Week No. 2 (a): Sharing of work among threads using Loop Construct in OpenMP

THEORY

Introduction

OpenMP's work-sharing constructs are the most important feature of OpenMP. They are used to distribute computation among the threads in a team. C/C++ has three work-sharing constructs. A work-sharing construct, along with its terminating construct where appropriate, specifies a region of code whose work is to be distributed among the executing threads; it also specifies the manner in which the work in the region is to be parceled out. A work-sharing region must bind to an active parallel region in order to have an effect. If a work-sharing directive is encountered in an inactive parallel region or in the sequential part of the program, it is simply ignored. Since work-sharing directives may occur in procedures that are invoked both from within a parallel region as well as outside of any parallel regions, they may be exploited during some calls and ignored during others.

The work-sharing constructs are listed below.

- **#pragma omp for:** Distribute iterations over the threads
- **#pragma omp sections:** Distribute independent work units
- **#pragma omp single:** Only one thread executes the code block

The two main rules regarding work-sharing constructs are as follows:

- Each work-sharing region must be encountered by all threads in a team or by none at all.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.

A work-sharing construct does not launch new threads and does not have a barrier on entry. By default, threads wait at a barrier at the end of a work-sharing region until the last thread has completed its share of the work. However, the programmer can suppress this by using the nowait clause.

The Loop Construct

The *loop construct* causes the iterations of the loop immediately following it to be executed in parallel. At run time, the loop iterations are distributed across the threads. This is probably the most widely used of the work-sharing features.

Format:

```
#pragma omp for [clause ...]  newline

                    schedule (type [,chunk])              ordered
        private (list)              firstprivate (list)              lastprivate
        (list)              shared (list)              reduction (operator:
        list)              nowait
  for_loop
```

Example of a work-sharing loop

Each thread executes a subset of the total iteration space $i = 0, \ldots, n - 1$

```
#include <omp.h>
main()
{
        #pragma omp parallel shared(n) private(i)
        {
                #pragma omp for for (i=0;
                i<n;i++)
                printf("Thread %d executes loop iteration %d\n",
                omp_get_thread_num(),i);
        }
}
```

Here we use a parallel directive to define a parallel region and then share its work among threads via the for work-sharing directive: the **#pragma omp for** directive states that iterations of the loop following it will be distributed. Within the loop, we use the OpenMP function omp_get_thread_num(), this time to obtain and print the number of the executing thread in each iteration. Parallel construct that state which data in the region is shared and which is private. Although not strictly needed since this is enforced by the compiler, loop variable i is explicitly declared to be a private variable, which means that each thread will have its own copy of i. its value is also undefined after the loop has finished. Variable n is made shared.

Output from the example which is executed for $n = 9$ and uses four threads.

```
Thread 0 executes loop iteration 0
Thread 0 executes loop iteration 1
Thread 0 executes loop iteration 2
Thread 3 executes loop iteration 7
Thread 3 executes loop iteration 8
Thread 2 executes loop iteration 5
Thread 2 executes loop iteration 6
Thread 1 executes loop iteration 3
Thread 1 executes loop iteration 4
```

Combined Parallel Work-Sharing Constructs

Combined parallel work-sharing constructs are shortcuts that can be used when a parallel region comprises precisely one work-sharing construct, that is, the work-sharing region includes all the code in the parallel region. The semantics of the shortcut directives are identical to explicitly specifying the parallel construct immediately followed by the worksharing construct.

Full version Combined construct	Combined construct

#pragma omp parallel
{
#pragma omp for for-loop
}
	#pragma omp parallel for
	{
	for-loop
	}

EXERCISE:

1. Code the above example programs and write down their outputs.

Output of Program:

_____
_____
_____
_____
_____
_____
_____
_____
_____

2. Write a parallel program, after discussing your instructor, which uses Loop Construct.

Program:

_____
_____
_____
_____
_____
_____
_____
_____
_____
_____

Week No. 2 (b): Clauses in Loop Construct

THEORY

Introduction

The OpenMP Data Scope Attribute Clauses are used to explicitly define how variables should be scoped.

Data Scope Attribute Clauses are used in conjunction with several directives (PARALLEL, DO/for, and SECTIONS) to control the scoping of enclosed variables.

These constructs provide the ability to control the data environment during execution of parallel constructs.

- They define how and which data variables in the serial section of the program are transferred to the parallel sections of the program (and back)
- They define which variables will be visible to all threads in the parallel sections and which variables will be privately allocated to all threads.

List of Clauses

- PRIVATE
- FIRSTPRIVATE
- LASTPRIVATE
- SHARED
- DEFAULT
- REDUCTION
- COPYIN

PRIVATE Clause

The PRIVATE clause declares variables in its list to be private to each thread.

