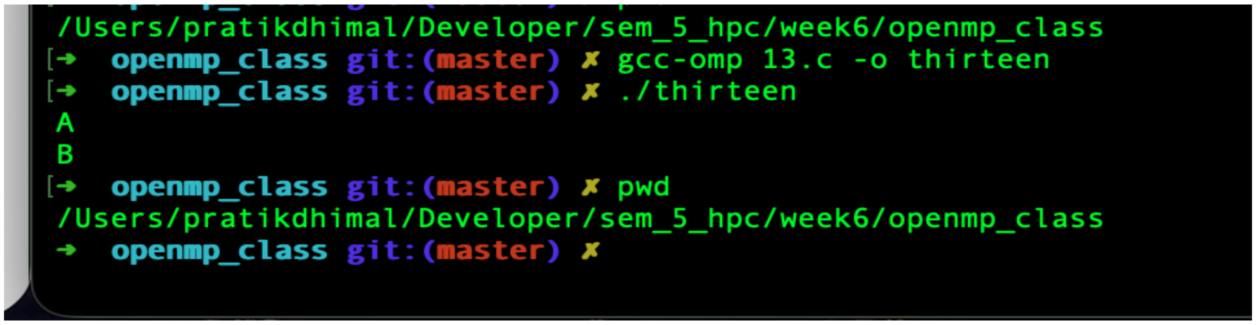
```
[➜  openmp_class git:(master) ✗ gcc-omp 11.c -o eleven
[➜  openmp_class git:(master) ✗ ./eleven
Master: 0
 ➜  openmp_class git:(master) ✗
```

This program master makes sure only the master thread (thread 0) runs that code block. So only thread 0 prints “Master: 0”, while all other threads skip it.



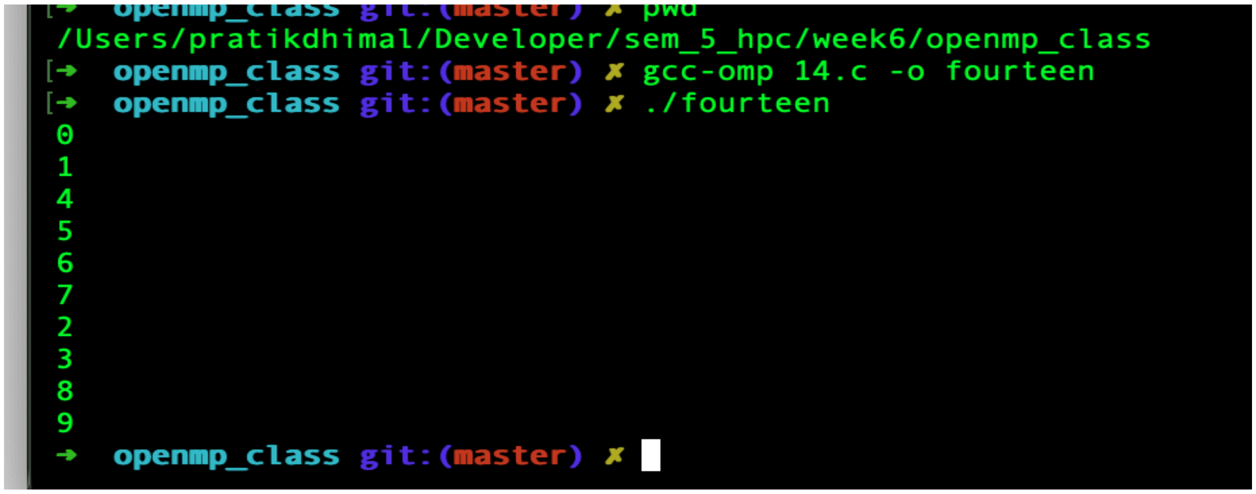
```
compilation terminated.
[➜  openmp_class git:(master) ✗
[➜  openmp_class git:(master) ✗ gcc-omp 12.c -o twelve
[➜  openmp_class git:(master) ✗ ./twelve
Only once
[➜  openmp_class git:(master) ✗ pwd
/Users/pratikdhimal/Developer/sem_5_hpc/week6/openmp_class
 ➜  openmp_class git:(master) ✗
```

This program’s single means only one random thread (not necessarily thread 0) runs that block, while all other threads skip it. So “Only once” is printed exactly one time.



