Reg. No: 23361
MTCS-103(P)
Parallel Processing
Assignment 1

Directives:

In OpenMP, directives are used to define sections of code that should be executed in parallel. These directives apply to the subsequent structured block, which can be a single statement or a compound statement enclosed within curly braces. They offer instructions to the compiler for creating parallel threads and managing synchronization among them. By strategically placing directives, programmers can specify which parts of the code should run concurrently, optimizing the performance of their programs on multicore processors and shared memory systems. This approach enhances program efficiency and scalability by harnessing parallel processing capabilities.

Parallel

The OpenMP parallel construct creates a team of threads and initiates parallel execution of the code enclosed within the construct. Each thread within the team executes the enclosed code independently and concurrently. This construct allows for efficient utilization of multiple processor cores and facilitates parallelism in the program, enhancing its overall performance and scalability.

#pragma omp parallel [clause[[,]clause] ...]

structured-block

Clause:

if(scalar-expression)

```
#include <stdio.h>
#include <omp.h>
int main() {
   int data[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10,11,12};
   int size = sizeof(data) / sizeof(data[0]);
   int threshold = 5;
   printf("Starting the program.\n");
   #pragma omp parallel if(size > threshold)
   {
       int thread_id = omp_get_thread_num();
       int chunk_size = size / omp_get_num_threads();
       int start = thread_id * chunk_size;
       int end = start + chunk_size;

       if (size > threshold) {
           printf("Thread %d is processing elements from index %d to %d.\n", thread_id, start,
end);
           for (int i = start; i < end; ++i) {
               data[i] *= 2;            }
       } else {
           printf("Thread %d is processing the entire array.\n", thread_id);
           for (int i = 0; i < size; ++i) {
               data[i] *= 2;            }
       }
   }

   printf("All processing has been completed.\n");

      printf("Modified array: ");
   for (int i = 0; i < size; ++i) {
       printf("%d ", data[i]);
   }
   printf("\n");

   return 0;
}
```

Output:

```
et2023@dmacs13-OptiPlex-9020:~/OPENMP$ gcc -fopenmp ifclause.c
et2023@dmacs13-OptiPlex-9020:~/OPENMP$ ./a.out
Starting the program.
Thread 0 is processing elements from index 0 to 3.
Thread 1 is processing elements from index 3 to 6.
Thread 3 is processing elements from index 9 to 12.
Thread 2 is processing elements from index 6 to 9.
All processing has been completed.
Modified array: 2 4 6 8 10 12 14 16 18 20 22 24
```

Explanation:

This code uses the if clause in the parallel construct to dynamically determine whether to enable parallelism based on the size of the array. If the array size exceeds a threshold, the work is distributed among threads to process chunks of the array concurrently. Otherwise, if the array is smaller, each thread processes the entire array sequentially.

num_threads(integer-expression):

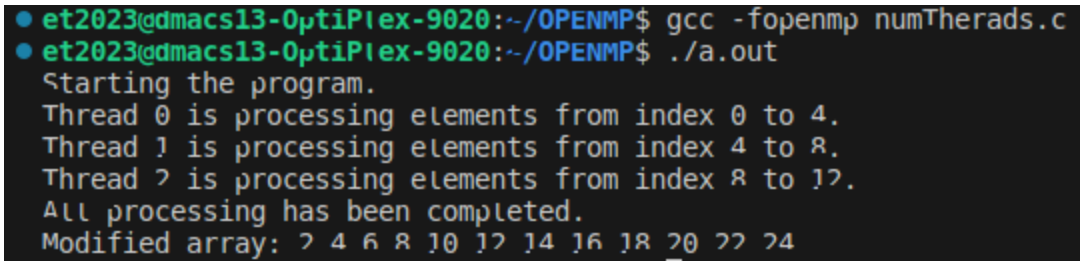
The num_threads clause in OpenMP is used to specify the exact number of threads that should be utilized within a parallel region. By providing an integer value as an argument to this clause, you can control the exact count of threads that will participate in the parallel execution. This allows for precise control over the level of concurrency and resource utilization in the program.

```c
#include <stdio.h>

#include <omp.h>

int main() {

   int data[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10,11,12};

   int size = sizeof(data) / sizeof(data[0]);
```

```
   int threshold = 5;

   int num_threads = 3;    printf("Starting the program.\n");

   #pragma omp parallel if(size > threshold) num_threads(num_threads)

   {

       int thread_id = omp_get_thread_num();

       int chunk_size = size / num_threads;

       int start = thread_id * chunk_size;

       int end = start + chunk_size;

             if (size > threshold) {

           printf("Thread %d is processing elements from index %d to
%d.\n", thread_id, start, end);

           for (int i = start; i < end; ++i) {

               data[i] *= 2;             }

       } else {

           printf("Thread %d is processing the entire array.\n",
thread_id);

           for (int i = 0; i < size; ++i) {

               data[i] *= 2;             }

       }

   }

   printf("All processing has been completed.\n");

   printf("Modified array: ");
```

```
    for (int i = 0; i < size; ++i) {

        printf("%d ", data[i]);

    }

    printf("\n");

    return 0;

}
```

Output:



Explanation:

The num_threads clause in OpenMP is used to specify the exact number of threads that should be utilized within a parallel region. By providing an integer value as an argument to this clause, you can control the exact count of threads that will participate in the parallel execution. This allows for precise control over the level of concurrency and resource utilization in the program.