Format: PRIVATE (list)

Notes:

- PRIVATE variables behave as follows:
  - A new object of the same type is declared once for each thread in the team o All references to the original object are replaced with references to the new object
  - Variables declared PRIVATE are uninitialized for each thread

Example of the private clause – Each thread has a local copy of variables i and a.

```
#pragma omp parallel for private(i,a)

for (i=0; i<n; i++)
{ a = i+1;
                printf("Thread %d has a value of a = %d for i =
    %d\n",
    omp_get_thread_num(),a,i); } /*--
End of parallel for --*/
```

It is the programmer's responsibility to ensure that multiple threads properly access SHARED variables (such as via CRITICAL sections)

SHARED Clause

The SHARED clause declares variables in its list to be shared among all threads in the team. **Format:** SHARED *(list)*

Notes:

 A shared variable exists in only one memory location and all threads can read or write to that address

Example of the shared clause – All threads can read from and write to vector a.

```
#pragma omp parallel for shared(a)

for (i=0; i<n; i++)
{ a[i] += i;
} /*-- End of parallel for --*/
```

DEFAULT Clause

The DEFAULT clause allows the user to specify a default PRIVATE, SHARED, or NONE scope for all variables in the lexical extent of any parallel region.
The default clause is used to give variables a default data-sharing attribute. Its usage is straightforward. For example, default (shared) assigns the shared attribute to all variables referenced in the construct. This clause is most often used to define the data-sharing attribute of the majority of the variables in a parallel region. Only the exceptions need to be explicitly listed.
If default(none) is specified instead, the programmer is forced to specify a data-sharing attribute for each variable in the construct. Although variables with a predetermined

datasharing attribute need not be listed in one of the clauses, it is strongly recommend that the attribute be explicitly specified for *all* variables in the construct.

Format: DEFAULT (SHARED | NONE)

Notes:

- Specific variables can be exempted from the default using the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses
- The C/C++ OpenMP specification does not include "private" as a possible default. However, actual implementations may provide this option.
- Only one DEFAULT clause can be specified on a PARALLEL directive

Example of the Deafulat clause: all variables to be shared, with the exception of a, b, and c.

```
#pragma omp for default(shared) private(a,b,c),
```

FIRSTPRIVATE Clause

The FIRSTPRIVATE clause combines the behavior of the PRIVATE clause with automatic initialization of the variables in its list.

Format: FIRSTPRIVATE *(LIST)*

Notes:

- Listed variables are initialized according to the value of their original objects prior to entry into the parallel or work-sharing construct

Example using the firstprivate clause – Each thread has a pre-initialized copy of variable indx. This variable is still private, so threads can update it individually.

```
for(i=0; i<vlen; i++) a[i] = -i-1; indx = 4;
{
#pragma omp parallel default(none) firstprivate(indx) private(i,TID)
shared(n,a)

        {
                TID = omp_get_thread_num(); indx +=
                n*TID;
                for(i=indx; i<indx+n; i++) a[i] = TID +
                1;
        }
}
```

```
printf("After the parallel region:\n"); for (i=0; i<vlen;
i++) printf("a[%d] = %d\n",i,a[i]);
```

LASTPRIVATE Clause

The LASTPRIVATE clause combines the behavior of the PRIVATE clause with a copy from the last loop iteration or section to the original variable object.

Format: LASTPRIVATE *(LIST)*

Notes:

- The value copied back into the original variable object is obtained from the last (sequentially) iteration or section of the enclosing construct.
- It ensures that the last value of a data object listed is accessible after the corresponding construct has completed execution

Example of the lastprivate clause – This clause makes the sequentially last value of variable a accessible outside the parallel loop.

```
#pragma omp parallel for private(i) lastprivate(a) for (i=0; i<n; i++)
        { a = i+1;
                                printf("Thread %d has a value of a = %d for i =
            %d\n",
            omp_get_thread_num(),a,i);
        } /*-- End of parallel for --*/

 printf("Value of a after parallel for: a = %d\n",a);
```

COPYIN Clause

The COPYIN clause provides a means for assigning the same value to THREADPRIVATE variables for all threads in the team.

Format: COPYIN *(LIST)*

Notes:

- List contains the names of variables to copy. The master thread variable is used as the copy source. The team threads are initialized with its value upon entry into the parallel construct.

REDUCTION Clause

The REDUCTION clause performs a reduction on the variables that appear in its list. A private copy for each list variable is created for each thread. At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable.

