• The **PARALLEL DO** and **PARALLEL SECTIONS** constructs are in fact simplifications of a more complicated parallel construction capability.

• The **PARALLEL/END PARALLEL** pair is the main parallelism initiator.

• When **PARALLEL** is encountered it creates a team of threads that are to cooperate on the code that they all receive and via other parallelism specifications. (Whether or not the threads are newly created or simply reactivated is an implementation detail that can affect performance.)

• When **END PARALLEL** is encountered the thread team is disbanded and sequential execution resumes.
• If `PARALLEL/END PARALLEL` constructs are nested, a second team is formed to support nested parallelism. Currently, the second level is run with a team of one thread, i.e., the lower levels are sequentialized. But the standard allows nested parallelism.

• There are several constructs that allow the threads started to share work specified in the code each receives, to suppress sharing the work and to synchronize the work.
!$OMP PARALLEL DEFAULT(SHARED), PRIVATE(i,iam)

    iam = OMP_GET_NUM_THREAD()
    do i=1,m
       call local(iam,x,n,m)
    end do

!$OMP DO SCHEDULE(RUNTIME), PRIVATE(j)

    do j=1,n
       call work(j,x,n)
    end do

!$OMP END DO
!$OMP SECTIONS

!$OMP SECTION
  call xderiv(x,iam)
!$OMP SECTION
  call yderiv(x,iam)

!$OMP END SECTIONS

!$OMP SINGLE
  call output(x,n)
  call input(y,n)
!$OMP END SINGLE

  call work(iam,y,n)

!$OMP END PARALLEL
• The **PARALLEL** construct starts the threads and sends each a copy of the code specified between the **PARALLEL/END PARALLEL** pair.

• The first do loop runs sequentially on each thread, i.e., each thread runs all m iterations on local data.

• The **DO/END DO** pair specifies a loop whose work is shared by all threads. It has all of the clauses discussed earlier.

• There is no implied barrier at the beginning of the **DO/END DO**. All threads must encounter it but they may enter at different times unless explicitly told otherwise.
• There is an implied barrier at the end of the **DO/END DO**. Threads will wait there until all threads have reached the **END DO** unless explicitly told otherwise.

• The **PARALLEL DO/END PARALLEL DO** pair is the same as a **PARALLEL /END PARALLEL** with a single **DO/END DO**
• The **SECTIONS/END SECTIONS** defines code sections that are each run by a (possibly distinct) thread.

• The clauses are as before with the **PARALLEL SECTIONS / END PARALLEL SECTIONS** construct.

• No barrier is implied at the beginning of the **PARALLEL SECTIONS** unless explicitly stated otherwise.
- A barrier is implied at the **END PARALLEL SECTIONS** unless explicitly stated otherwise.

- The **PARALLEL SECTIONS/END PARALLEL SECTIONS** pair is the same as a **PARALLEL /END PARALLEL** with a single **SECTIONS/END SECTIONS**
• The **SINGLE/END SINGLE** pair defines a section of code that *only one thread from the team executes*.  

• Here it is used to do I/O assuming reentrant I/O routines.  

• You do not know which thread runs this code unless you explicitly test the thread number.  

• No barrier is implied at the beginning of the **SINGLE** unless explicitly stated otherwise.
• A barrier is implied at the **END SINGLE** unless explicitly stated otherwise.

• As with the other worksharing constructs, all threads must reach this section.

• Those that do not execute the code go immediately to the barrier at the **END SINGLE** and wait for all other threads.
• Note that the function of the `SINGLE/END SINGLE` pair is more restrictive than a critical section.

• After the `END SINGLE` barrier in the example the threads continue with the code sent to each by the `PARALLEL` construct.

• In this case, all threads then execute a call to the work routine.

• The `END PARALLEL` has an implied barrier.

• All threads must encounter all worksharing directives within a parallel region in the same order.
• Note that most of these constructs have implied barriers at the end directive.

• This can be overcome by a **NO WAIT** clause.

