Parallel Computing

Ekkapot Charoenwanit

Software Systems Engineering
TGGS
KMUTNB

Lecture 8:

- ❑ Shared-Memory Programming with OpenMP
  - ➢ Tasking

OpenMP: Tasking

OpenMP tasking provides a new parallel programming paradigm

- ❑ called the work-oriented paradigm
- ❑ based on the concept of a task pool/task queue

In this work-oriented paradigm, units of work (referred to as OpenMP tasks)

- ❑ are generated or "pushed" into the task pool by threads
- ❑ are retrieved or "pulled" off the task pool and executed by threads

Recall the producer-consumer pattern that we discussed in Lecture 3 !!!

OpenMP: Tasking

In classic OpenMP, threads are treated as a fundamental concept:

- ❑called the thread-centric paradigm

In the new work-oriented paradigm, we focus on units of work referred to as tasks.

- ❑We must now think how the code can be broken into units of work that can be executed in parallel.

- ❑We can think of a task as
  - ➢ code
  - ➢ data (data environment)

  packaged up as an independent schedulable unit.

OpenMP: Tasking

Threads are assigned to perform the work of each task:

- ❑Tasks may be deferred.
- ❑Tasks may be immediately executed.

Tasks enable irregular computational problems to be parallelized in a natural way, e.g.,

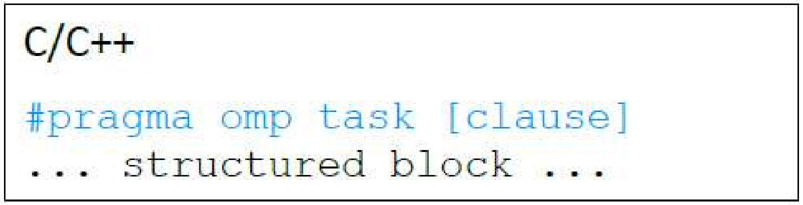
- ❑ Traverse a linked list while performing work on each node in parallel
- ❑ Implement parallel recursive algorithms

OpenMP: Tasks and Threads

A task has

❑ code

❑ a data environment



C/C++

```
#pragma omp task [clause]
... structured block ...
```

Each encountering thread packages a new instance of a task, e.g. code and data.

❑Tasks can be deferred ,i.e., they do not need to be immediately executed.

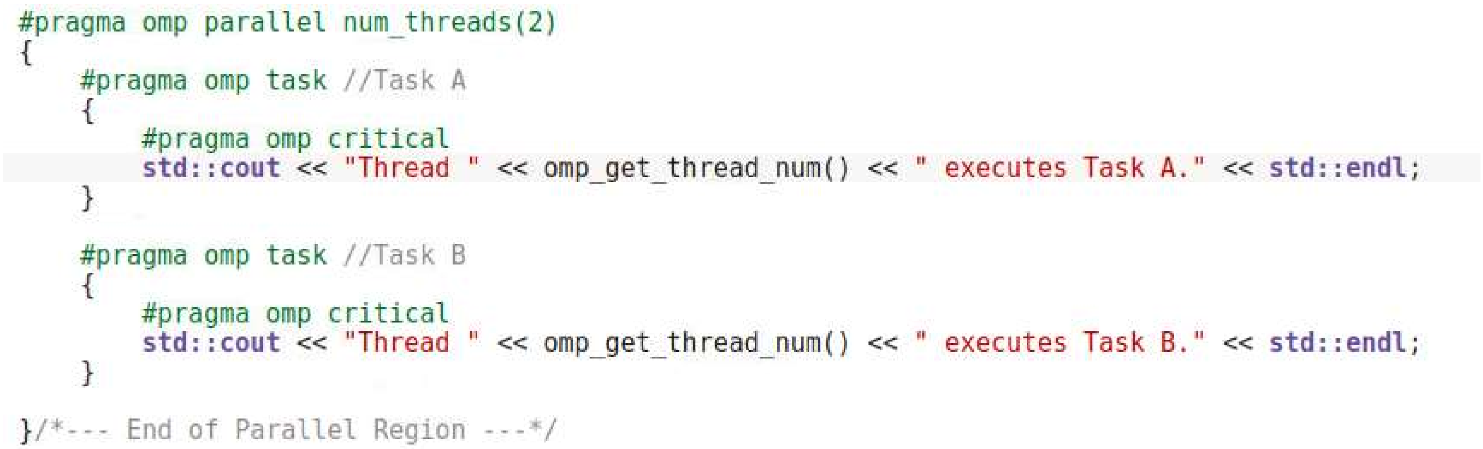
❑Some thread in the team executes the task at some time later.

