Reg. No: 23361
MTCS-103(P)
Parallel Processing
Assignment 5

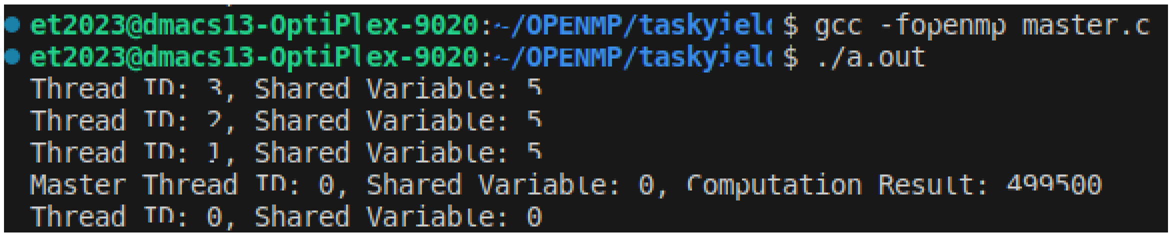
Master clause:

The master construct specifies a structured block that is executed by the master thread of the team. There is no implied barrier either on entry to, or exit from, the master construct.
#pragma omp master structured-block

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

int main() {
   int shared_var = 5;

   #pragma omp parallel
   {
       #pragma omp master
       {
           shared_var = omp_get_thread_num();
           int result = 0;
           for (int i = 0; i < 1000; ++i) {
               result += i;
           }
           printf("Master Thread ID: %d, Shared Variable: %d, Computation
Result: %d\n", omp_get_thread_num(), shared_var, result);
       }

       printf("Thread ID: %d, Shared Variable: %d\n",
omp_get_thread_num(), shared_var);
   }
```

```
    return 0;
}
```

Output:



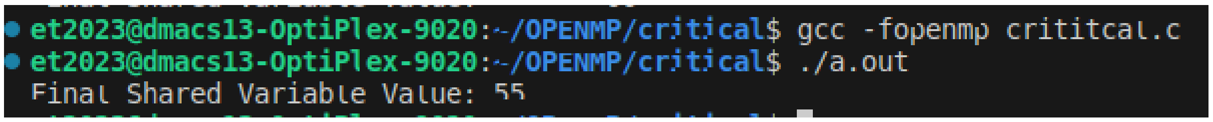
```
et2023@dmacs13-OptiPlex-9020:~/OPENMP/taskyiel$ gcc -fopenmp master.c
et2023@dmacs13-OptiPlex-9020:~/OPENMP/taskyiel$ ./a.out
Thread ID: 3, Shared Variable: 5
Thread ID: 2, Shared Variable: 5
Thread ID: 1, Shared Variable: 5
Master Thread ID: 0, Shared Variable: 0, Computation Result: 499500
Thread ID: 0, Shared Variable: 0
```

Explanation of the code:
The provided code demonstrates the use of the OpenMP master construct within a parallel region. The master construct designates a structured block of code that is executed exclusively by the master thread of the team. Inside the master block, the shared variable shared_var is assigned the thread ID of the master thread. The master thread then performs a computation to calculate the sum of integers from 0 to 999, displaying the computed result along with the master thread's ID and the value of shared_var. Outside the master block, all threads within the parallel region print their individual thread IDs and the value of the shared variable. The output reflects this behaviour, showing that the master thread's computation is separate from the parallel execution of other threads. The shared variable's value remains 5 for all threads except the master thread, which changes it to its thread ID (0). This showcases the distinct role of the master thread and the shared nature of the variable in a parallel context.

Critical clause:

```
#include <stdio.h>
#include <omp.h>
#define ARRAY_SIZE 5
int main() {
    int shared_var = 0;
    int array[ARRAY_SIZE];

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

    #pragma omp parallel
    {
        #pragma omp for
        for (int i = 0; i < ARRAY_SIZE; ++i) {
            int temp_result = array[i] * array[i];

            #pragma omp critical
            {
                                shared_var += temp_result;
            }
        }
    }

    printf("Final Shared Variable Value: %d\n", shared_var);
    return 0;
}
```

Output:



```
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/critical$ gcc -fopenmp crititcal.c
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/critical$ ./a.out
  Final Shared Variable Value: 55
```

Explanation of the Code:
In the code, the #pragma omp critical construct is used to create a critical section within the parallel loop. This critical section ensures that only one thread can access and modify the shared_var variable at a time. Inside the loop, each thread calculates the square of an element in the array and stores it temporarily in the temp_result variable. To avoid race conditions and ensure correct accumulation of the squared values, the critical section is employed to sequentially update the shared_var by adding the temp_result value. This prevents multiple threads from concurrently modifying the shared variable and ensures the accuracy of the final computation. As a result, the critical clause enables proper synchronization among threads, allowing them to perform their calculations safely and contributing to the accurate summation of the squared values, which is then displayed as the final shared variable value.

Takswait clause:

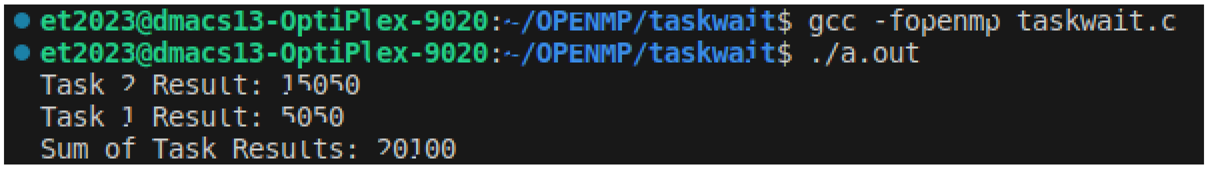
Taskwait

The taskwait construct specifies a wait on the completion of child tasks of the current task.

#pragma omp taskwait

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

int main() {
   int result1 = 0;
   int result2 = 0;

   #pragma omp parallel
   {
       #pragma omp single
       {
           #pragma omp task
           {
               for (int i = 1; i <= 100; ++i) {
                   result1 += i;
               }
               printf("Task 1 Result: %d\n", result1);
           }

           #pragma omp task
           {
               for (int i = 101; i <= 200; ++i) {
                   result2 += i;
               }
               printf("Task 2 Result: %d\n", result2);
           }

           #pragma omp taskwait

           printf("Sum of Task Results: %d\n", result1 + result2);
       }
   }

   return 0;
}
```

Output:



```
et2023@dmacs13-OptiPlex-9020:~/OPENMP/taskwait$ gcc -fopenmp taskwait.c
et2023@dmacs13-OptiPlex-9020:~/OPENMP/taskwait$ ./a.out
Task 2 Result: 15050
Task 1 Result: 5050
Sum of Task Results: 20100
```

Explanation:
In this code, two child tasks are created within the single construct, and each task calculates the sum of integers in a specific range. After the child tasks are spawned, the taskwait construct ensures that the execution of the main task (the code following taskwait) is delayed until both child tasks are completed. This synchronization mechanism guarantees that the calculation of the sum of task results occurs only after both child tasks have finished their computations. The output of the program provides the results of each individual task as well as the sum of their results, demonstrating how taskwait facilitates proper synchronization among parallel tasks.

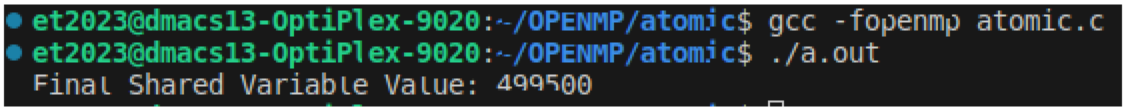
Atomic clause:

The atomic construct ensures that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.
#pragma omp atomic [read | write | update | capture]

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

int main() {
   int shared_var = 0;

   #pragma omp parallel
   {
       #pragma omp for
       for (int i = 0; i < 1000; ++i) {
           #pragma omp atomic update
           shared_var += i;
       }
   }