• Suppose we have two independent parallel loops. Work sharing on the second can begin for threads who have completed their part of the first loop before the others.
 !$OMP PARALLEL DEFAULT(SHARED)
 !$OMP DO SCHEDULE(RUNTIME), PRIVATE(j)
     do j=1,n
       b(j) = a(j)
     end do
 !$OMP END DO NO WAIT
 !$OMP DO SCHEDULE(RUNTIME), PRIVATE(j)
     do j=1,n
       z(j) = c(j)
     end do
 !$OMP END DO
 !$OMP END PARALLEL
• The **SINGLE/END SINGLE** pair is a quite restrictive form of synchronization.

• A slightly modified version allows the selection of a particular thread to execute the code in the block.

• The **MASTER/END MASTER** pair implies only the master executes the block of code. No entry or exit barrier is implied.
• The simplest synchronization in OPENMP is the **BARRIER**

• This can be used to enforce a barrier across all threads in the team executing a parallel region.

• It is also useful to enforce a barrier at the entry to the work sharing constructs that only have an implied barrier on exit.
!$OMP PARALLEL DEFAULT(SHARED), PRIVATE(i,iam)
   iam = OMP_GET_NUM_THREAD()
   do i=1,m
      call local(iam,x,n,m)
   end do
!$OMP BARRIER
!$OMP DO SCHEDULE(RUNTIME), PRIVATE(j)
   do j=1,n
      call work(j,x,n)
   end do
!$OMP END DO
!$OMP END PARALLEL
• There is also support for a critical section.

• This can be used in any parallel setting.

• The code in the section can only be executed by a single thread at any given time.

• All threads may pass through the section as opposed to the SINGLE construct which is a critical section that only one thread is allowed to enter.

• The following has two critical sections.

• Notice the name associated with each.
!$OMP PARALLEL DEFAULT(SHARED),PRIVATE(iam,sumx,sumy)
iam = OMP_GET_NUM_THREAD()
sumx= call local(iam,x,n)
sumy= call local(iam,y,n)

!$OMP CRITICAL (XSUM)
totalx = totalx + sumx
!$OMP END CRITICAL (XSUM)

!$OMP CRITICAL (YSUM)
totaly = totaly + sumy
!$OMP END CRITICAL (YSUM)

!$OMP END PARALLEL
• If the critical section is simply an update that needs to be atomic in its application the **ATOMIC** directive can be used.

• It only applies to the next statement so `sumy` is not protected in the following example.
 !$OMP PARALLEL DO
 do i =1,n
  sumx= call local(i,x,n)
  sumy= call local(i,y,n)
 !$OMP ATOMIC
  totalx = totalx + sumx
  totaly = totaly + sumy
 end do
 !$OMP END PARALLEL DO
- There is support for locks in order to allow a user to implement synchronization primitives.

- **man omp_lock** contains the details

- Lock variables must be initialized explicitly using **OMP_INIT_LOCK**

- They can be locked and unlocked as shown.

- Locking a locked lock will cause the thread to wait until it is given the lock.

- **OMP_TEST_LOCK** allows the routine to return when encountering a locked lock. If the lock was successfully locked then it behaves as **OMP_SET_LOCK**.

- A lock variable can be liberated from use as a lock via **OMP_DESTROY_LOCK**.
PROGRAM LOCK_USAGE
EXTERNAL OMP_TEST_LOCK
LOGICAL OMP_TEST_LOCK
INTEGER*8 LCK ! THIS VARIABLE SHOULD BE POINTER
! SIZED. 64 BIT ADDRESSES
! ARE ASSUMED HERE

CALL OMP_INIT_LOCK(LCK)
!$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
ID = OMP_GET_THREAD_NUM()
CALL OMP_SET_LOCK(LCK)
PRINT *, 'MY THREAD ID IS ', ID
CALL OMP_UNSET_LOCK(LCK)

DO WHILE (.NOT. OMP_TEST_LOCK(LCK))
   CALL SKIP(ID) ! WE DO NOT YET HAVE THE LOCK
   ! SO WE MUST DO SOMETHING ELSE
END DO

CALL WORK(ID) ! WE NOW HAVE THE LOCK
! AND CAN DO THE WORK
CALL OMP_UNSET_LOCK( LCK )
!$OMP END PARALLEL

CALL OMP_DESTROY_LOCK( LCK )
END