Name: Nitesh Sharma
Reg No: 23361
MTCS-103(P)
Parallel Processing Practicals
ASSignment 8

===

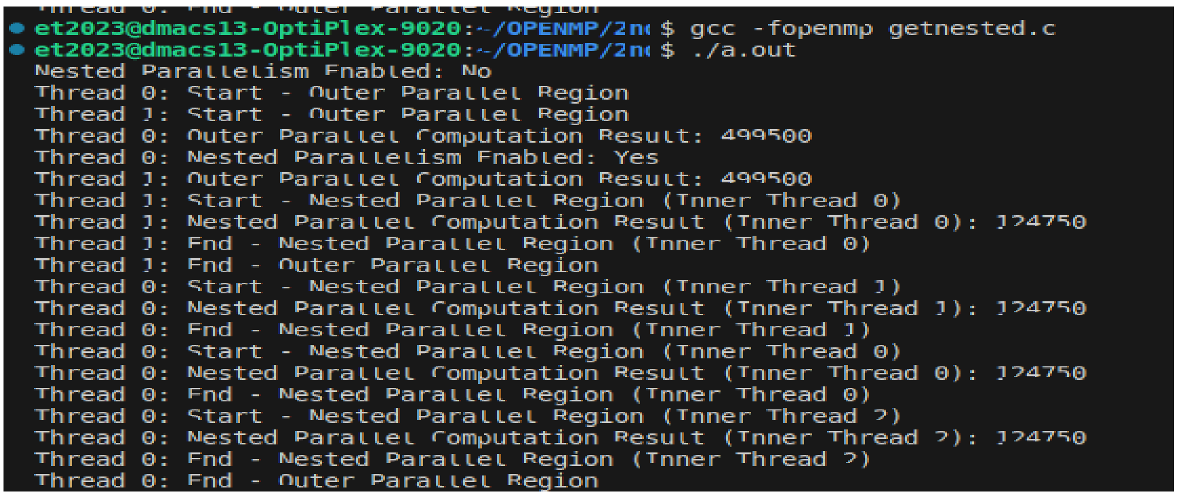
int omp_get_nested(void):

The function omp_get_nested() in OpenMP returns the value of the nested parallelism indicator. This indicator, represented by the nest-var internal control variable (ICV), determines whether nested parallelism is enabled or disabled within the program. Nested parallelism refers to the ability to create parallel regions within the context of an existing parallel region. When nested parallelism is enabled (value of nest-var is true), a thread within an active parallel region can create additional parallel regions, allowing further parallelism. If nested parallelism is disabled (value of nest-var is false), then parallel regions cannot be nested inside other parallel regions. This setting has implications for the behavior of nested parallel constructs, impacting the hierarchical structure and concurrency of parallel code execution.

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

int main() {
   int nested_enabled = omp_get_nested();
   printf("Nested Parallelism Enabled: %s\n", nested_enabled ? "Yes" :
"No");

   #pragma omp parallel num_threads(2)
   {
       int thread_id = omp_get_thread_num();
       printf("Thread %d: Start - Outer Parallel Region\n", thread_id);

       int outer_result = 0;
       for (int i = 0; i < 1000; ++i) {
           outer_result += i;
       }
       printf("Thread %d: Outer Parallel Computation Result: %d\n",
thread_id, outer_result);

       #pragma omp master
       {
           omp_set_nested(1);
           nested_enabled = omp_get_nested();
```

```c
            printf("Thread %d: Nested Parallelism Enabled: %s\n",
thread_id, nested_enabled ? "Yes" : "No");
        }

        #pragma omp parallel num_threads(3)
        {
            int nested_thread_id = omp_get_thread_num();
            printf("Thread %d: Start - Nested Parallel Region (Inner
Thread %d)\n", thread_id, nested_thread_id);

            int nested_result = 0;
            for (int j = 0; j < 500; ++j) {
                nested_result += j;
            }
            printf("Thread %d: Nested Parallel Computation Result (Inner
Thread %d): %d\n", thread_id, nested_thread_id, nested_result);

            printf("Thread %d: End - Nested Parallel Region (Inner
Thread %d)\n", thread_id, nested_thread_id);
        }

        printf("Thread %d: End - Outer Parallel Region\n", thread_id);
    }

    return 0;
}
```

Output:



