This page intentionally left blank.
Introduction

This collection of programming examples supplements the OpenMP API for Shared Memory Parallelization specifications, and is not part of the formal specifications. It assumes familiarity with the OpenMP specifications, and shares the typographical conventions used in that document.

Note – This initial release of the OpenMP Examples reflects the OpenMP Version 3.2 specifications and is a copy of the 3.2 Appendix A. A subsequent release that adds examples corresponding to the new features in the OpenMP Version 4.0 specifications is in progress.

The OpenMP API specification provides a model for parallel programming that is portable across shared memory architectures from different vendors. Compilers from numerous vendors support the OpenMP API.

The directives, library routines, and environment variables demonstrated in this document allow users to create and manage parallel programs while permitting portability. The directives extend the C, C++ and Fortran base languages with single program multiple data (SPMD) constructs, tasking constructs, device constructs, worksharing constructs, and synchronization constructs, and they provide support for sharing and privatizing data. The functionality to control the runtime environment is provided by library routines and environment variables. Compilers that support the OpenMP API often include a command line option to the compiler that activates and allows interpretation of all OpenMP directives.

Complete information about the OpenMP API and a list of the compilers that support the OpenMP API can be found at the OpenMP.org web site

http://www.openmp.org
This page is intentionally blank.
Examples

The following are examples of the OpenMP API directives, constructs, and routines.

A statement following a directive is compound only when necessary, and a non-compound statement is indented with respect to a directive preceding it.

1 A Simple Parallel Loop

The following example demonstrates how to parallelize a simple loop using the parallel loop construct. The loop iteration variable is private by default, so it is not necessary to specify it explicitly in a `private` clause.

Example 1.1c

```c
void simple(int n, float *a, float *b)
{
    int i;

    #pragma omp parallel for
    for (i=1; i<n; i++) /* i is private by default */
        b[i] = (a[i] + a[i-1]) / 2.0;
}
```
Example 1.1f

SUBROUTINE SIMPLE(N, A, B)
  INTEGER I, N
  REAL B(N), A(N)
  !$OMP PARALLEL DO !I is private by default
  DO I=2,N
    B(I) = (A(I) + A(I-1)) / 2.0
  ENDDO
  !$OMP END PARALLEL DO
END SUBROUTINE SIMPLE
2

The OpenMP Memory Model

In the following example, at Print 1, the value of \( x \) could be either 2 or 5, depending on the timing of the threads, and the implementation of the assignment to \( x \). There are two reasons that the value at Print 1 might not be 5. First, Print 1 might be executed before the assignment to \( x \) is executed. Second, even if Print 1 is executed after the assignment, the value 5 is not guaranteed to be seen by thread 1 because a flush may not have been executed by thread 0 since the assignment.

The barrier after Print 1 contains implicit flushes on all threads, as well as a thread synchronization, so the programmer is guaranteed that the value 5 will be printed by both Print 2 and Print 3.

Example 2.1c

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

int main(){
    int x;

    x = 2;
    #pragma omp parallel num_threads(2) shared(x)
    {
        if (omp_get_thread_num() == 0) {
            x = 5;
        } else {
            /* Print 1: the following read of x has a race */
            printf("1: Thread% d: x = %d\n", omp_get_thread_num(),x);
        }

        /*pragma omp barrier*/

        if (omp_get_thread_num() == 0) {
            /* Print 2 */
            printf("2: Thread% d: x = %d\n", omp_get_thread_num(),x);
        } else {
            /* Print 3 */
            printf("3: Thread% d: x = %d\n", omp_get_thread_num(),x);
        }
    }
    return 0;
}
```
Example 2.1f

```
PROGRAM MEMMODEL
  INCLUDE "omp_lib.h"      ! or USE OMP_LIB
  INTEGER X

  X = 2
  !$OMP PARALLEL NUM_THREADS(2) SHARED(X)
  IF (OMP_GET_THREAD_NUM() .EQ. 0) THEN
    X = 5
  ELSE
    ! PRINT 1: The following read of x has a race
    PRINT *,"1: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
  ENDIF

  !$OMP BARRIER
  IF (OMP_GET_THREAD_NUM() .EQ. 0) THEN
    ! PRINT 2
    PRINT *,"2: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
  ELSE
    ! PRINT 3
    PRINT *,"3: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
  ENDIF

  !$OMP END PARALLEL
END PROGRAM MEMMODEL
```

The following example demonstrates why synchronization is difficult to perform correctly through variables. The value of flag is undefined in both prints on thread 1 and the value of data is only well-defined in the second print.
Example 2.2c

