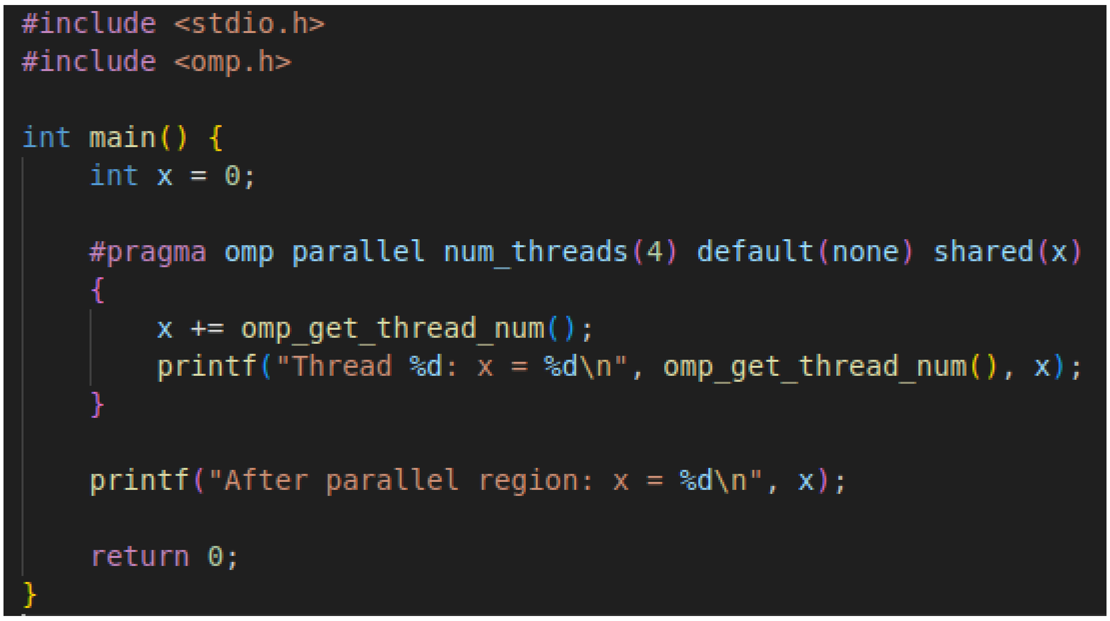
Name: Nitesh Sharma
Reg No: 23361
MTCS-103(P)
Parallel Processing Practicals
ASSignment 10

===

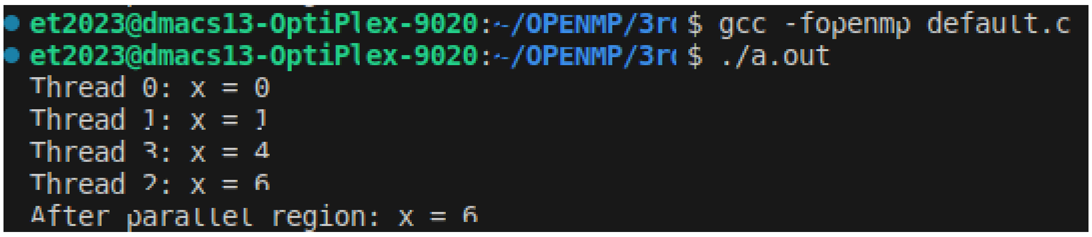
Data-Sharing Attributes:
Manage variable sharing in parallel sections. Set variables as shared, private, or firstprivate, enhancing parallel execution and resource use.

default(shared | none):
Controls default data-sharing for variables in parallel or task constructs. shared shares unspecified variables among threads, while none ensures private variables unique to each thread.



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

int main() {
    int x = 0;

    #pragma omp parallel num_threads(4) default(none) shared(x)
    {
        x += omp_get_thread_num();
        printf("Thread %d: x = %d\n", omp_get_thread_num(), x);
    }

    printf("After parallel region: x = %d\n", x);

    return 0;
}
```

Output:



```
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/3r $ gcc -fopenmp default.c
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/3r $ ./a.out
  Thread 0: x = 0
  Thread 1: x = 1
  Thread 3: x = 4
  Thread 2: x = 6
  After parallel region: x = 6
```

Explanation:
In this code, the default(none) clause is used in the parallel construct to ensure that all variables must be explicitly specified for data-sharing attributes. The shared(x) clause

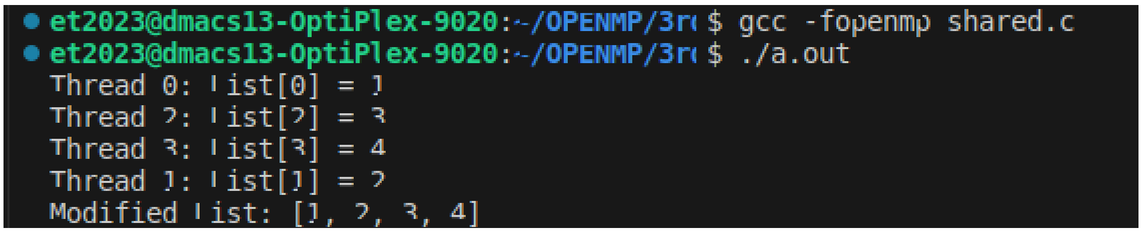
indicates that the variable x is shared among all threads. Each thread increments x by its thread number, and the final value of x is printed outside the parallel region.

shared(list):

The shared(list) clause in OpenMP is used to declare specific variables as shared among tasks generated by a parallel or task construct. This clause ensures that the listed variables are accessible and modifiable by all the tasks in the construct. By explicitly marking certain variables as shared, programmers can control the data-sharing behavior of the tasks, ensuring that the listed variables are consistent and synchronized across the tasks' executions. This helps manage data dependencies and access to shared resources, enhancing the reliability and correctness of parallel code.