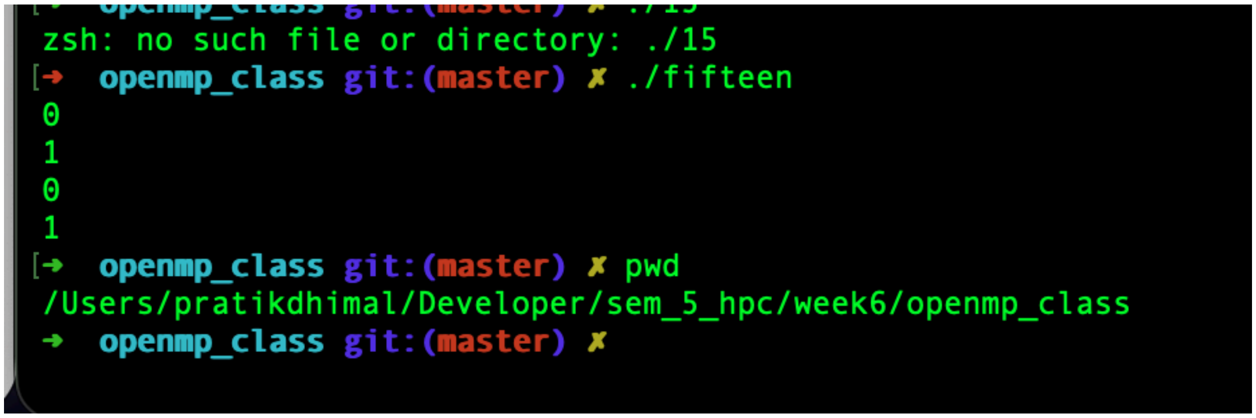
```
/Users/pratikdhimal/Developer/sem_5_hpc/week6/openmp_class
[➜  openmp_class git:(master) ✗ gcc-omp 13.c -o thirteen
[➜  openmp_class git:(master) ✗ ./thirteen
A
B
[➜  openmp_class git:(master) ✗ pwd
/Users/pratikdhimal/Developer/sem_5_hpc/week6/openmp_class
➜  openmp_class git:(master) ✗
```

This program creates multiple sections where each section runs on a different thread. One
thread prints “A” and another prints “B”. Both happen in parallel, and the order depends on
which thread runs first.



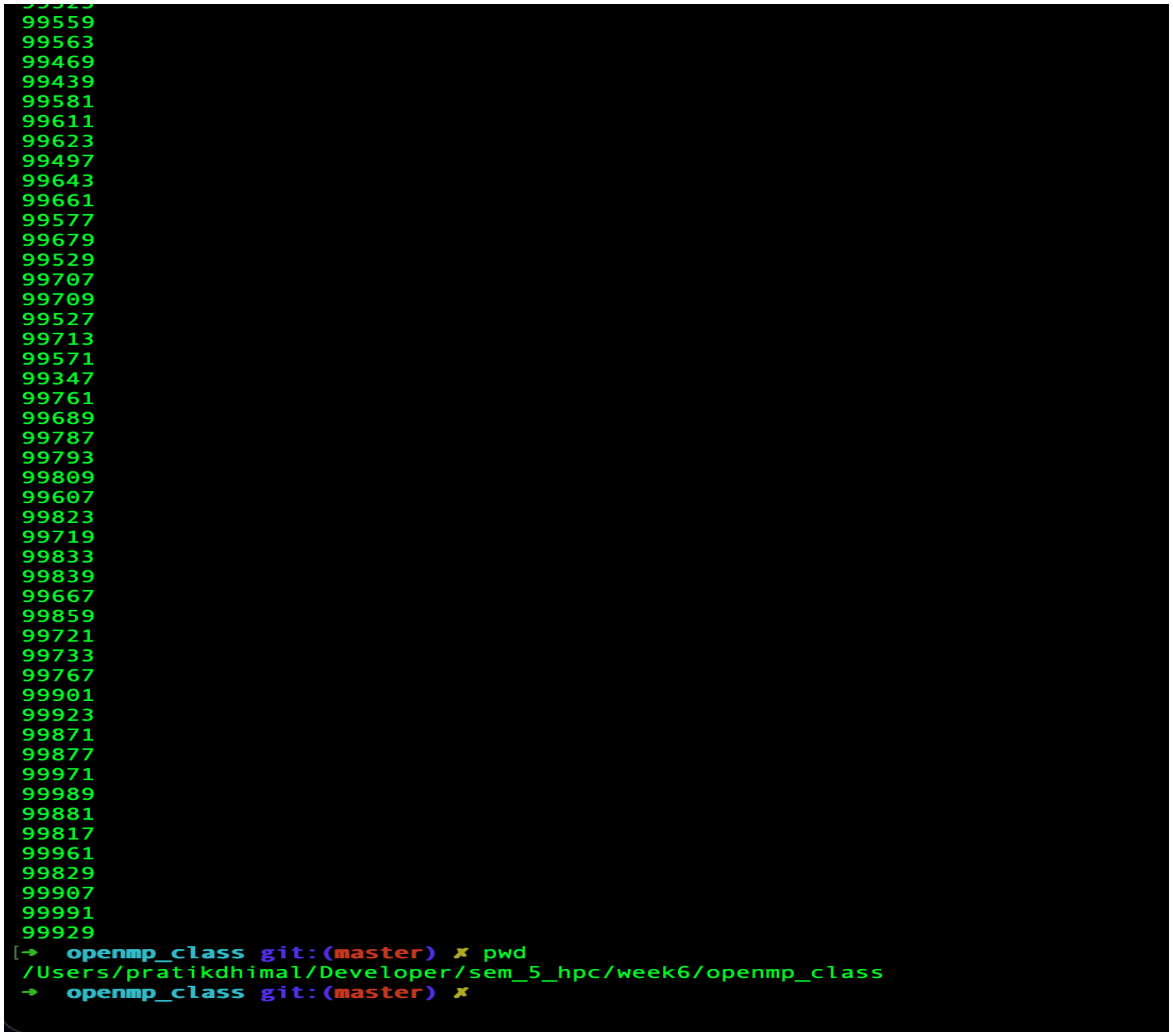
```
[➜  openmp_class git:(master) ✗ pwd
/Users/pratikdhimal/Developer/sem_5_hpc/week6/openmp_class
[➜  openmp_class git:(master) ✗ gcc-omp 14.c -o fourteen
[➜  openmp_class git:(master) ✗ ./fourteen
0
1
4
5
6
7
2
3
8
9
➜  openmp_class git:(master) ✗
```