❑The encountering thread that generate a task does not have to be the same thread that eventually executes the task.

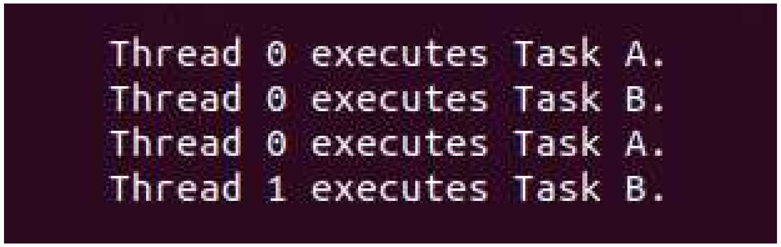
OpenMP: Task Creation

- Parallel regions create tasks:
    - One implicit task is created and assigned to each thread in the team.
- Each thread that encounters a *task* construct:
    - Packages up code and data
    - Creates a new explicit task

OpenMP: Task Creation

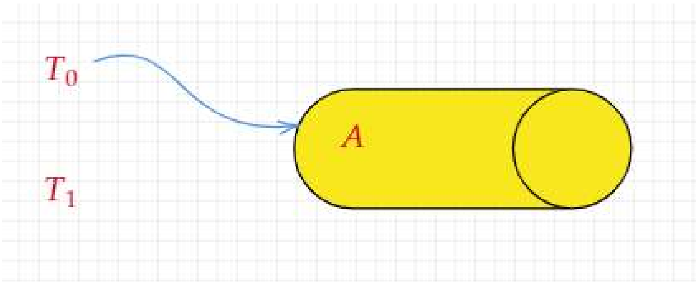


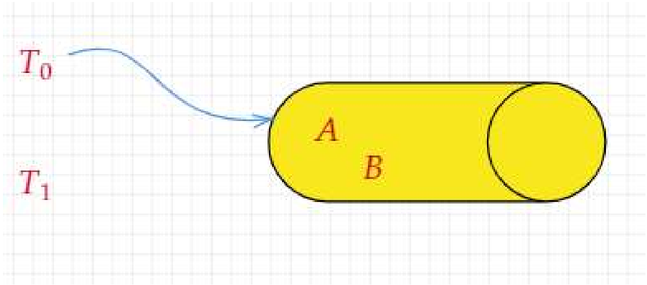
```
#pragma omp parallel num_threads(2)
{
    #pragma omp task //Task A
    {
        #pragma omp critical
        std::cout << "Thread " << omp_get_thread_num() << " executes Task A." << std::endl;
    }

    #pragma omp task //Task B
    {
        #pragma omp critical
        std::cout << "Thread " << omp_get_thread_num() << " executes Task B." << std::endl;
    }

}/*--- End of Parallel Region ---*/
```