```c
#include <omp.h>
#include <stdio.h>
int main()
{
    int data;
    int flag=0;
    #pragma omp parallel num_threads(2)
    {
        if (omp_get_thread_num()==0)
        {
            /* Write to the data buffer that will be
               read by thread */
            data = 42;
            /* Flush data to thread 1 and strictly order
               the write to data relative to the write to the flag */
            #pragma omp flush(flag, data)
            /* Set flag to release thread 1 */
            flag = 1;
            /* Flush flag to ensure that thread 1 sees
               the change */
            #pragma omp flush(flag)
        }
        else if(omp_get_thread_num()==1)
        {
            /* Loop until we see the update to the flag */
            #pragma omp flush(flag, data)
            while (flag < 1)
            {
                #pragma omp flush(flag, data)
            }
            /* Values of flag and data are undefined */
            printf("flag=%d data=%d\n", flag, data);
            #pragma omp flush(flag, data)
            /* Values data will be 42, value of flag
               still undefined */
            printf("flag=%d data=%d\n", flag, data);
        }
    }
    return 0;
}
```
Example 2.2f

```fortran
PROGRAM EXAMPLE
INCLUDE "omp_lib.h" ! or USE OMP_LIB
INTEGER DATA
INTEGER FLAG

FLAG = 0
!$OMP PARALLEL NUM_THREADS(2)
  IF(OMP_GET_THREAD_NUM() .EQ. 0) THEN
    ! Write to the data buffer that will be read by thread 1
    DATA = 42
    ! Flush DATA to thread 1 and strictly order the write to DATA
    ! relative to the write to the FLAG
    !$OMP FLUSH(FLAG, DATA)
    ! Set FLAG to release thread 1
    FLAG = 1;
    ! Flush FLAG to ensure that thread 1 sees the change */
    !$OMP FLUSH(FLAG)
  ELSE IF(OMP_GET_THREAD_NUM() .EQ. 1) THEN
    ! Loop until we see the update to the FLAG
    !$OMP FLUSH(FLAG, DATA)
    DO WHILE(FLAG .LT. 1)
      !$OMP FLUSH(FLAG, DATA)
    ENDDO
    ! Values of FLAG and DATA are undefined
    PRINT *, 'FLAG=', FLAG, ' DATA=', DATA
    !$OMP FLUSH(FLAG, DATA)
  ENDIF
!$OMP END PARALLEL
END
```

The next example demonstrates why synchronization is difficult to perform correctly through variables. Because the write(1)-flush(1)-flush(2)-read(2) sequence cannot be guaranteed in the example, the statements on thread 0 and thread 1 may execute in either order.
Example 2.3c

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

int main()
{
    int flag = 0;

    #pragma omp parallel num_threads(3)
    {
        if(omp_get_thread_num()==0)
        {
            /* Set flag to release thread 1 */
            #pragma omp atomic update
            flag++;
            /* Flush of flag is implied by the atomic directive */
        }
        else if(omp_get_thread_num()==1)
        {
            /* Loop until we see that flag reaches 1*/
            #pragma omp flush(flag)
            while(flag < 1)
            {
                #pragma omp flush(flag)
            }
            printf("Thread 1 awoken\n");
            /* Set flag to release thread 2 */
            #pragma omp atomic update
            flag++;
            /* Flush of flag is implied by the atomic directive */
        }
        else if(omp_get_thread_num()==2)
        {
            /* Loop until we see that flag reaches 2 */
            #pragma omp flush(flag)
            while(flag < 2)
            {
                #pragma omp flush(flag)
            }
            printf("Thread 2 awoken\n");
        }
    }
    return 0;
}
```
Example 2.3f

PROGRAM EXAMPLE
INCLUDE "omp_lib.h" ! or USE OMP_LIB
INTEGER FLAG

FLAG = 0
!$OMP PARALLEL NUM_THREADS(3)
IF(OMP_GET_THREAD_NUM() .EQ. 0) THEN
   ! Set flag to release thread 1
   !$OMP ATOMIC UPDATE
   FLAG = FLAG + 1
   !Flush of FLAG is implied by the atomic directive
ELSE IF(OMP_GET_THREAD_NUM() .EQ. 1) THEN
   ! Loop until we see that FLAG reaches 1
   !$OMP FLUSH(FLAG, DATA)
   DO WHILE(FLAG .LT. 1)
      !$OMP FLUSH(FLAG, DATA)
   ENDDO
   PRINT *, 'Thread 1 awoken'
   ! Set FLAG to release thread 2
   !$OMP ATOMIC UPDATE
   FLAG = FLAG + 1
   !Flush of FLAG is implied by the atomic directive
ELSE IF(OMP_GET_THREAD_NUM() .EQ. 2) THEN
   ! Loop until we see that FLAG reaches 2
   !$OMP FLUSH(FLAG, DATA)
   DO WHILE(FLAG .LT. 2)
      !$OMP FLUSH(FLAG, DATA)
   ENDDO
   PRINT *, 'Thread 2 awoken'
ENDIF
!$OMP END PARALLEL
END
3 Conditional Compilation

