Name: Nitesh Sharma
Reg No: 23361
MTCS-103(P)
Parallel Processing Practicals
ASSignment 7

==

Execution Environment Routines

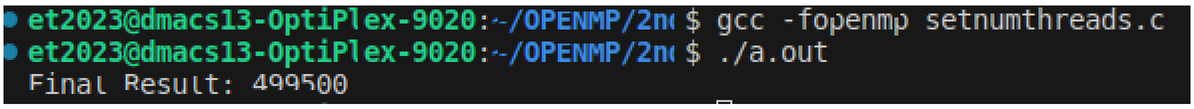
Execution environment routines in OpenMP provide mechanisms to control and monitor the behavior of threads, processors, and the parallel execution environment. These routines allow programmers to influence various aspects of parallel execution, such as setting the number of threads to be used in subsequent parallel regions, checking whether a call is within an active parallel region, and obtaining the maximum thread limit for the program.

void omp_set_num_threads(int num_threads);

Affects the number of threads used for subsequent parallel regions that do not specify a num_threads clause.

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

int main() {
   omp_set_num_threads(3);

   int result = 0;

   #pragma omp parallel for reduction(+:result)
   for (int i = 1; i < 1000; ++i) {
       result += i;
   }

   printf("Final Result: %d\n", result);

   return 0;
}
```

Output:



```
et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ gcc -fopenmp setnumthreads.c
et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ ./a.out
Final Result: 499500
```

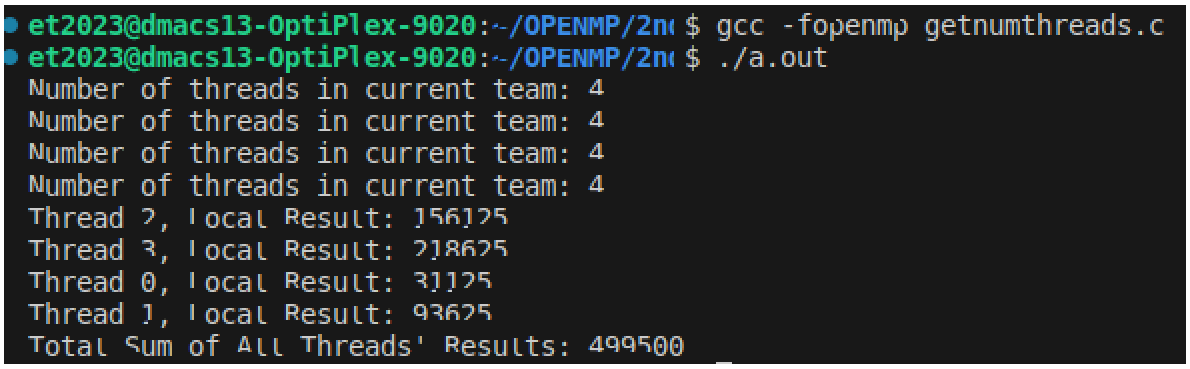
Explanation:
In this code, the omp_set_num_threads function is used to set the number of threads to 3 for subsequent parallel regions. The parallel region calculates the sum of integers from 1 to 999 using a reduction operation. Since the number of threads is set to 3, the computation is performed in parallel using three threads. The final result is then printed.

int omp_get_num_threads(void);
The function omp_get_num_threads() is an essential part of the OpenMP library that provides valuable insight into the current parallel execution environment. When called within an OpenMP parallel region, this function returns the count of threads actively participating in the execution of that specific parallel region, essentially indicating the size of the "team" of threads. This information is invaluable for understanding the level of parallelism being utilized and can be used to tailor computations or resource allocation accordingly. By enabling program logic to adapt to the number of available threads, developers can optimize workload distribution and achieve efficient parallel processing.

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

int main() {
   int num_threads;
   int total_sum = 0;

   #pragma omp parallel
   {
       num_threads = omp_get_num_threads();
       printf("Number of threads in current team: %d\n", num_threads);

       int local_result = 0;
       #pragma omp for
       for (int i = 0; i < 1000; ++i) {
           local_result += i;
       }

       #pragma omp critical
```

```
        {
            total_sum += local_result;
        }

        printf("Thread %d, Local Result: %d\n", omp_get_thread_num(),
local_result);
    }

    printf("Total Sum of All Threads' Results: %d\n", total_sum);

    return 0;
}
```

