Reg. No: 23361
MTCS-103(P)
Parallel Processing
Assignment 4

Task :

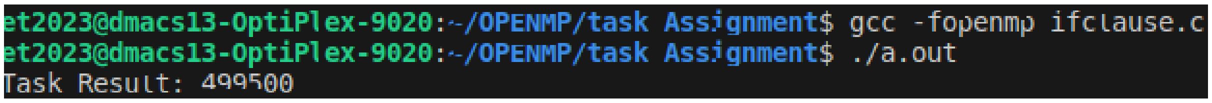
The task construct defines an explicit task. The data environment of the task is created according to the data-sharing attribute clauses on the task construct and any defaults that apply.

If Clause:

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

int main() {
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task if(omp_get_thread_num() == 0)
            {
                int result = 0;
                for (int i = 0; i < 1000; ++i) {
                    result += i;
                }
                printf("Task Result: %d\n", result);
            }
        }
    }
    return 0;
}
```

Output:



```
et2023@dmacs13-OptiPlex-9020:~/OPENMP/task Assignment$ gcc -fopenmp ifclause.c
et2023@dmacs13-OptiPlex-9020:~/OPENMP/task Assignment$ ./a.out
Task Result: 499500
```

Explanation of the above code:
Here a single task is defined using the task directive and a conditional clause if(omp_get_thread_num() == 0). This clause specifies that the task will be executed only by the thread with thread ID 0. The task's purpose is to calculate the sum of integers from 0 to 999999 and display the result. The loop within the task iterates over the range [0, 999999], adding each value to the result variable. Once the computation is complete, the result is printed out. This code effectively demonstrates the concept of a task within OpenMP, showcasing its potential for parallelizing independent computations and optimizing parallel execution.

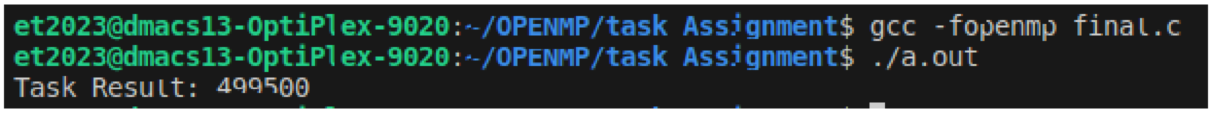
<u>Final Clause:</u>

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

int main() {
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task final(omp_get_thread_num() == 0)
            {
                int result = 0;
                for (int i = 0; i < 1000; ++i) {
                    result += i;
                }
                printf("Task Result: %d\n", result);
            }
        }
    }

    return 0;
}
```

Output:



```
et2023@dmacs13-OptiPlex-9020:~/OPENMP/task Assignment$ gcc -fopenmp final.c
et2023@dmacs13-OptiPlex-9020:~/OPENMP/task Assignment$ ./a.out
Task Result: 499500
```

Code Explanation:
The provided code employs OpenMP's task construct within a parallel region to create a task that calculates the sum of integers from 0 to 999999. Enclosed within a single region, the task is generated only by the master thread (thread ID 0) due to the final clause. Each thread within the parallel region remains idle while the master thread executes this task. After the computation, the result is displayed, showcasing the parallel task's role in efficiently calculating the sum across a large range of integers.

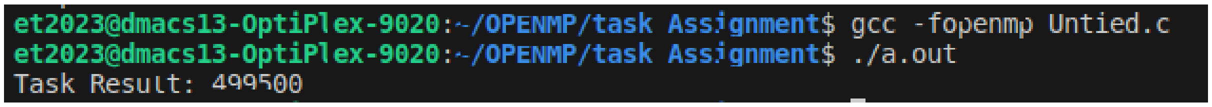
<u>Untied Clause:</u>

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

int main() {
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task untied
            {
                int result = 0;
                for (int i = 0; i < 1000; ++i) {
                    result += i;
                }
                printf("Task Result: %d\n", result);
            }
        }
    }

    return 0;
}
```

Output:



```
et2023@dmacs13-OptiPlex-9020:~/OPENMP/task Assignment$ gcc -fopenmp Untied.c
et2023@dmacs13-OptiPlex-9020:~/OPENMP/task Assignment$ ./a.out
Task Result: 499500
```

Code Explanation:
The given code employs OpenMP's task construct within a parallel region to facilitate parallel execution. The untied clause specifies that the task can be executed by any available thread, emphasizing the absence of a binding requirement between the generating thread and the executing thread. Consequently, multiple threads are eligible to execute the task. The task calculates the sum of integers from 0 to 999999, distributing the computation across threads. Each thread independently contributes to the calculation, resulting in efficient parallel processing. Upon completion, the calculated sum is printed, demonstrating how task untied enhances parallelism by allowing flexible thread assignment and optimal utilization of available resources.

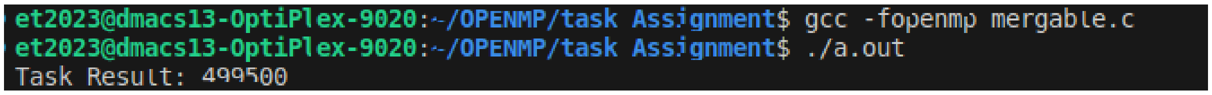
Mergeable Clause:

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

int main() {
    #pragma omp parallel
    {
        #pragma omp single
        {
            #pragma omp task mergeable
            {
                int result = 0;
                for (int i = 0; i < 1000; ++i) {
                    result += i;
                }
                printf("Task Result: %d\n", result);
            }
        }
    }

    return 0;
}
```