The following example illustrates the use of conditional compilation using the OpenMP macro `_OPENMP`. With OpenMP compilation, the `_OPENMP` macro becomes defined.

Example 3.1c

```c
#include <stdio.h>

int main()
{
    #ifdef _OPENMP
        printf("Compiled by an OpenMP-compliant implementation.\n");
    #endif
    return 0;
}
```

Fortran

The following example illustrates the use of the conditional compilation sentinel. With OpenMP compilation, the conditional compilation sentinel `!$` is recognized and treated as two spaces. In fixed form source, statements guarded by the sentinel must start after column 6.

Example 3.1f

```fortran
PROGRAM EXAMPLE
C234567890
!$    PRINT *, "Compiled by an OpenMP-compliant implementation."
END PROGRAM EXAMPLE
```
4 Internal Control Variables (ICVs)

According to $, an OpenMP implementation must act as if there are ICVs that control the behavior of the program. This example illustrates two ICVs, nthreads-var and max-active-levels-var. The nthreads-var ICV controls the number of threads requested for encountered parallel regions; there is one copy of this ICV per task. The max-active-levels-var ICV controls the maximum number of nested active parallel regions; there is one copy of this ICV for the whole program.

In the following example, the nest-var, max-active-levels-var, dyn-var, and nthreads-var ICVs are modified through calls to the runtime library routines omp_set_nested, omp_set_max_active_levels, omp_set_dynamic, and omp_set_num_threads respectively. These ICVs affect the operation of parallel regions. Each implicit task generated by a parallel region has its own copy of the nest-var, dyn-var, and nthreads-var ICVs.

In the following example, the new value of nthreads-var applies only to the implicit tasks that execute the call to omp_set_num_threads. There is one copy of the max-active-levels-var ICV for the whole program and its value is the same for all tasks. This example assumes that nested parallelism is supported.

The outer parallel region creates a team of two threads; each of the threads will execute one of the two implicit tasks generated by the outer parallel region.

Each implicit task generated by the outer parallel region calls omp_set_num_threads(3), assigning the value 3 to its respective copy of nthreads-var. Then each implicit task encounters an inner parallel region that creates a team of three threads; each of the threads will execute one of the three implicit tasks generated by that inner parallel region.

Since the outer parallel region is executed by 2 threads, and the inner by 3, there will be a total of 6 implicit tasks generated by the two inner parallel regions.

Each implicit task generated by an inner parallel region will execute the call to omp_set_num_threads(4), assigning the value 4 to its respective copy of nthreads-var.

The print statement in the outer parallel region is executed by only one of the threads in the team. So it will be executed only once.

The print statement in an inner parallel region is also executed by only one of the threads in the team. Since we have a total of two inner parallel regions, the print statement will be executed twice -- once per inner parallel region.
Example 4.1c

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

int main (void)
{
   omp_set_nested(1);
   omp_set_max_active_levels(8);
   omp_set_dynamic(0);
   omp_set_num_threads(2);
    #pragma omp parallel
    {
        omp_set_num_threads(3);

        #pragma omp parallel
        {
            omp_set_num_threads(4);
            #pragma omp single
            {
                /*
                 * The following should print:
                 * Inner: max_act_lev=8, num_thds=3, max_thds=4
                 * Inner: max_act_lev=8, num_thds=3, max_thds=4
                 */
                printf("Inner: max_act_lev=%d, num_thds=%d, max_thds=%d\n", 
                       omp_get_max_active_levels(), omp_get_num_threads(), 
                       omp_get_max_threads());
            }
        }

        #pragma omp barrier
        #pragma omp single
        {
            /*
             * The following should print:
             * Outer: max_act_lev=8, num_thds=2, max_thds=3
             */
            printf("Outer: max_act_lev=%d, num_thds=%d, max_thds=%d\n", 
                   omp_get_max_active_levels(), omp_get_num_threads(), 
                   omp_get_max_threads());
        }
    }
    return 0;
}
```
Example 4.1f