```
  Thread 0: End - Outer Parallel Region
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ gcc -fopenmp getnested.c
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ ./a.out
  Nested Parallelism Enabled: No
  Thread 0: Start - Outer Parallel Region
  Thread 1: Start - Outer Parallel Region
  Thread 0: Outer Parallel Computation Result: 499500
  Thread 0: Nested Parallelism Enabled: Yes
  Thread 1: Outer Parallel Computation Result: 499500
  Thread 1: Start - Nested Parallel Region (Inner Thread 0)
  Thread 1: Nested Parallel Computation Result (Inner Thread 0): 124750
  Thread 1: End - Nested Parallel Region (Inner Thread 0)
  Thread 1: End - Outer Parallel Region
  Thread 0: Start - Nested Parallel Region (Inner Thread 1)
  Thread 0: Nested Parallel Computation Result (Inner Thread 1): 124750
  Thread 0: End - Nested Parallel Region (Inner Thread 1)
  Thread 0: Start - Nested Parallel Region (Inner Thread 0)
  Thread 0: Nested Parallel Computation Result (Inner Thread 0): 124750
  Thread 0: End - Nested Parallel Region (Inner Thread 0)
  Thread 0: Start - Nested Parallel Region (Inner Thread 2)
  Thread 0: Nested Parallel Computation Result (Inner Thread 2): 124750
  Thread 0: End - Nested Parallel Region (Inner Thread 2)
  Thread 0: End - Outer Parallel Region
```

Explanation:

The code demonstrates nested parallelism using OpenMP. It first checks if nested parallelism is enabled and prints the result. Then, it enters an outer parallel region where each thread performs a computation. Inside this region, it enables nested parallelism using the master thread and checks if it's enabled. Then, it enters a nested parallel region within each thread of the outer parallel region. The computation is performed in both the outer and nested parallel regions. The output provides information about the start and end of each parallel region, the computation results, and whether nested parallelism is enabled or disabled at various stages.

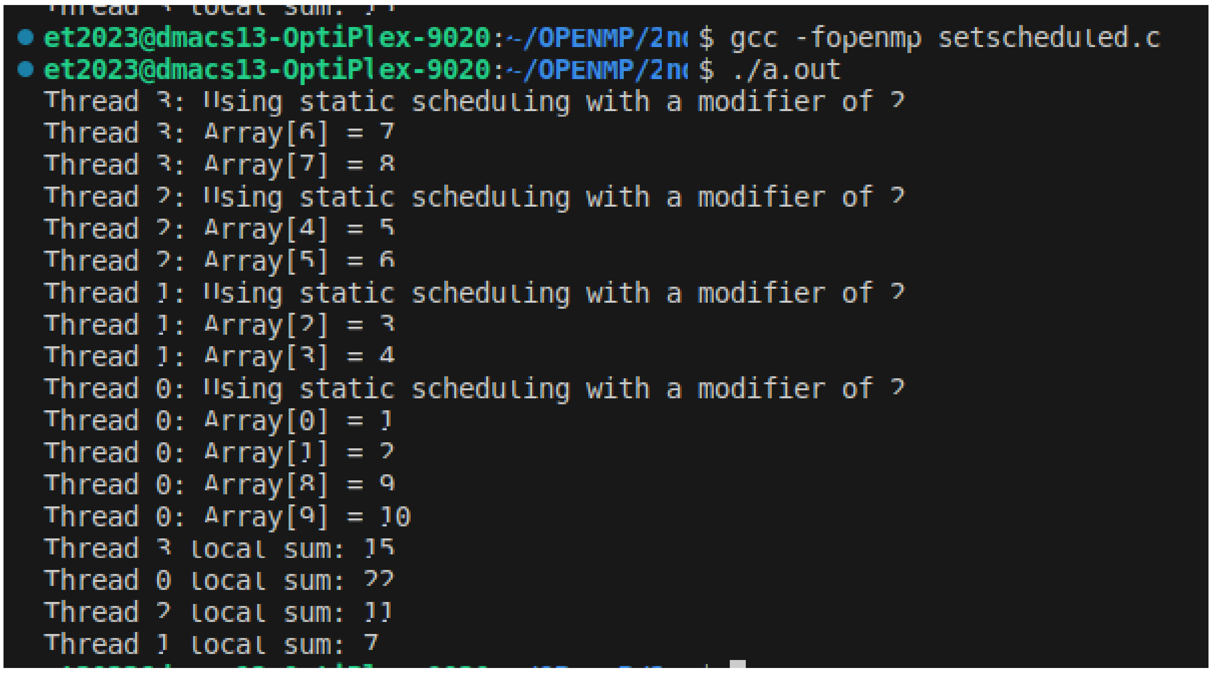
<u>void omp_set_schedule(omp_sched_t kind, int modifier);</u>

The function omp_set_schedule(kind, modifier) in OpenMP affects the scheduling plicy applied when the runtime schedule kind is used. It adjusts the value of the run-sched-var internal control variable (ICV), determining how loop iterations are distributed among threads. The kind parameter specifies the scheduling policy, such as static, dynamic, guided, auto, or an implementation-defined schedule. The modifier parameter provides additional configuration depending on the chosen scheduling policy. This function allows programmers to control the thread distribution and load balancing when executing loops in parallel, influencing the efficiency and performance of parallel code.

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

int main() {
   int num_threads = 4;
   int array_size = 10;
   int array[array_size];

   for (int i = 0; i < array_size; ++i) {
       array[i] = i + 1;
   }

   omp_set_num_threads(num_threads);

   #pragma omp parallel
   {
       int thread_id = omp_get_thread_num();
       int local_sum = 0;

       omp_set_schedule(omp_sched_static, 2);

       printf("Thread %d: Using static scheduling with a modifier of
2\n", thread_id);

       #pragma omp for schedule(runtime)
```

```
        for (int i = 0; i < array_size; ++i) {
            local_sum += array[i];
            printf("Thread %d: Array[%d] = %d\n", thread_id, i,
array[i]);
        }

        printf("Thread %d local sum: %d\n", thread_id, local_sum);
    }

    return 0;
}
```