Output:



```
et2023@dmacs13-OptiPlex-9020:~/OPENMP/task Assignment$ gcc -fopenmp mergable.c
et2023@dmacs13-OptiPlex-9020:~/OPENMP/task Assignment$ ./a.out
Task Result: 499500
```

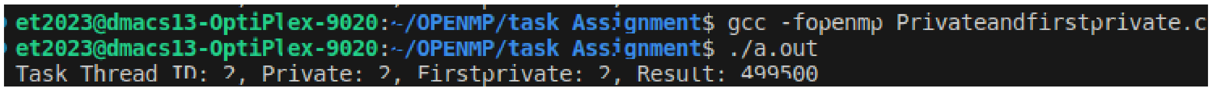
Code Explanation:

The provided code employs the OpenMP task construct within a parallel region to enable parallel processing. The mergeable clause is applied to the task, indicating that the runtime system can potentially merge this task with other similar tasks for better task scheduling efficiency. The code starts by creating a parallel region using #pragma omp parallel, where multiple threads collaborate. Within this region, a single thread is designated as the master using #pragma omp single. The subsequent #pragma omp task mergeable directive establishes a task that calculates the sum of integers from 0 to 999999. Each thread independently carries out this computation, and the mergeable attribute enhances the runtime's flexibility in merging similar tasks if beneficial. After the computation, the calculated result is printed. This design illustrates how task mergeable can optimize task scheduling in a parallel environment, enhancing efficiency in managing tasks with similar characteristics.

<u>Private and firstprivate clause:</u>

```c
#include <stdio.h>
#include <omp.h>
int main() {
   int private_var = 0;
   int firstprivate_var = 0;

   #pragma omp parallel
   {
       #pragma omp single
       {
           #pragma omp task private(private_var)
firstprivate(firstprivate_var)
           {
               private_var = omp_get_thread_num(); // Each task has its
own private_var
               firstprivate_var = omp_get_thread_num(); // Initialize
firstprivate_var with thread num
               int result = 0;
               for (int i = 0; i < 1000; ++i) {
                   result += i;
               }
               printf("Task Thread ID: %d, Private: %d, Firstprivate: %d,
Result: %d\n",
                      omp_get_thread_num(), private_var, firstprivate_var,
result);
           }
           #pragma omp taskwait
       }
   }
   return 0;
}
```
Output:



```
et2023@dmacs13-OptiPlex-9020:~/OPENMP/task Assignment$ gcc -fopenmp Privateandfirstprivate.c
et2023@dmacs13-OptiPlex-9020:~/OPENMP/task Assignment$ ./a.out
Task Thread ID: 2, Private: 2, Firstprivate: 2, Result: 499500
```

Code Explanation:
The provided code showcases the usage of the OpenMP task construct within a parallel region. In this code, the private clause ensures that each task has its own isolated copy of

the private_var variable. The firstprivate clause initializes each task's firstprivate_var with the value of the master thread's omp_get_thread_num(), which is the thread number. The task then computes the sum of integers from 0 to 999 using a loop. After the computation, the output displays the thread ID, the private and firstprivate variable values (both set to the thread number), and the calculated result. This example illustrates how the private and firstprivate clauses offer thread-specific private data and initial values, respectively, within parallel tasks, enhancing parallelism in OpenMP programs.

__Shared Clause:__

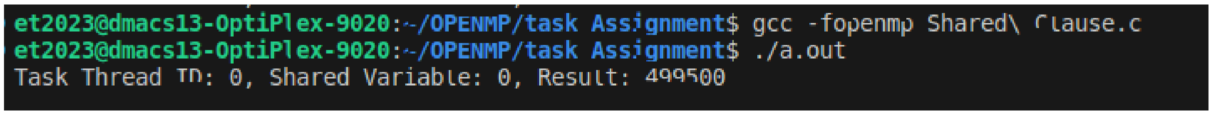
```
#include <stdio.h>
#include <omp.h>

int main() {
   int shared_var = 0;

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

   return 0;
}
```

Output:



```
et2023@dmacs13-OptiPlex-9020:~/OPENMP/task Assignment$ gcc -fopenmp Shared\ Clause.c
et2023@dmacs13-OptiPlex-9020:~/OPENMP/task Assignment$ ./a.out
Task Thread ID: 0, Shared Variable: 0, Result: 499500
```

Code Explaination:
The given code demonstrates the application of the OpenMP shared clause within a parallel region. In this code, the shared clause ensures that the shared_var variable is shared among all threads in the parallel region. The task construct within the single construct creates a task that computes the sum of integers from 0 to 999 using a loop. The shared variable shared_var is modified within the task to have a value of 0, and the computed result is also displayed. However, since the shared clause makes the variable shared among all threads, any modifications made by one thread can be visible to others as well. Therefore, the displayed value of shared_var remains 0 across threads even after the modification within the task. This example showcases how the shared clause allows data to be accessed and modified by multiple threads in a parallel context, potentially leading to unintended data races or conflicts.