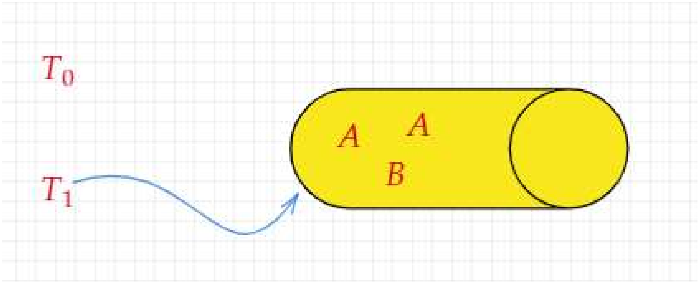


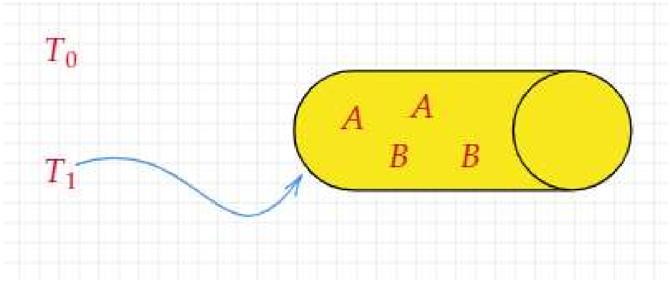
```
Thread 0 executes Task A.
Thread 0 executes Task B.
Thread 0 executes Task A.
Thread 1 executes Task B.
```