Output:



```
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ gcc -fopenmp setscheduled.c
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ ./a.out
  Thread 3: Using static scheduling with a modifier of 2
  Thread 3: Array[6] = 7
  Thread 3: Array[7] = 8
  Thread 2: Using static scheduling with a modifier of 2
  Thread 2: Array[4] = 5
  Thread 2: Array[5] = 6
  Thread 1: Using static scheduling with a modifier of 2
  Thread 1: Array[2] = 3
  Thread 1: Array[3] = 4
  Thread 0: Using static scheduling with a modifier of 2
  Thread 0: Array[0] = 1
  Thread 0: Array[1] = 2
  Thread 0: Array[8] = 9
  Thread 0: Array[9] = 10
  Thread 3 local sum: 15
  Thread 0 local sum: 22
  Thread 2 local sum: 11
  Thread 1 local sum: 7
```

Explanation:

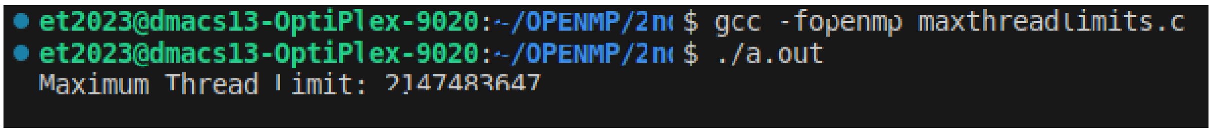
The provided code demonstrates the utilization of OpenMP's omp_set_schedule function to control scheduling of computations in a parallel program. After initializing an array of integers, the code sets the number of threads and applies static scheduling with a modifier of 2 using the omp_set_schedule function. Within the parallel region, each thread operates on different parts of the array using runtime scheduling. The output displays the thread ID, the selected scheduling kind, and the array elements being processed by each thread. Additionally, the local sums of each thread's computations are presented. This output effectively highlights the impact of omp_set_schedule on the distribution of work among threads, as well as the resulting local sum computations and array element processing within the parallel context.