This program schedule(dynamic,2) gives each thread 2 iterations at a time, and when a thread finishes its chunk, it grabs the next available chunk. So the loop runs in parallel, but the order of printed numbers is mixed because threads take work dynamically.



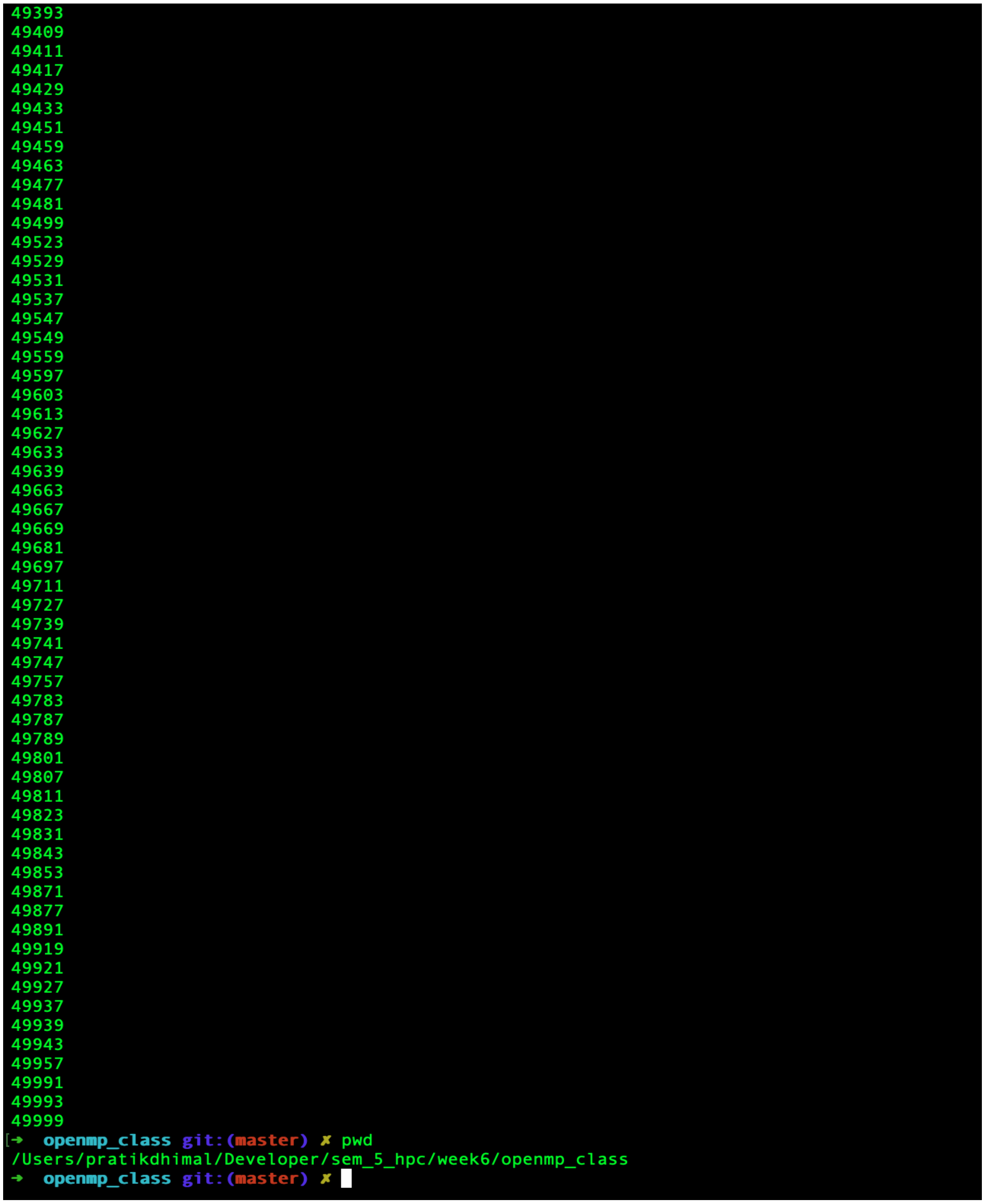
```
[→  openmp_class git:(master) ✗ ./15
zsh: no such file or directory: ./15
[→  openmp_class git:(master) ✗ ./fifteen
0
1
0
1
[→  openmp_class git:(master) ✗ pwd
/Users/pratikdhimal/Developer/sem_5_hpc/week6/openmp_class
→  openmp_class git:(master) ✗
```