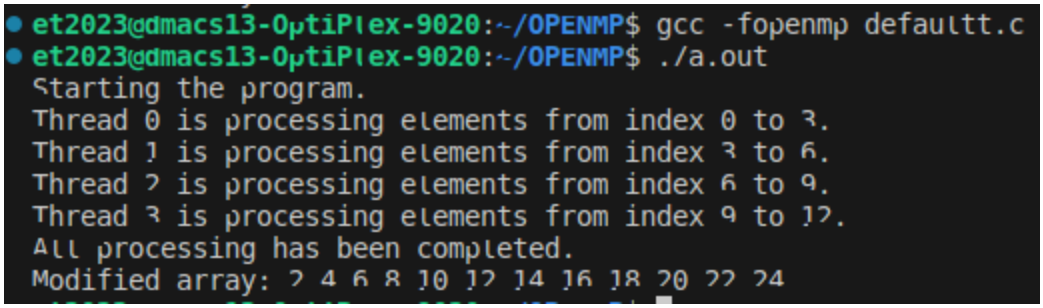
default(shared | none)

The default clause in OpenMP is used to control the default data-sharing attributes of variables in a parallel or task construct. When default(shared) is specified, all variables not explicitly listed in other data-sharing clauses are

treated as shared by default. On the other hand, when default(none) is used, all variables must be explicitly declared using data-sharing clauses to avoid unintended shared access. This clause provides a way to ensure that the program's data-sharing behavior is explicitly specified, enhancing the clarity and safety of parallel execution.

```
#include <stdio.h>
#include <omp.h>
int main() {
   int data[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10,11,12,13};
   int size = sizeof(data) / sizeof(data[0]);
   int threshold = 5;
   printf("Starting the program.\n");
   #pragma omp parallel default(shared)
   {
       int thread_id = omp_get_thread_num();
       int chunk_size = size / omp_get_num_threads();
       int start = thread_id * chunk_size;
       int end = start + chunk_size;

            printf("Thread %d is processing elements from index %d to
%d.\n", thread_id, start, end);
       for (int i = start; i < end; ++i) {
           data[i] *= 2;         }
   }
   printf("All processing has been completed.\n");

   printf("Modified array: ");
   for (int i = 0; i < size; ++i) {
       printf("%d ", data[i]);
   }
   printf("\n");
   return 0;
}
```

Output:



private(list):

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

Explanation:

This code uses OpenMP to run multiple threads concurrently. Each thread calculates its private variable, which is the thread ID plus 1. Threads then synchronize, update a shared variable atomically by adding their private variable, and synchronize again. The program prints each thread's private variable and the updated shared variable, showcasing parallel execution and synchronization

```c
#include <stdio.h>
#include <omp.h>
int main() {
   int list[4] = {0, 0, 0, 0};
   #pragma omp parallel firstprivate(list)
   {
       int thread_id = omp_get_thread_num();
       list[thread_id] = thread_id + 1;
       printf("Thread %d: List[%d] = %d\n", thread_id, thread_id,
list[thread_id]);
   }
   printf("Original List: [%d, %d, %d, %d]\n", list[0], list[1], list[2],
list[3]);

   return 0;
}.
```

Output:

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

Explanation:
Here, the firstprivate(list) directive ensures that each thread gets its private copy of the list array with initial values copied from the original list. Threads then modify their private copy of the list, and the program prints each thread's private copy. Finally, the program prints the original list values, which remain unaffected by the threads' modifications.

Shared(list):

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

Explanation:

Here the shared(list) directive indicates that the list array is shared among all threads within the parallel region. Threads concurrently modify the shared list, and a critical section is used to ensure that only one thread prints the shared list at a time, preventing interleaved output.

reduction(operator:list):

```c
#include <stdio.h>
#include <omp.h>
int main() {
   int list[4] = {1, 2, 3, 4};
   int sum = 0;      #pragma omp parallel for reduction(+:sum)
   for (int i = 0; i < 4; ++i) {
       sum += list[i];
   printf("Sum of the list: %d\n", sum);
   return 0;
}
```

Output:

```
et2023@dmacs13-OptiPlex-9020:~/OPENMP$ gcc -fopenmp reduction.c
et2023@dmacs13-OptiPlex-9020:~/OPENMP$ ./a.out
Sum of the list: 10
```

Explanation:

Here the reduction(+:sum) clause is used in conjunction with the parallel for construct. Each thread computes a partial sum of the list array, and the reduction operation + accumulates the partial sums into the sum variable.