Format: REDUCTION *(OPERATOR: LIST)*

Notes:

- Variables in the list must be named scalar variables. They can not be array or structure type variables. They must also be declared SHARED in the enclosing context.
- Reduction operations may not be associative for real numbers.
- The REDUCTION clause is intended to be used on a region or work-sharing construct in which the reduction variable is used only in statements which have one of following forms:

C / C++
x = x op expr **x = expr op x** (except subtraction) **x binop = expr** **x++ ++x** **x-- --x**
x is a scalar variable in the list ***expr*** is a scalar expression that does not reference ***x*** ***op*** is not overloaded, and is one of +, *, -, /, &, ^, |, &&, || ***binop*** is not overloaded, and is one of +, *, -, /, &, ^, |

Example of REDUCTION - Vector Dot Product. Iterations of the parallel loop will be distributed in equal sized blocks to each thread in the team (SCHEDULE STATIC). At the end of the parallel loop construct, all threads will add their values of "result" to update the master thread's global copy.

```
#include <omp.h> main ()
{ int   i, n, chunk; float a[100], b[100], result;
        n = 100; chunk = 10; result = 0.0;

        for (i=0; i < n; i++)
          {
          a[i] = i * 1.0;   b[i] = i *
        2.0;
          }
        #pragma omp parallel for  default(shared) private(i)  schedule(static,chunk)
        reduction(+:result)
          {
            for (i=0; i < n; i++)
            result = result + (a[i] * b[i]);

        printf("Final result= %f\n",result);
```

```
        }
}
```

SCHEDULE:

Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent.

STATIC

Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads.

DYNAMIC

Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads; when a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1.

GUIDED

For a chunk size of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a chunk size with value k (greater than 1), the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than k iterations (except for the last chunk to be assigned, which may have fewer than k iterations). The default chunk size is 1.

Nowait Clause

The nowait clause allows the programmer to fine-tune a program's performance. When we introduced the work-sharing constructs, we mentioned that there is an implicit barrier at the end of them. This clause overrides that feature of OpenMP; in other words, if it is added to a construct, the barrier at the end of the associated construct will be suppressed. When threads reach the end of the construct, they will immediately proceed to perform other work. Note, however, that the barrier at the end of a parallel region cannot be suppressed.

Example of the nowait clause in C/C++ – The clause ensures that there is no barrier at the end of the loop.

```
#pragma omp for nowait for (i=0;
        i<n; i++)
        {
        .....
        }
```

Clauses / Directives Summary

Table 1 Comparative Analysis for a set of instruction

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●			●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

- The following OpenMP directives do not accept clauses:
  - MASTER
  - CRITICAL
  - BARRIER
  - ATOMIC
  - FLUSH
  - ORDERED
  - THREADPRIVATE
- Implementations may (and do) differ from the standard in which clauses are supported by each directive.

EXERCISE:

1. Code the above example programs and write down their outputs.

Output of example programs

_____
_____
_____
_____

2. Write a parallel program that sums a given arrays using reduction clause.

Program:

Week No. 2 (c): Parallelizing a Loop with Scheduling

We use the **schedule** clause to specify how loop iterations are distributed among threads. The **dynamic** scheduling allows the scheduler to assign a new iteration to the next available thread.

Objective: Understand and experiment with loop scheduling in OpenMP to parallelize a loop with different scheduling options.

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

int main() {
   int n = 100;
   int result[n];

   #pragma omp parallel
   {
      // Experiment with different scheduling options: static, dynamic, guided
      #pragma omp for schedule(dynamic)
      for (int i = 0; i < n; i++) {
         int thread_id = omp_get_thread_num();
         result[i] = thread_id * n + i;
      }
   }

   printf("Results with dynamic scheduling:\n");
   for (int i = 0; i < n; i++) {
      printf("%d ", result[i]);
   }
   printf("\n");

   return 0;
}
```

1. **Explanation:**
   - The code sets up a parallel region using **#pragma omp parallel**.
   - Inside the parallel region, we parallelize a loop with dynamic scheduling using **#pragma omp for schedule(dynamic)**.
   - The loop assigns unique values to the elements of the **result** array based on thread ID and loop iteration.
2. **Experiment:**
   - Compile and run the program to observe the output with dynamic scheduling.
   - Experiment with different scheduling options (**static**, **dynamic**, **guided**) by changing the **schedule** clause in the code.
   - Observe how changing the scheduling strategy affects the order in which loop iterations are executed.

3. **Task:**
    - Discuss the differences between static, dynamic, and guided scheduling.
    - Analyze how different scheduling strategies impact the order of execution and potentially improve load balancing in parallel loops.