```c
#include <stdio.h>
#include <omp.h>
int main() {
  int list[4] = {0, 0, 0, 0};
  #pragma omp parallel shared(list)
  {
     int thread_id = omp_get_thread_num();
     list[thread_id] = thread_id + 1;

     #pragma omp critical
     printf("Thread %d: List[%d] = %d\n", thread_id, thread_id, list[thread_id]);
  }
   printf("Modified List: [%d, %d, %d, %d]\n", list[0], list[1], list[2], list[3]);
  return 0;
}
```

Output:



```
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/3r $ gcc -fopenmp shared.c
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/3r $ ./a.out
  Thread 0: list[0] = 1
  Thread 2: list[2] = 3
  Thread 3: list[3] = 4
  Thread 1: list[1] = 2
  Modified list: [1, 2, 3, 4]
```

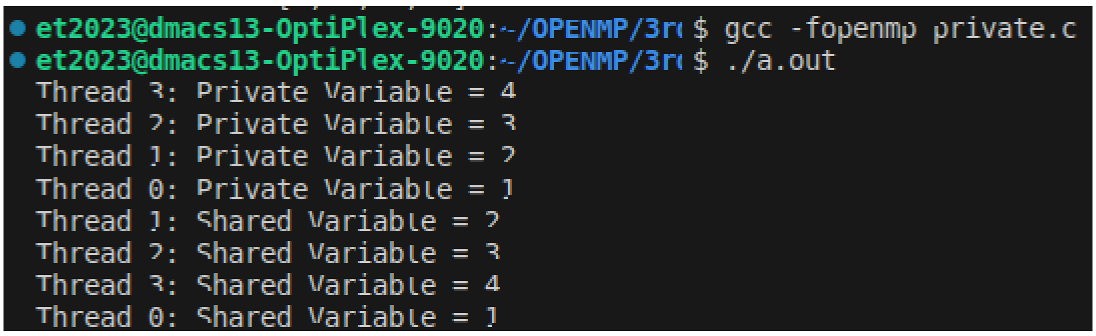
Explanation:

In the above code, the shared(list) clause is used within a parallel region to declare a shared list named list among the parallel tasks. Each task, represented by a thread, modifies its respective element in the shared list, incrementing its value by 1. The #pragma omp critical directive is used to ensure that only one thread at a time accesses and prints the modified values of the shared list. After the parallel region, the modified shared list values are printed, demonstrating the effect of the shared(list) clause.

private(list):

The private(list) clause is used to declare one or more variables in a parallel or task construct as private to each individual task. This means that each task will have its own private copy of the specified variables, allowing them to be manipulated independently without interfering with other tasks. This clause ensures that each task works with its own separate copy of the variables, preventing unintended data sharing and race conditions. It is particularly useful when tasks perform operations that require modifying variables in a parallel region without affecting other tasks' computations.

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

int main() {
  int shared_variable = 0;

  #pragma omp parallel private(shared_variable)
  {
     int thread_id = omp_get_thread_num();
     int private_variable = thread_id + 1;
     printf("Thread %d: Private Variable = %d\n", thread_id, private_variable);

     #pragma omp barrier

     #pragma omp atomic
     shared_variable += private_variable;

     #pragma omp barrier

     printf("Thread %d: Shared Variable = %d\n", thread_id, shared_variable);
  }

  return 0;
}
```
Output:



```
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/3r $ gcc -fopenmp private.c
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/3r $ ./a.out
  Thread 3: Private Variable = 4
  Thread 2: Private Variable = 3
  Thread 1: Private Variable = 2
  Thread 0: Private Variable = 1
  Thread 1: Shared Variable = 2
  Thread 2: Shared Variable = 3
  Thread 3: Shared Variable = 4
  Thread 0: Shared Variable = 1
```

Explanation:
In the above code, within the parallel region, each thread has its own private variable private_variable that is initialized with the thread's ID plus one. This ensures that each thread works with its individual private copy of the variable, preventing conflicts. After initializing the private variables, a barrier synchronization is used to ensure all threads have completed their computations before proceeding. Then, an atomic operation is employed to update the shared variable shared_variable by adding the private variable's value, ensuring safe concurrent updates without data races. Another barrier synchronization follows to ensure that all updates are completed before printing the final values of the shared variable.