int omp_get_thread_limit(void):

The function omp_get_thread_limit in OpenMP returns the value of the thread-limit-var internal control variable (ICV), which signifies the maximum number of OpenMP threads that can be utilized within a program. This value indicates the upper boundary for the number of threads that can be created in parallel regions. It is determined based on factors such as hardware capabilities, system resources, and configuration settings. By querying omp_get_thread_limit, a program can ascertain the maximum number of threads that it can effectively utilize for parallel computations, allowing for better resource management and optimization of parallel execution.

```c
#include <stdio.h>
#include <omp.h>
int main() {
   int max_thread_limit = omp_get_thread_limit();
   printf("Maximum Thread Limit: %d\n\n", max_thread_limit);
    return 0;
}
```

Output:



```
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ gcc -fopenmp maxthreadlimits.c
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ ./a.out
  Maximum Thread Limit: 2147483647
```

Explanation:
The provided code utilizes the omp_get_thread_limit function in OpenMP to retrieve and display the maximum thread limit that the system can support. In this specific output, the maximum thread limit is shown as "2147483647," which indicates that the system supports a very high number of threads. This value represents the upper bound on the number of threads that can be used effectively in parallel regions within the program.

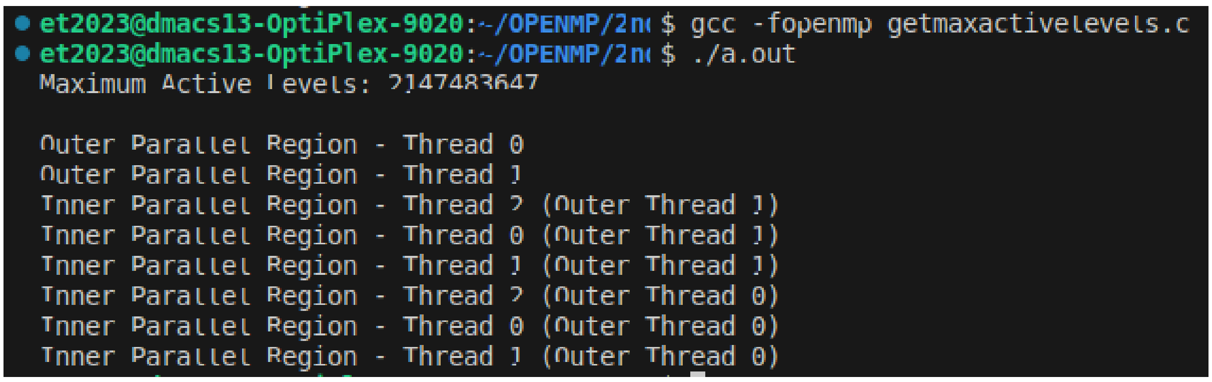
<u>int omp_get_max_active_levels(void);</u>

The function omp_get_max_active_levels in OpenMP returns the value of the max-active-levels-var internal control variable (ICV), which represents the maximum number of nested active parallel regions that can be created within the program. This value provides a limit on the depth of nested parallelism that can be utilized. By querying this function, a program can determine the maximum level of parallel nesting supported by the system. This information is valuable for understanding the level of nested parallelism that can be leveraged for improved performance in parallel programs.

