When parallelism is encountered, the thread team is disbanded and sequential execution resumes.

- When \texttt{END PARALLEL} is encountered, the thread team is
  implementation detail that can affect performance.

threads are newly created or simply reactivated is an
and via other parallelism specifications. Whether or not the
threads that are to cooperate on the code that they all receive
- When parallelism is encountered, it creates a team of
  parallelism initiator.

The parallel construction capability
constructs are in fact simplifications of a more complicated

- The parallel do and parallel sections
Sharing the work and to synchronize the work.

There are several constructs that allow the threads started to share work specified in the code each receives, to suppress allows nested parallelism.

Currently, the second level is run with a thread of one thread, second team is formed to support nested parallelism.

If PARALLEL END PARALLEL
!!$OMP END DO
!
end do
!
do j=1,n
!
call work(j,x,n)
!
!!$OMP END DO
!
end do
!
do i=1,m
!
call local(iam,x,n,m)
!
iam = OMP_GET_THREAD_NUM()
!
!!$OMP PARALLEL DEFAULT(SHARED), PRIVATE(i,iam)
!
end do
$OMP END PARALLEL
call work(iam,y,n)

$OMP END PARALLEL
call input(x,n)

$OMP SINGLE

call output(x,n)

$OMP END PARALLEL
call yderiv(x,iam)

$OMP SECTION

call xderiv(x,iam)

$OMP END PARALLEL
different times unless explicitly told otherwise.

DO. All threads must encounter it but they may enter at

DO/END. There is no implicit barrier at the beginning of the DO/END

END/DO pair. If has all of the clauses discussed earlier.

END/DO. The DO/END DO pair specifies a loop whose work is shared

END. Each thread runs all iterations on local data.

DO. The first do loop runs sequentially on each thread, i.e., each

PARALLEL. A copy of the code specified between the PARALLEL/END

PARALLEL-END. The PARALLEL construct starts the threads and sends each
DO/END DO
same as a PARALLEL END PARALLEL pair is the PARALLEL DO/END PARALLEL DO.

• END DO unless explicitly told otherwise.

• Threads will wait there until all threads have reached the DO/END DO.

• There is an implicit barrier at the end of the DO/END DO.
SECTIONS unless explicitly stated otherwise.

- No barrier is implied at the beginning of the PARALLEL SECTIONS construct.

- The clauses are as before with the PARALLEL SECTIONS that are each run by a (possibly distinct) thread.

- The SECTIONS/END SECTIONS defines code sections
The parallel sections pair is the same as a parallel/endpoint parallel sections pair, unless explicitly stated otherwise.

A barrier is implied at the end of parallel sections.
explicitly stated otherwise.

No barrier is implied at the beginning of the SINGLE

• explicitly test the thread number.

You do not know which thread runs this code unless you

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

that only one thread from the team executes.

• The SINGLE/END SINGLE pair defines a section of code
barrier at the **END SINGLE** and wait for all other threads.

- Those that do not execute the code go immediately to the
  reach this section.

As with the other worksheeting constructs, all threads must
stated otherwise.

A barrier is implied at the **END SINGLE** unless explicitly
parallel region in the same order.

• All threads must encounter all worksharing directives within a
  parallel region and all parallel regions have an implicit barrier.

• In this case, all threads then execute a call to the work routine,
  continue with the code sent to each by the PARALLEL
  construct.

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

• Note that the function of the SINGLE/END SINGLE pair
  is more restrictive than a critical section.
part of the first loop before the others.

Suppose we have two independent parallel loops. Work sharing

This can be overcome by a NO WAIT clause.

end directive.

Note that most of these constructs have implicit barriers at the
parallel i

enddo

default xr runtime, private(f)

enddo

default xr runtime, private(f)

enddo

default xr runtime, private(f)

enddo

enddo

default xr runtime, private(f)

enddo

default xr runtime, private(f)

enddo

default xr runtime, private(f)

enddo

default xr runtime, private(f)

enddo

default xr runtime, private(f)

enddo

Impiéd.

master executes the block of code. No entity or exits barrier is

the master/END MASTER pair implies only the

thread to execute the code in the block.