OpenMP: Task Creation

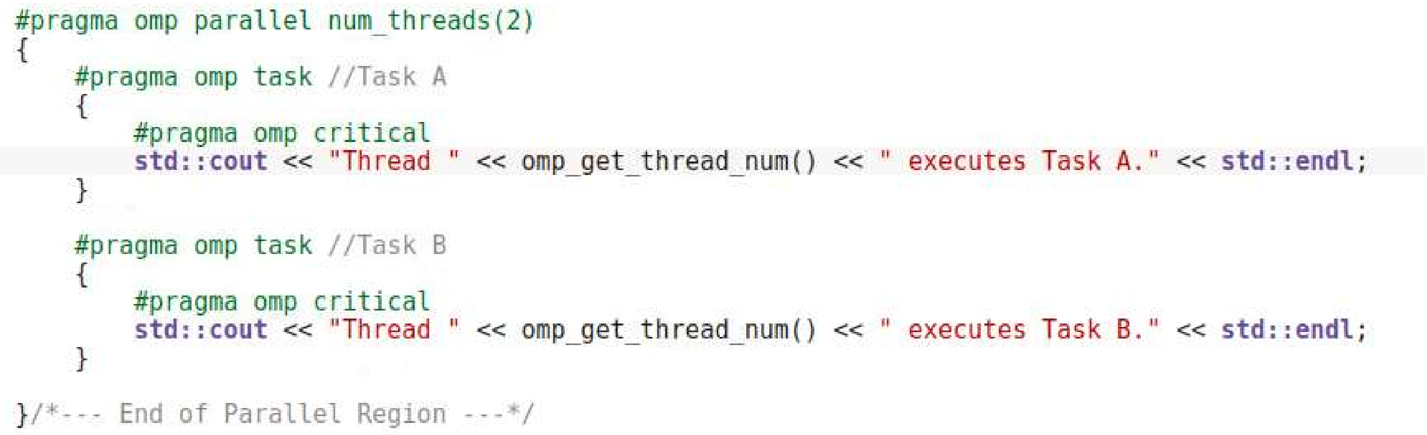








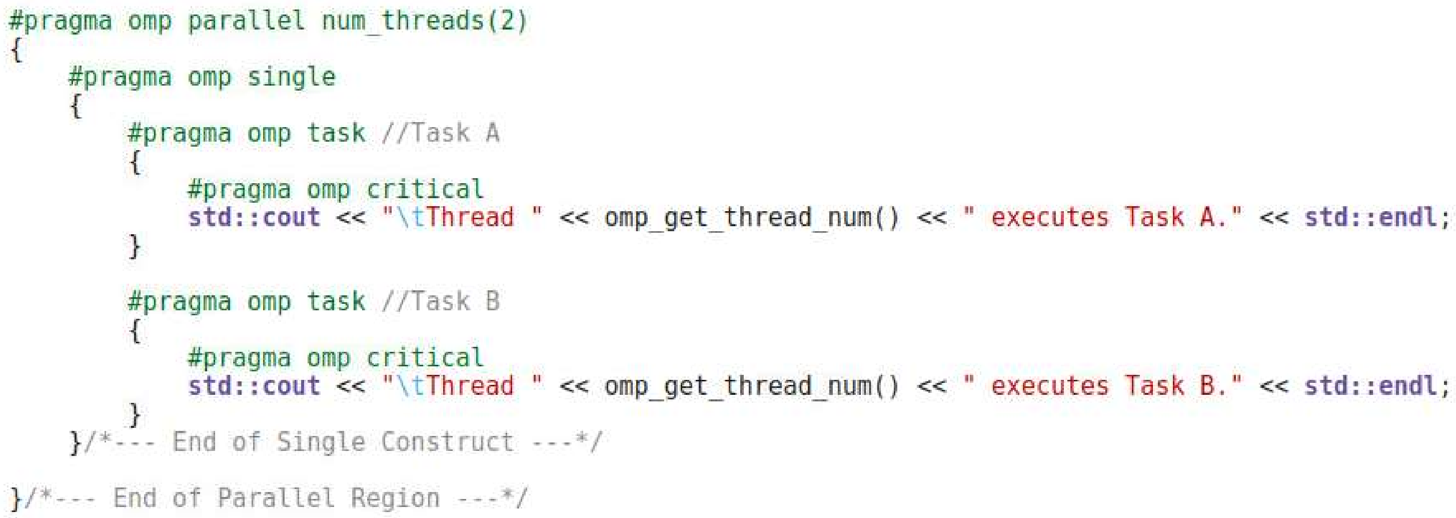
OpenMP: Task Creation



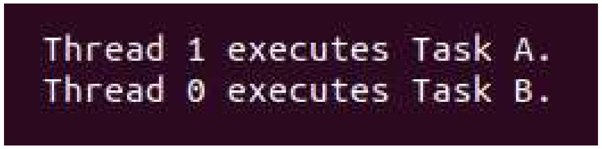
```
#pragma omp parallel num_threads(2)
{
    #pragma omp task //Task A
    {
        #pragma omp critical
        std::cout << "Thread " << omp_get_thread_num() << " executes Task A." << std::endl;
    }

    #pragma omp task //Task B
    {
        #pragma omp critical
        std::cout << "Thread " << omp_get_thread_num() << " executes Task B." << std::endl;
    }

}/*--- End of Parallel Region ---*/
```

- ❑ Each thread in the parallel region pushes two tasks to the task queue, i.e., four tasks are pushed to the task queue in total.
- ❑ There is an implicit barrier at the end of the parallel region, which acts as a task synchronization construct.

OpenMP: Task Creation

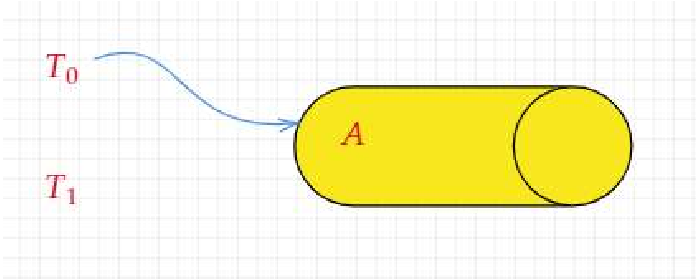


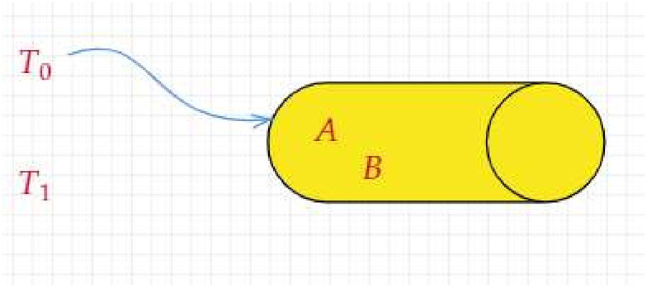
```
#pragma omp parallel num_threads(2)
{
    #pragma omp single
    {
        #pragma omp task //Task A
        {
            #pragma omp critical
            std::cout << "\tThread " << omp_get_thread_num() << " executes Task A." << std::endl;
        }

        #pragma omp task //Task B
        {
            #pragma omp critical
            std::cout << "\tThread " << omp_get_thread_num() << " executes Task B." << std::endl;
        }
    }/*--- End of Single Construct ---*/

}/*--- End of Parallel Region ---*/
```



```
Thread 1 executes Task A.
Thread 0 executes Task B.
```

OpenMP: Task Creation





- ❑ One of the two threads inside the parallel region is in charge of pushing tasks to the task queue.
- ❑ There is an implicit barrier at the end of the single construct, which acts as a task synchronization construct.
- ❑ The *nowait* clause can be used at the single construct to remove this implicit barrier.