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

int main() {
   int max_active_levels = omp_get_max_active_levels();
   printf("Maximum Active Levels: %d\n\n", max_active_levels);

   omp_set_max_active_levels(2);

```

```
    omp_set_nested(1);

    #pragma omp parallel num_threads(2)
    {
        int outer_thread_id = omp_get_thread_num();
        printf("Outer Parallel Region - Thread %d\n", outer_thread_id);

        #pragma omp parallel num_threads(3)
        {
            int inner_thread_id = omp_get_thread_num();
            printf("Inner Parallel Region - Thread %d (Outer Thread
%d)\n", inner_thread_id, outer_thread_id);
        }
    }

    return 0;
}
```

Output:



```
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ gcc -fopenmp getmaxactivelevels.c
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ ./a.out
  Maximum Active Levels: 2147483647

  Outer Parallel Region - Thread 0
  Outer Parallel Region - Thread 1
  Inner Parallel Region - Thread 2 (Outer Thread 1)
  Inner Parallel Region - Thread 0 (Outer Thread 1)
  Inner Parallel Region - Thread 1 (Outer Thread 1)
  Inner Parallel Region - Thread 2 (Outer Thread 0)
  Inner Parallel Region - Thread 0 (Outer Thread 0)
  Inner Parallel Region - Thread 1 (Outer Thread 0)
```

Explanation:

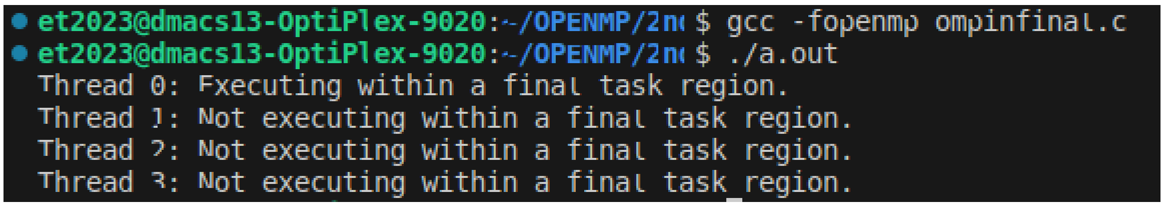
The above begins by utilizing the omp_get_max_active_levels() function to obtain the maximum allowable active levels of parallelism. Subsequently, omp_set_max_active_levels(2) is employed to establish a maximum of two active levels. The code then employs omp_set_nested(1) to enable nested parallelism explicitly. Within a main parallel region executed by two threads, each thread enters an outer parallel region. Inside this outer region, each thread generates an inner parallel region with three threads. The output provides insights into the execution sequence: two threads concurrently execute the outer parallel region, and within each outer thread, a three-threaded inner parallel region is executed. The thread IDs are printed to emphasize both outer and inner thread participation, highlighting parallelism at different levels.

int omp_in_final(void);

The function ‘omp_in_final’ is used to determine whether the current execution context is within a final task region or an included task region in OpenMP. It returns true if the code is being executed within such a region, indicating that the task is in its final state or if the task has been included explicitly, and false if the code is not within any task region. This function is useful to conditionally adjust behaviour or perform certain actions based on whether the code is in a final or included task region, allowing for dynamic adaptation of the code's execution depending on the context of the task region.

```
#include <stdio.h>
#include <omp.h>
int main() {
   int num_threads = 4;
   omp_set_num_threads(num_threads);
   #pragma omp parallel
   {
       int thread_id = omp_get_thread_num();
       int in_final_task = 0;
       #pragma omp task final(thread_id == 0)
       {
           in_final_task = omp_in_final();
            if (in_final_task) {
           printf("Thread %d: Executing within a final task region.\n",
thread_id);
       } else {
           printf("Thread %d: Not executing within a final task
region.\n", thread_id);
       }
       }
   }

   return 0;
}
```

Output:



```
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ gcc -fopenmp ompinfinal.c
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ ./a.out
  Thread 0: Executing within a final task region.
  Thread 1: Not executing within a final task region.
  Thread 2: Not executing within a final task region.
  Thread 3: Not executing within a final task region.
```

Explanation:

The above code creates a parallel region where four threads are utilized. Within this region, each thread evaluates whether it is currently executing within a final task region using the omp_in_final() function. The code also features a task construct with a final clause that specifies that the task should only be executed by thread 0. As a result, only thread 0 enters the task region, and the program checks whether this thread is executing in a final task or not. The output illustrates that thread 0 is indeed executing within a final task region, while the other threads are not.