Output:



```
et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ gcc -fopenmp getnumthreads.c
et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ ./a.out
Number of threads in current team: 4
Number of threads in current team: 4
Number of threads in current team: 4
Number of threads in current team: 4
Thread 2, Local Result: 156125
Thread 3, Local Result: 218625
Thread 0, Local Result: 31125
Thread 1, Local Result: 93625
Total Sum of All Threads' Results: 499500
```

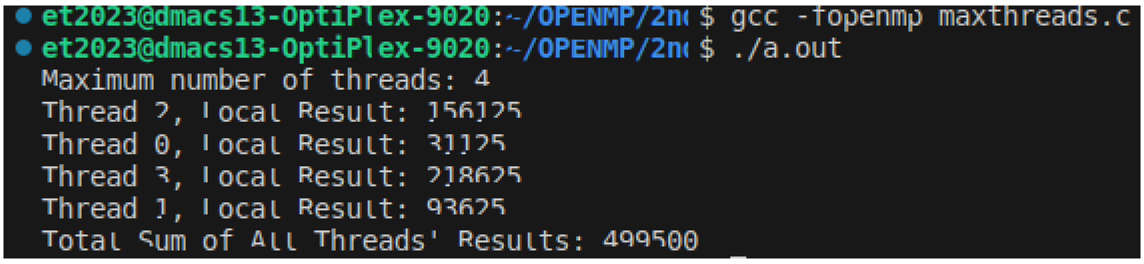
Explanation:
This code demonstrates the usage of the omp_get_num_threads function within an OpenMP parallel region. The program starts by initializing variables for the total sum of results across all threads and the number of threads in the current team. Within the parallel region, each thread retrieves the number of threads in the team using omp_get_num_threads and prints this value. Then, a local sum is calculated for each thread using a loop that iterates from 0 to 999. Each thread accumulates its local result within the local_result variable. To avoid data race conditions, a critical section is used to update the total_sum variable with each thread's local result. Finally, the program prints the local result of each thread along with its thread ID, and the total sum of all threads' results. This code showcases how omp_get_num_threads provides insight into the number of threads actively participating in the parallel computation, allowing effective monitoring and utilization of the parallel environment.

int omp_get_max_threads(void):

The omp_get_max_threads() function is a part of the OpenMP library that provides valuable information about the execution environment's thread capabilities. It returns the maximum number of threads that can be utilized to form a new team in a parallel construct without explicitly specifying the number of threads using the num_threads clause. This function is useful for understanding the upper limit of the available thread resources in a given environment. By obtaining this value, programmers can make informed decisions about the degree of parallelism they should aim for in their parallelized computations. The function helps ensure efficient utilization of available resources while avoiding oversubscription, where the number of threads exceeds the available physical cores, potentially leading to performance degradation due to increased context switching and overhead.

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

int main() {
    int max_threads = omp_get_max_threads();
    printf("Maximum number of threads: %d\n", max_threads);

    int total_sum = 0;

    #pragma omp parallel num_threads(max_threads)
    {
        int local_result = 0;
        #pragma omp for
        for (int i = 0; i < 1000; ++i) {
            local_result += i;
        }

        #pragma omp critical
        {
            total_sum += local_result;
            printf("Thread %d, Local Result: %d\n",
omp_get_thread_num(), local_result);
        }
    }

    printf("Total Sum of All Threads' Results: %d\n", total_sum);

    return 0;
}
```

Output:



```
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ gcc -fopenmp maxthreads.c
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ ./a.out
  Maximum number of threads: 4
  Thread 2, Local Result: 156125
  Thread 0, Local Result: 31125
  Thread 3, Local Result: 218625
  Thread 1, Local Result: 93625
  Total Sum of All Threads' Results: 499500
```

Explanation:

The above code demonstrates the use of the omp_get_max_threads() function in OpenMP to obtain the maximum number of threads that can be employed in a parallel construct without specifying the number of threads explicitly. The code first retrieves the maximum thread count and prints it out. It then initializes a total_sum variable to accumulate the final result of the parallel computation. Inside the parallel region, the program calculates local results for each thread by iterating through a loop. A critical section ensures proper aggregation of the local results into the total_sum variable while printing out each thread's local result. This approach guarantees efficient utilization of available resources by utilizing the maximum number of threads that the environment can support for parallel processing, while also ensuring proper synchronization to avoid race conditions during result accumulation.

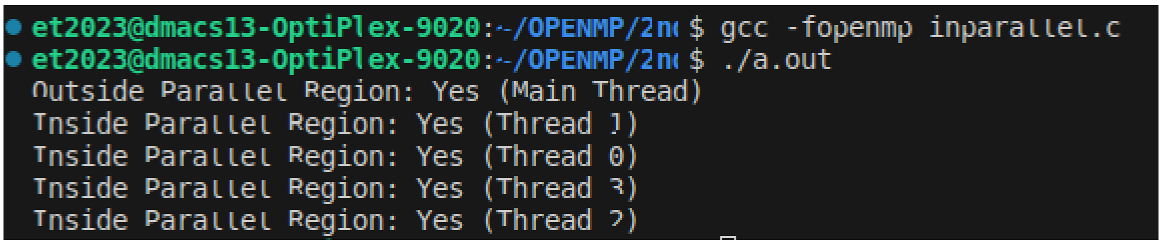
int omp_in_parallel(void):

The omp_in_parallel() function is designed to provide information about the execution context of a program with respect to parallel regions. When called, it evaluates whether the current code segment is enclosed within an active parallel region or not. If the routine detects that the code is indeed within an active parallel region, it returns true, indicating that multiple threads are executing concurrently. On the other hand, if the code is not enclosed within a parallel region, the function returns false, signifying that the execution is occurring in a single-threaded context. This mechanism allows programmers to understand the parallelism of their code and make informed decisions based on the context of parallelism in their program execution.

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

int main() {
   int in_parallel;
     in_parallel = omp_in_parallel();
   if (in_parallel) {
       printf("Outside Parallel Region: No (Main Thread)\n");
   } else {
       printf("Outside Parallel Region: Yes (Main Thread)\n");
   }

   #pragma omp parallel
   {
       in_parallel = omp_in_parallel();
       if (in_parallel) {
           printf("Inside Parallel Region: Yes (Thread %d)\n",
omp_get_thread_num());
```

```
        } else {
            printf("Inside Parallel Region: No (Thread %d)\n",
omp_get_thread_num());
        }
    }

    return 0;
}
```

Output:



```
et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ gcc -fopenmp inparallel.c
et2023@dmacs13-OptiPlex-9020:~/OPENMP/2n $ ./a.out
 Outside Parallel Region: Yes (Main Thread)
 Inside Parallel Region: Yes (Thread 1)
 Inside Parallel Region: Yes (Thread 0)
 Inside Parallel Region: Yes (Thread 3)
 Inside Parallel Region: Yes (Thread 2)
```

Explanation of the code:

The Above code demonstrates the usage of the omp_in_parallel() function to determine the presence of an active parallel region during program execution. Initially, the code checks if it is executed outside an active parallel region, and it correctly identifies itself as being outside. Upon entering the parallel region created by #pragma omp parallel, each thread queries the same function to ascertain whether it is executing within an active parallel context. As the output shows, all threads within the parallel region confirm that they are indeed within an active parallel context, accurately reflecting their participation in parallel execution. The code effectively showcases the utility of omp_in_parallel() in determining the parallel execution context and verifying whether code sections are running concurrently in a multi-threaded environment.

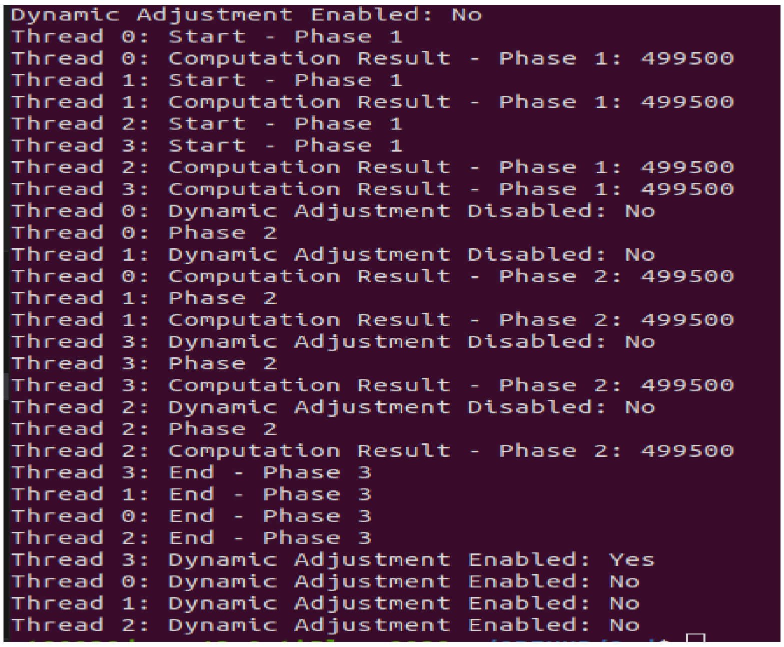
int omp_get_dynamic(void):