OpenMP: Data Scoping

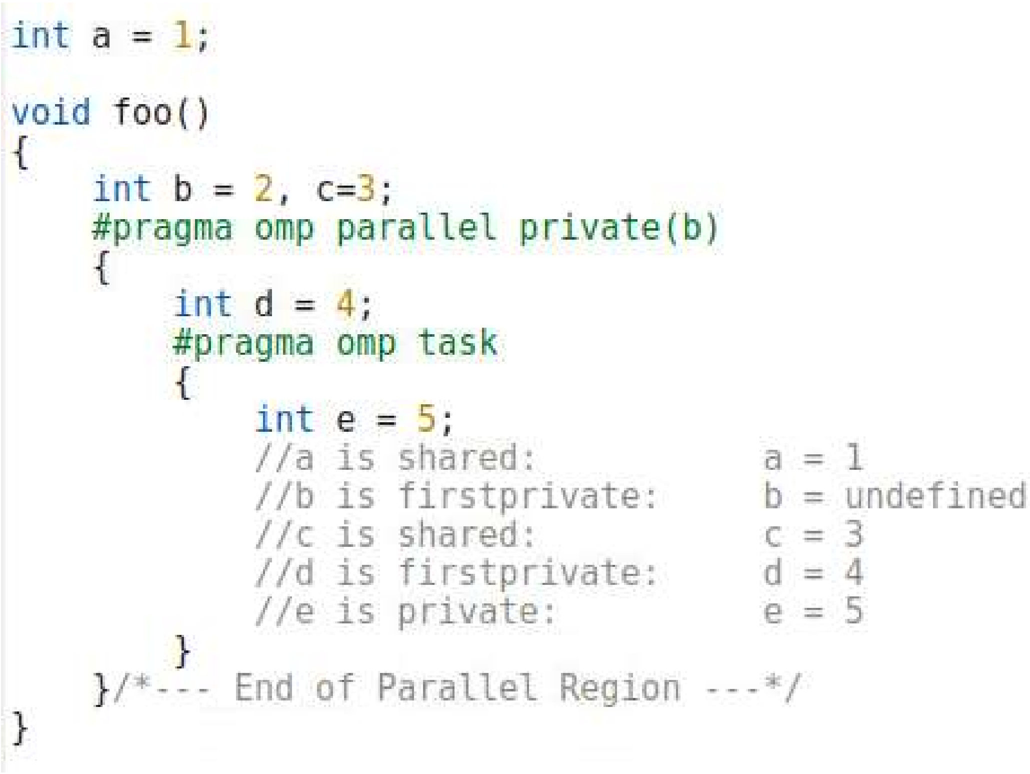
Data Scoping Rules: some rules from the parallel region still apply.

With no *default* clause,

- ❑ static and global variables are *shared*.
- ❑ automatic (local) variables are *private*.
- ❑ Otherwise,
  - ➢ they are *firstprivate*
  - ➢ the *shared* attribute is lexically inherited.