A slightly modified version allows the selection of a particular

of synchronization.

the SINGLE/END SINGLE pair is a quite restrictive form
Sharing constructs that only have an implied barrier on exit.
- It is also useful to enforce a barrier at the entry to the work
- Team executing a parallel region.
- This can be used to enforce a barrier across all threads in the
  The simplest synchronization in OpenMP is the BARRIER
```c
i$omp end parallel
i$omp end do
   end do
   call work(j, x, n)
   do j=1, n
       i$omp do schedule(runtime), private(f)
       i$omp barrier
       end do
   call local(ia[m], x, n, m)
   do i=1, m
       ia[m] = omp_get_thread_num()
   end do
i$omp parallel default(shared), private(i, ia[m])
```
Notice the name associated with each.

The following has two critical sections.

Thread is allowed to enter.

SINGLE construct which is a critical section that only one
All threads may pass through the section as opposed to the
Thread at any given time.

The code in the section can only be executed by a single
This can be used in any parallel setting.

There is also support for a critical section.
OMP END PARALLEL

(Y$OM)

OMP END CRITICAL (Y$OM)

Y$OM sumy = totaly + sumy

(OMP CRITICAL (Y$OM)

(OMP END CRITICAL (X$OM)

X$OM sumx = totalx + sumx

(OMP CRITICAL (X$OM)

Y$OM sumy = call local (jam, y'n)

(OMP CALL local (jam, x'n)

(omp = omp-get-thread-num)

(omp PARALLEL DEFAULT (SHARED), PRIVATE(jam, sumx, sumy)
In the following example:
- It only applies to the next statement so simply is not protected.

atomic in its application the \texttt{ATOMIC} directive can be used.
- If the critical section is simply an update that needs to be
0$OMP END PARALLEL DO
end do

\[ \text{sumy} = \text{totaly} + \text{sumy} \]
\[ \text{sumx} = \text{totalx} + \text{sumx} \]

$OMP ATOMIC

( \text{sumy} = \text{call local}(i',y',n) )
( \text{sumx} = \text{call local}(i',x',n) )

do i = 1,n

$OMP PARALLEL DO
OMP DESTROY LOCK.

A lock variable can be liberated from use as a lock via
then it behaves as OMP SET LOCK.
encountering a locked lock. If the lock was successfully locked
OMP TEST LOCK allows the routine to return when
Given the lock.

Locking a locked lock will cause the thread to wait until it is
They can be locked and unlocked as shown.

OMP INIT LOCK

Lock variables must be initialization explicitly. using
man omp-lock contains the details
Implementation synchronization primitives.
There is support for locks in order to allow a user to

OMP DESTROY LOCK.

A lock variable can be liberated from use as a lock via
then it behaves as OMP SET LOCK.
encountering a locked lock. If the lock was successfully locked
OMP TEST LOCK allows the routine to return when
Given the lock.

Locking a locked lock will cause the thread to wait until it is
They can be locked and unlocked as shown.

OMP INIT LOCK

Lock variables must be initialization explicitly. using
man omp-lock contains the details
Implementation synchronization primitives.
There is support for locks in order to allow a user to
CALL OMP-UNLOCK-LOCK(CK)
PRINT *, 'MY THREAD ID IS ', ID
CALL OMP-SET-LOCK(CK)
ID = OMP-GET-THREAD-NUM()
!$OMP PARALLEL SHARED(CK) PRIVATE(ID)
CALL OMP-INIT-LOCK(CK)

ARE ASSUMED HERE

SIZED 64 BIT ADDRESSES
INTEGER*8 CK ! THIS VARIABLE SHOULD BE POINTER
LOGICAL OMP-TEST-LOCK
EXTERNAL OMP-TEST-LOCK
PROGRAM LOCK-USAGE
END

CALL OMP-DESTROY-LOCK (LCK)

 !$OMP END PARALLEL

CALL OMP-UNSET-LOCK (LCK)

; i AND CAN DO THE WORK
CALL WORK (ID)

; i WE NOW HAVE THE LOCK

END DO

; i SO WE MUST DO SOMETHING ELSE
CALL SKIP (ID) ; i WE DO NOT YET HAVE THE LOCK

DO WHILE (! NOT . OMP-TEST-LOCK (LCK))