In this program nested parallelism is turned on, so each of the 2 outer threads creates its own inner team of 2 threads. That means you get multiple inner threads printing their thread IDs, and the output looks messy because both levels run in parallel.



```
99559
99563
99469
99439
99581
99611
99623
99497
99643
99661
99577
99679
99529
99707
99709
99527
99713
99571
99347
99761
99689
99787
99793
99809
99607
99823
99719
99833
99839
99667
99859
99721
99733
99767
99901
99923
99871
99877
99971
99989
99881
99817
99961
99829
99907
99991
99929
[→  openmp_class git:(master) ✗ pwd
/Users/pratikdhimal/Developer/sem_5_hpc/week6/openmp_class
 →  openmp_class git:(master) ✗
```

This software is executed with OpenMP to speed up the work with concurrent threads and find all prime numbers between 1 and 100,000. The isprime function, by trying to divide a number by all other numbers up to its square root, identifies a particular number as a prime number or not. The line in main, which has the pragma omp parallel for schedule(dynamic) command, tells OpenMP to split the loop into multiple threads dynamically so that each thread will take the next available integer as its check. It prints a prime number. Essentially, it uses a large number of CPU cores to check primes faster, however, since threads are independent of each other, the reported numbers do not necessarily have a consecutive order.



```
49393
49409
49411
49417
49429
49433
49451
49459
49463
49477
49481
49499
49523
49529
49531
49537
49547
49549
49559
49597
49603
49613
49627
49633
49639
49663
49667
49669
49681
49697
49711
49727
49739
49741
49747
49757
49783
49787
49789
49801
49807
49811
49823
49831
49843
49853
49871
49877
49891
49919
49921
49927
49937
49939
49943
49957
49991
49993
49999
[➜  openmp_class git:(master) ✗ pwd
/Users/pratikdhimal/Developer/sem_5_hpc/week6/openmp_class
➜  openmp_class git:(master) ✗ 
```

In this program, the work is subdivided with the help of OpenMP parts instead of a parallel loop, however, all the prime numbers between 1 and 100,000 are still shown. Here, p1() and p2() are in the range of 1 to 50, 000 and 50,001 to 100,000, respectively. The #pragma omp parallel sections generates multiple threads, and each of the functions is implemented by the #pragma omp section in parallel. Through this, two threads are running simultaneously on different half ranges. It does this faster, although since each threads print separately the digits printed might appear in an odd order.