The omp_get_dynamic() function in OpenMP returns the value of the dynamic adjustment ICV (Internal Control Variable), which indicates whether dynamic adjustment of the number of threads is enabled or disabled. When dynamic adjustment is enabled, the OpenMP runtime system has the flexibility to adjust the number of threads for a parallel region dynamically based on the available resources and the workload. This means that the actual number of threads can change from one parallel region to another. On the other hand, when dynamic adjustment is disabled, the number of threads remains constant and is determined by the initial settings or the OMP_NUM_THREADS environment variable. The ability to enable or disable dynamic thread adjustment allows programmers to have more control over the number of threads used in parallel regions, optimizing performance for specific workloads and resource constraints.

```
#include <stdio.h>

#include <omp.h>
```

```c
int main() {
    int dynamic_enabled = omp_get_dynamic();
    printf("Dynamic Adjustment Enabled: %s\n", dynamic_enabled ? "Yes" : "No");
    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        printf("Thread %d: Start - Phase 1\n", thread_id);
              int result = 0;
        for (int i = 0; i < 1000; ++i) {
            result += i;
        }
        printf("Thread %d: Computation Result - Phase 1: %d\n", thread_id, result);
        #pragma omp barrier
        #pragma omp single
        omp_set_dynamic(0);
        dynamic_enabled = omp_get_dynamic();
        printf("Thread %d: Dynamic Adjustment Disabled: %s\n", thread_id, dynamic_enabled ? "Yes" : "No");
        printf("Thread %d: Phase 2\n", thread_id);
        result = 0;
        for (int i = 0; i < 1000; ++i) {
            result += i;
        }
        printf("Thread %d: Computation Result - Phase 2: %d\n", thread_id, result);
        #pragma omp barrier

        printf("Thread %d: End - Phase 3\n", thread_id);

        #pragma omp single

        omp_set_dynamic(1);

        dynamic_enabled = omp_get_dynamic();

        printf("Thread %d: Dynamic Adjustment Enabled: %s\n", thread_id,
dynamic_enabled ? "Yes" : "No");

    }

    return 0;

}
```

Output:



```
Dynamic Adjustment Enabled: No
Thread 0: Start - Phase 1
Thread 0: Computation Result - Phase 1: 499500
Thread 1: Start - Phase 1
Thread 1: Computation Result - Phase 1: 499500
Thread 2: Start - Phase 1
Thread 3: Start - Phase 1
Thread 2: Computation Result - Phase 1: 499500
Thread 3: Computation Result - Phase 1: 499500
Thread 0: Dynamic Adjustment Disabled: No
Thread 0: Phase 2
Thread 1: Dynamic Adjustment Disabled: No
Thread 0: Computation Result - Phase 2: 499500
Thread 1: Phase 2
Thread 1: Computation Result - Phase 2: 499500
Thread 3: Dynamic Adjustment Disabled: No
Thread 3: Phase 2
Thread 3: Computation Result - Phase 2: 499500
Thread 2: Dynamic Adjustment Disabled: No
Thread 2: Phase 2
Thread 2: Computation Result - Phase 2: 499500
Thread 3: End - Phase 3
Thread 1: End - Phase 3
Thread 0: End - Phase 3
Thread 2: End - Phase 3
Thread 3: Dynamic Adjustment Enabled: Yes
Thread 0: Dynamic Adjustment Enabled: No
Thread 1: Dynamic Adjustment Enabled: No
Thread 2: Dynamic Adjustment Enabled: No
```

Explanation:
The above code demonstrates the use of the omp_get_dynamic() function to determine whether dynamic adjustment of the number of threads is enabled or disabled. In the beginning, it checks if dynamic adjustment is enabled (usually it is by default). Then, within a parallel region, it performs computations in three phases. In Phase 1 and Phase 3, dynamic adjustment is enabled, while in Phase 2, it is explicitly disabled using omp_set_dynamic(0). The output displays whether dynamic adjustment is enabled or disabled for each thread during these phases. Additionally, the computed result for each thread in each phase is printed. In the output, it can be observed how dynamic adjustment affects the number of threads and their behaviour in different phases of the program.