Reg. No: 23361
MTCS-103(P)
Parallel Processing
Assignment 2

Loop:

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

for-loops

The loop construct in OpenMP, specified by #pragma omp for, enables the distribution and parallel execution of loop iterations across the available threads in a team. This construct optimizes the execution of for-loops by dividing their iterations among the threads. The clauses associated with the loop construct provide control over loop distribution, scheduling, and data sharing among the threads, allowing for efficient and parallel computation of loop-based tasks.

Private:

```c
#include <stdio.h>

#include <omp.h>

#define ARRAY_SIZE 10

int main() {

    int my_array[ARRAY_SIZE];

    int total_sum = 0;
```

```
    for (int i = 0; i < ARRAY_SIZE; i++) {
        my_array[i] = i + 1;
    }
    int local_sum = 0;
    #pragma omp parallel for private(local_sum)
    for (int i = 0; i < ARRAY_SIZE; i++) {
        local_sum += my_array[i];
        printf("Thread %d: Partial sum = %d\n", omp_get_thread_num(), local_sum);
    }
    return 0;
}
```

Output:

```
Thread 0: Partial sum = 1
Thread 2: Partial sum = 4
Thread 1: Partial sum = 3
Thread 3: Partial sum = 5
Thread 0: Partial sum = 6
Thread 2: Partial sum = 9
Thread 1: Partial sum = 8
Thread 3: Partial sum = 10
Thread 0: Partial sum = 11
Thread 2: Partial sum = 14
```

Explanation:

This code employs the OpenMP private clause to establish private copies of a variable within a parallel loop. It initializes an integer array my_array and enters a parallel loop where each thread possesses its personal local_sum variable. This variable accumulates partial sums for each thread as they process the elements of my_array. The printf statement showcases the evolving partial sum within each thread. By isolating the variable using the private clause, conflicts are avoided, and individual threads can independently compute their partial sums. Although the final sum isn't explicitly shown, the code reveals how threads perform distinct computations while accumulating partial sums.

Firstprivate:

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

int main() {
    int base_value = 5;

    #pragma omp parallel for firstprivate(base_value)
    for (int i = 0; i < 4; i++) {
        int private_copy = base_value;
         private_copy += i;
        printf("Thread %d: Private Copy = %d, i = %d\n", omp_get_thread_num(),
private_copy, i);
    }

    printf("After parallel region: Base Value = %d\n", base_value);
    return 0;
}
```

Output:

```
Thread 0: Partial sum = 1
Thread 2: Partial sum = 4
Thread 1: Partial sum = 3
Thread 3: Partial sum = 5
Thread 0: Partial sum = 6
Thread 2: Partial sum = 9
Thread 1: Partial sum = 8
Thread 3: Partial sum = 10
Thread 0: Partial sum = 11
Thread 2: Partial sum = 14
```

Explanation:
In this code, the variable base_value is shared among threads. The parallel loop is executed by multiple threads, each with its own private copy of base_value, initialized with the shared value of 5. The loop increments the private copy by the loop index i. The printf statement displays the private copy of base_value and the loop index i for each thread. After the parallel loop, the main thread outputs the unchanged shared value of base_value. The output will show how each thread's private copy is modified independently within the loop, while the original shared value remains unchanged outside the loop.

lastPrivate:

```c
#include <stdio.h>
#include <omp.h>
int main() {
    int last_private_sum = 0;
    #pragma omp parallel for lastprivate(last_private_sum)
    for (int i = 1; i <= 10; i++) {
        last_private_sum += i;
        printf("Thread %d: Partial sum = %d\n", omp_get_thread_num(),
last_private_sum);
    }
    printf("After parallel region: Last Private Sum = %d\n", last_private_sum);
    return 0;
}
```

Output:

```
Thread 0: Partial sum = 1
Thread 1: Partial sum = 3
Thread 2: Partial sum = 6
Thread 3: Partial sum = 10
Thread 4: Partial sum = 15
Thread 5: Partial sum = 21
Thread 6: Partial sum = 28
Thread 7: Partial sum = 36
Thread 8: Partial sum = 45
Thread 9: Partial sum = 55
After parallel region: Last Private Sum = 55
```

Explanation:
This code demonstrates the use of the lastprivate clause in OpenMP. Within a parallel region, a loop iterates from 1 to 10, with each thread maintaining a partial sum using the variable last_private_sum. The lastprivate clause ensures that after the loop, the value of last_private_sum from the last iteration of the loop is assigned to the shared variable outside the parallel region. Each thread prints its partial sum during the loop iterations, and after the loop, the program outputs the final value of last_private_sum, which is the sum of numbers from 1 to 10.

Reduction:

```c
#include <stdio.h>
#include <omp.h>
#define ARRAY_SIZE 10
int main() {
    int data[ARRAY_SIZE];
    int total_sum = 0;
    for (int i = 0; i < ARRAY_SIZE; i++) {
        data[i] = i + 1;
    }
    int partial_sum = 0;
```

```
    #pragma omp parallel for reduction(+ : partial_sum) schedule(dynamic, 2)
    for (int i = 0; i < ARRAY_SIZE; i++) {
        int thread_id = omp_get_thread_num();
        printf("Thread %d: Processing element %d (Value: %d)\n", thread_id, i,
data[i]);
        partial_sum += data[i];
    }

    int thread_id = omp_get_thread_num();
    printf("Thread %d: Partial Sum = %d\n", thread_id, partial_sum);

    return 0;
}
```

Output:

```
Thread 0: Processing element 0 (Value: 1)
Thread 2: Processing element 2 (Value: 3)
Thread 0: Processing element 4 (Value: 5)
Thread 1: Processing element 1 (Value: 2)
Thread 2: Processing element 6 (Value: 7)
Thread 1: Processing element 8 (Value: 9)
Thread 3: Processing element 3 (Value: 4)
Thread 3: Processing element 5 (Value: 6)
Thread 1: Processing element 7 (Value: 8)
Thread 3: Processing element 9 (Value: 10)
Thread 0: Partial Sum = 55
```

Explanation:

The above code computes the sum of elements in an array of size 10. Each thread processes a subset of elements using a dynamic scheduling strategy with a chunk size of 2. The reduction clause efficiently calculates the partial sum for each thread and aggregates them into a final result. The output includes information about each thread's processing, the index and value of the processed element, as well as the individual thread's partial sum.

Collapse:

```
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
#define ARRAY_SIZE 3

int main() {
    int array[ARRAY_SIZE][ARRAY_SIZE];

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

    printf("Original 2D Array:\n");
    for (int i = 0; i < ARRAY_SIZE; ++i) {
        for (int j = 0; j < ARRAY_SIZE; ++j) {
            printf("%3d ", array[i][j]);
        }
        printf("\n");
    }

    #pragma omp parallel for collapse(2)
    for (int i = 0; i < ARRAY_SIZE; ++i) {
        for (int j = 0; j < ARRAY_SIZE; ++j) {
            array[i][j] *= 2;  // Perform some processing on the element
            printf("Thread %d: Array[%d][%d] = %d\n", omp_get_thread_num(), i, j,
array[i][j]);
        }
    }
    printf("Modified 2D Array:\n");
    for (int i = 0; i < ARRAY_SIZE; ++i) {
        for (int j = 0; j < ARRAY_SIZE; ++j) {
            printf("%3d ", array[i][j]);
        }
```

```
        printf("\n");
    }
    return 0;
}
```

Output:

```
Original 2D Array:
   1    2    3
   4    5    6
   7    8    9

Thread 1: Array[0][0] = 2
Thread 0: Array[0][1] = 4
Thread 2: Array[0][2] = 6
Thread 1: Array[1][0] = 10
Thread 2: Array[1][1] = 12
Thread 0: Array[1][2] = 14
Thread 1: Array[2][0] = 18
Thread 0: Array[2][1] = 20
Thread 2: Array[2][2] = 22

Modified 2D Array:
   2    4    6
  10   12   14
  18   20   22
```

Explanation:
The array is initialized and displayed, and then each element is doubled using the parallel for loop with a collapse(2) clause. The output indicates which thread is processing each element and its corresponding value. Finally, the modified array is displayed. The collapse clause helps to parallelize nested loops efficiently, improving performance by distributing the iterations among multiple threads.

Ordered:

```
#include <stdio.h>
#include <omp.h>
int main() {
    #pragma omp parallel num_threads(3)
    {
```

```
        #pragma omp for ordered
        for (int i = 1; i <= 10; ++i) {
            int thread_num = omp_get_thread_num();
            #pragma omp ordered
            {
                printf("Thread %d: Number %d\n", thread_num, i);
            }
        }
    }

    return 0;
}
```

Output:

```
Thread 0: Number 1
Thread 0: Number 4
Thread 0: Number 7
Thread 0: Number 10
Thread 1: Number 2
Thread 1: Number 5
Thread 1: Number 8
Thread 2: Number 3
Thread 2: Number 6
Thread 2: Number 9
```

Explanation:

The code uses the `ordered` clause within a parallel region with three threads. Each thread enters the loop and prints the numbers from 1 to 10. The `ordered` clause ensures that the output for each thread is printed in order of the thread number. While the threads may execute their iterations concurrently, the `ordered` clause maintains the output order based on thread number.

Sections

The sections construct contains a set of structured blocks that are to be distributed among and executed by the encountering team of threads.

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

{

[#pragma omp section]

Structured-block

[#pragma omp section

structured-block] ...

}

The `sections` construct in OpenMP divides a program's sections of code into separate blocks that can be executed concurrently by the threads in a parallel region. Each block is enclosed within a `section` directive, and the work within each section is independent of the others. Threads within the parallel region execute these sections in parallel, distributing the work evenly among themselves. This construct is particularly useful when multiple independent tasks need to be executed concurrently by different threads, improving parallelism and resource utilization. The execution order of the sections is not guaranteed, allowing for efficient load balancing among threads.

private(list):

```
#include <stdio.h>

#include <omp.h>

int main() {

    int shared_value = 10;

    #pragma omp parallel

    {

        #pragma omp sections private(shared_value)

        {

            #pragma omp section

            {

                int thread_id = omp_get_thread_num();

                shared_value += thread_id;

                printf("Thread %d in Section 1: shared_value = %d\n", thread_id,
shared_value);

            }

            #pragma omp section

            {

                int thread_id = omp_get_thread_num();

                shared_value -= thread_id;
```

```
                printf("Thread %d in Section 2: shared_value = %d\n", thread_id,
shared_value);

            }

        }

    }

    printf("After sections: shared_value = %d\n", shared_value);

    return 0;

}
```

Output:

```
Thread 0 in Section 1: shared_value = 10
Thread 0 in Section 2: shared_value = 10
Thread 3 in Section 1: shared_value = 13
Thread 2 in Section 2: shared_value = 9
Thread 1 in Section 1: shared_value = 11
Thread 1 in Section 2: shared_value = 9
Thread 2 in Section 1: shared_value = 12
Thread 3 in Section 2: shared_value = 8
After sections: shared_value = 10
```

Explanation:

In this code, the `parallel` construct is used to create a parallel region first. Inside the parallel region, the `sections` construct is used with the `private(shared_value)` clause, which ensures that each thread has its private copy of the `shared_value` variable within each section.

lastprivate(list):

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

int main() {
    int shared_value = 10;

    #pragma omp parallel sections lastprivate(shared_value)
    {
        #pragma omp section
        {
            shared_value += 1;
            printf("Thread %d in Section 1: shared_value = %d\n",
omp_get_thread_num(), shared_value);
        }

        #pragma omp section
        {
            shared_value -= 2;
            printf("Thread %d in Section 2: shared_value = %d\n",
omp_get_thread_num(), shared_value);
        }
    }

    printf("After sections: shared_value = %d\n", shared_value);

    return 0;
}
```

Output:

```
Thread 0 in Section 1: shared_value = 11
Thread 1 in Section 2: shared_value = 8
After sections: shared_value = 11
```

Explanation:
The provided code uses OpenMP's `sections` construct with the `lastprivate` clause. Within the parallel sections, two blocks of code are executed by different threads. Each block modifies a shared variable `shared_value`. The `lastprivate` clause ensures that the value of `shared_value` from the last executed section is preserved after the sections construct. In the output, the values of `shared_value` within each section are updated by the threads, and the value from the last section is captured by `lastprivate`, ultimately displayed as the final value of `shared_value` after the sections construct.

Nowait:

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

int main() {
    int shared_value = 0;

    #pragma omp parallel sections nowait
    {
        #pragma omp section
        {
            shared_value += 1;
            printf("Thread %d: Section 1 - Shared Value = %d\n",
omp_get_thread_num(), shared_value);
        }

        #pragma omp section
        {
            shared_value += 2;
            printf("Thread %d: Section 2 - Shared Value = %d\n",
omp_get_thread_num(), shared_value);
        }

        #pragma omp section
        {
            shared_value += 3;
            printf("Thread %d: Section 3 - Shared Value = %d\n",
omp_get_thread_num(), shared_value);
```

```
        }
    }

    printf("After parallel sections: Shared Value = %d\n", shared_value);

    return 0;
}
```

Output:

```
Thread 1: Section 1 - Shared Value = 1
Thread 0: Section 2 - Shared Value = 2
Thread 2: Section 3 - Shared Value = 3
After parallel sections: Shared Value = 6
```

Explanation:

In this code, the `nowait` clause is used in conjunction with the `sections` construct. The three sections are executed concurrently by different threads, and each section modifies the `shared_value`. The `nowait` clause ensures that there is no implicit barrier at the end of the `sections` construct, allowing the threads to continue executing immediately after completing their respective sections. As a result, the order of section execution and the final value of `shared_value` are non-deterministic due to the parallel execution.