   printf("Final Shared Variable Value: %d\n", shared_var);
```

```
    return 0;
}
```

Outtput:



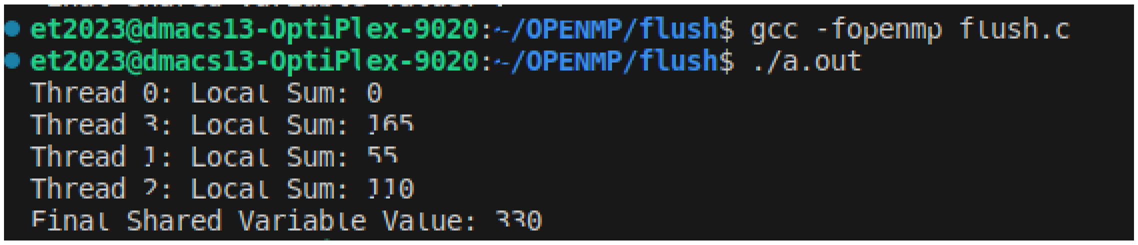
```
et2023@dmacs13-OptiPlex-9020:~/OPENMP/atomic$ gcc -fopenmp atomic.c
et2023@dmacs13-OptiPlex-9020:~/OPENMP/atomic$ ./a.out
Final Shared Variable Value: 499500
```

Explanation of the code:
In this code, multiple threads increment the shared_var variable within a loop. The #pragma omp atomic update construct ensures that the update to shared_var is performed atomically, preventing data races that can occur due to simultaneous updates from multiple threads. After all the loop iterations, the final value of the shared_var variable is printed, showcasing the correctness of atomic updates in a parallel context

<u>Flush clause:</u>

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

#define ARRAY_SIZE 10

int main() {
    int shared_var = 0;
    int array[ARRAY_SIZE];

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

    #pragma omp parallel
    {
        int thread_id = omp_get_thread_num();
        int local_sum = 0;
        for (int i = 0; i < ARRAY_SIZE; ++i) {
            local_sum += array[i] * thread_id;
        }
        #pragma omp critical
        {
            shared_var += local_sum;
            #pragma omp flush(shared_var)
        }

        printf("Thread %d: Local Sum: %d\n", thread_id, local_sum);
    }

    printf("Final Shared Variable Value: %d\n", shared_var);

    return 0;
}
```

Output:



```
et2023@dmacs13-OptiPlex-9020:~/OPENMP/flush$ gcc -fopenmp flush.c
et2023@dmacs13-OptiPlex-9020:~/OPENMP/flush$ ./a.out
Thread 0: Local Sum: 0
Thread 3: Local Sum: 165
Thread 1: Local Sum: 55
Thread 2: Local Sum: 110
Final Shared Variable Value: 330
```

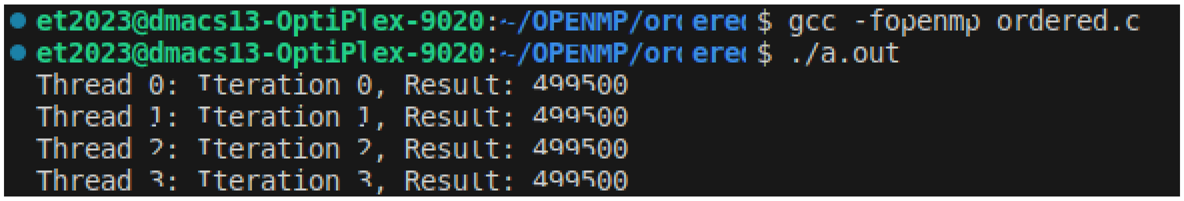
Explanation of the code:
The above code demonstrates the use of the #pragma omp flush construct within a parallel region involving multiple threads. Each thread performs its individual computation by multiplying array elements with its assigned thread ID. The local sums are then accumulated into the shared_var variable within a critical section to ensure mutual exclusion. The #pragma omp flush(shared_var) construct is utilized to enforce memory consistency, guaranteeing that the updated value of shared_var is visible to all threads. The output displays the local sums calculated by each thread and the final value of the shared variable, showcasing the coordination of memory synchronization to achieve accurate computation results.

Ordered clause:

The ordered construct specifies a structured block in a loop region that will be executed in the order of the loop iterations. This sequentializes and orders the code within an ordered region while allowing code outside the region to run in parallel.
#pragma omp ordered

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

int main() {
   #pragma omp parallel
   {
       #pragma omp for ordered
       for (int i = 0; i < 4; ++i) {
           int thread_id = omp_get_thread_num();
           int result = 0;
           for (int j = 0; j < 1000; ++j) {
               result += j;
           }
           #pragma omp ordered
           printf("Thread %d: Iteration %d, Result: %d\n", thread_id, i,
result);
       }
   }

   return 0;
}
```

Output:



```
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/ordered$ gcc -fopenmp ordered.c
● et2023@dmacs13-OptiPlex-9020:~/OPENMP/ordered$ ./a.out
  Thread 0: Iteration 0, Result: 499500
  Thread 1: Iteration 1, Result: 499500
  Thread 2: Iteration 2, Result: 499500
  Thread 3: Iteration 3, Result: 499500
```

Explanation of the above code:
The provided code demonstrates the use of the ordered construct in OpenMP. Inside a parallel region, a loop is marked with #pragma omp for ordered, ensuring that the loop iterations are executed in a predetermined order. In each iteration, a separate thread performs a computation by summing up integers from 0 to 999. Despite executing concurrently, the #pragma omp ordered directive ensures that the results are printed in the correct order, maintaining the original sequence of loop iterations.