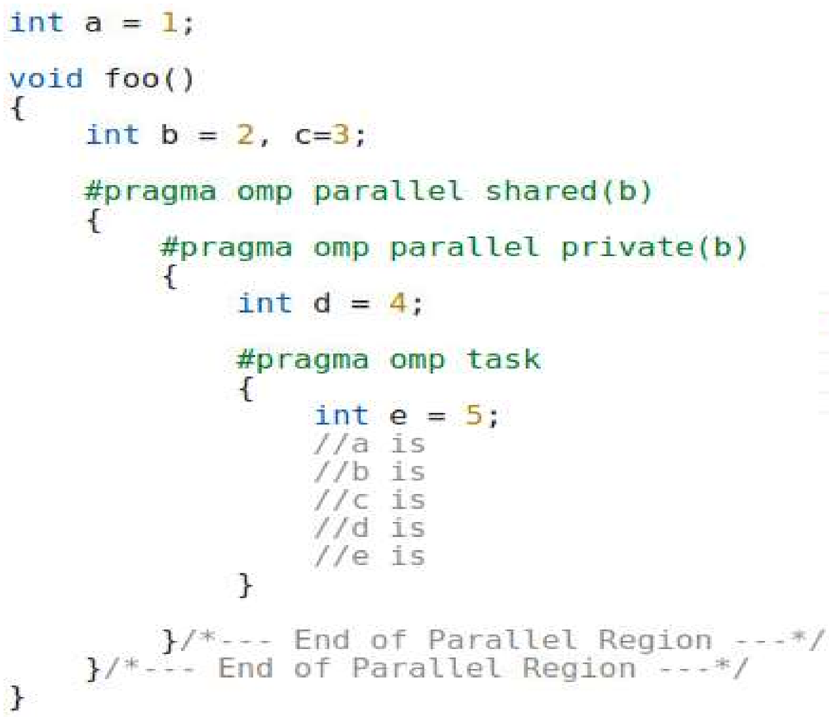
Always use default(none) to force yourself to think carefully !!!

OpenMP: Data Scoping

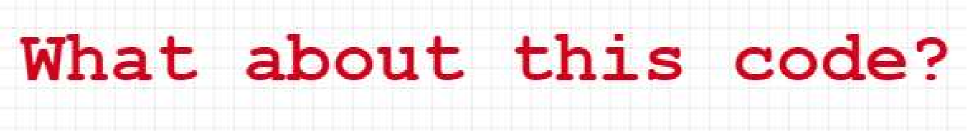


```
int a = 1;

void foo()
{
    int b = 2, c=3;
    #pragma omp parallel private(b)
    {
        int d = 4;
        #pragma omp task
        {
            int e = 5;
            //a is shared:             a = 1
            //b is firstprivate:       b = undefined
            //c is shared:             c = 3
            //d is firstprivate:       d = 4
            //e is private:            e = 5
        }
    }/*--- End of Parallel Region ---*/
}
```

OpenMP: Data Scoping



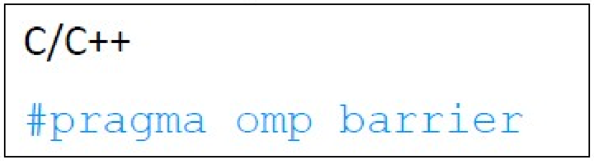
```
int a = 1;

void foo()
{
    int b = 2, c=3;

    #pragma omp parallel shared(b)
    {
        #pragma omp parallel private(b)
        {
            int d = 4;

            #pragma omp task
            {
                int e = 5;
                //a is
                //b is
                //c is
                //d is
                //e is
            }

        }/*--- End of Parallel Region ---*/
    }/*--- End of Parallel Region ---*/
}
```



What about this code?

OpenMP: Task Synchronization

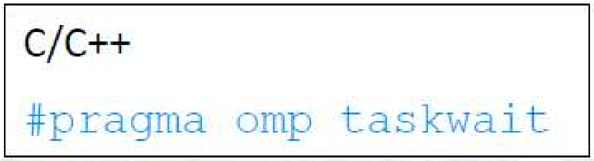
- All tasks created by any thread of the current team are guaranteed to have completed at a barrier (implicit or explicit).



C/C++

```
#pragma omp barrier
```

- A task that encounters a task barrier is suspended until all child tasks complete.
  - This applies only to child tasks, not all descendant tasks.



C/C++

```
#pragma omp taskwait
```

Reference

[1] *Ruud van der Pas, Eric Stotzer, and Christian Terboven. 2017. Using OpenMP -- The Next Step: Affinity, Accelerators, Tasking, and SIMD (1st. ed.). The MIT Press.*