OpenMP

Vipin Vasu

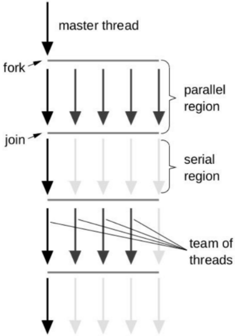
Outline

OpenMP

- Shared memory parallel programming
- OpenMP is a set of compiler directives
- The central entity in an OpenMP program is not a process but a thread.
- Threads are also called "lightweight processes" because several of them can share a common address space and mutually access data. Spawning a thread is much less costly than forking a new process, because threads share everything but instruction pointer (the address of the next instruction to be executed), stack pointer and register state.

Parallel Execution

- Master thread - runs immediately after startup
- Parallel execution happens inside parallel regions
- Between two parallel regions, no thread except the master thread executes any code. This is also called the "fork-join model"



Parallel Execution

```
  #include <omp.h>

  std::cout << "I am the master, and I am alone";
#pragma omp parallel
{
  do_work_package(omp_get_thread_num(),omp_get_num_threads());
}
```

Data Scoping

True work sharing, makes sense only if each thread can have its own, private variables. OpenMP supports this concept by defining a separate stack for every thread. There are three ways to make private variables:

- A variable that exists before entry to a parallel construct can be privatized, i.e.,made available as a private instance for every thread, by a PRIVATE clause to the OMP PARALLEL directive. The private variable's scope extends until the end of the parallel construct.
- The index variable of a worksharing loop is automatically made private
- Local variables in a subroutine called from a parallel region are private to each calling thread. This pertains also to copies of actual arguments generated by the call-by-value semantics

Private Scope

```
  integer :: bstart, bend, blen, numth, tid, i
  integer :: N
  double precision, dimension(N) :: a,b,c
  ...
!$OMP PARALLEL PRIVATE(bstart,bend,blen,numth,tid,i)
  numth = omp_get_num_threads()
  tid = omp_get_thread_num()
  blen = N/numth
  if(tid.lt.mod(N,numth)) then
    blen = blen + 1
    bstart = blen * tid + 1
  else
    bstart = blen * tid + mod(N,numth) + 1
  endif
  bend = bstart + blen - 1
  do i = bstart,bend
    a(i) = b(i) + c(i)
  enddo
!$OMP END PARALLEL
```

Private Scope

PRIVATE clause to the PARALLEL directive privatizes all specified variables, i.e., each thread gets its own instance of each variable on its local stack, with an undefined initial value (C++ objects will be instantiated using the default constructor). Using FIRSTPRIVATE instead of PRIVATE would initialize the privatize instances with the contents of the shared instance (in C++, the copy constructor is employed).

Worksharing for Loops

- A DO directive in front of a do loop starts a worksharing construct.
- The iterations of the loop are distributed among the threads (which are running because we are in a parallel region). Each thread gets its own iteration space, i.e., is assigned to a different set of i values.
- How threads are mapped to iterations is implementation-dependent by default, but can be influenced by the programmer

Synchronization

- Concurrent write access to a shared variable or, in more general terms, a shared resource, must be avoided by all means to circumvent race conditions.
- Critical regions solve this problem by making sure that at most one thread at a time executes some piece of code.
- If a thread is executing code inside a critical region, and another thread wants to enter, the latter must wait (block) until the former has left the region.
- Critical regions hold the danger of deadlocks when used inappropriately. A deadlock arises when one or more "agents" (threads in this case) wait for resources that will never become available.
- A critical region may be given a name that distinguishes it from others.

Barrier

- The barrier is a synchronization point, which guarantees that all threads have reached it before any thread goes on executing the code below it.
- Barriers should be used with caution in OpenMP programs, partly because of their potential to cause deadlocks, but also due to their performance impact (synchronization is overhead).
- Every parallel region executes an implicit barrier at its end, which cannot be removed.
- There is also a default implicit barrier at the end of worksharing loops and some other constructs to prevent race conditions.

The End