```fortran
program icv
  use omp_lib

  call omp_set_nested(.true.)
  call omp_set_max_active_levels(8)
  call omp_set_dynamic(.false.)
  call omp_set_num_threads(2)

  !$omp parallel
  call omp_set_num_threads(3)

  !$omp parallel
    call omp_set_num_threads(4)
  !$omp single
    ! The following should print:
    ! Inner: max_act_lev= 8 , num_thds= 3 , max_thds= 4
    ! Inner: max_act_lev= 8 , num_thds= 3 , max_thds= 4
    print *, "Inner: max_act_lev=", omp_get_max_active_levels(),
    & " , num_thds=", omp_get_num_threads(),
    & " , max_thds=", omp_get_max_threads()
  !$omp end single
  !$omp end parallel

  !$omp barrier
  !$omp single
    ! The following should print:
    ! Outer: max_act_lev= 8 , num_thds= 2 , max_thds= 3
    print *, "Outer: max_act_lev=", omp_get_max_active_levels(),
    & " , num_thds=", omp_get_num_threads(),
    & " , max_thds=", omp_get_max_threads()
  !$omp end single
  !$omp end parallel
end
```
The parallel Construct

The `parallel` construct can be used in coarse-grain parallel programs. In the following example, each thread in the `parallel` region decides what part of the global array `x` to work on, based on the thread number:

```c/c++
#include <omp.h>

void subdomain(float *x, int istart, int ipoints)
{
    int i;

    for (i = 0; i < ipoints; i++)
        x[istart+i] = 123.456;
}

void sub(float *x, int npoints)
{
    int iam, nt, ipoints, istart;

    #pragma omp parallel default(shared) private(iam,nt,ipoints,istart)
    {
        iam = omp_get_thread_num();
        nt = omp_get_num_threads();
        ipoints = npoints / nt; /* size of partition */
        istart = iam * ipoints; /* starting array index */
        if (iam == nt - 1) /* last thread may do more */
            ipoints = npoints - istart;
        subdomain(x, istart, ipoints);
    }

    int main()
    {
        float array[10000];
        sub(array, 10000);
        return 0;
    }
```
Example 5.1f

```fortran
SUBROUTINE SUBDOMAIN(X, ISTART, IPOINTS)
    INTEGER ISTART, IPOINTS
    REAL X(*)
    INTEGER I

    DO 100 I=1,IPOINTS
        X(ISTART+I) = 123.456
    100      CONTINUE

END SUBROUTINE SUBDOMAIN

SUBROUTINE SUB(X, NPOINTS)
    INCLUDE "omp_lib.h"     ! or USE OMP_LIB
    REAL X(*)
    INTEGER NPOINTS
    INTEGER IAM, NT, IPOINTS, ISTART

    !$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,NPOINTS)
    IAM = OMP_GET_THREAD_NUM()
    NT = OMP_GET_NUM_THREADS()
    IPOINTS = NPOINTS/NT
    ISTART = IAM * IPOINTS
    IF (IAM .EQ. NT-1) THEN
        IPOINTS = NPOINTS - ISTART
    ENDIF
    CALL SUBDOMAIN(X,ISTART,IPOINTS)
    !$OMP END PARALLEL

END SUBROUTINE SUB

PROGRAM PAREXAMPLE
    REAL ARRAY(10000)
    CALL SUB(ARRAY, 10000)
END PROGRAM PAREXAMPLE
```
6 Controlling the Number of Threads on Multiple Nesting Levels

The following examples demonstrate how to use the `OMP_NUM_THREADS` environment variable to control the number of threads on multiple nesting levels:

```
C/C++

Example 6.1c

#include <stdio.h>
#include <omp.h>
int main (void)
{
    omp_set_nested(1);
    omp_set_dynamic(0);
    #pragma omp parallel
    {
        #pragma omp parallel
        {
            #pragma omp single
            {
                /*
                 * If OMP_NUM_THREADS=2,3 was set, the following should print:
                 * Inner: num_thds=3
                 * Inner: num_thds=3
                 * If nesting is not supported, the following should print:
                 * Inner: num_thds=1
                 * Inner: num_thds=1
                 */
                printf("Inner: num_thds=%d\n", omp_get_num_threads());
            }
        }
        #pragma omp barrier
        omp_set_nested(0);
        #pragma omp parallel
        {
            #pragma omp single
            {
                /*
                 * Even if OMP_NUM_THREADS=2,3 was set, the following should print, because nesting is disabled:
                 * Inner: num_thds=1
                 * Inner: num_thds=1
                 */
                printf("Inner: num_thds=%d\n", omp_get_num_threads());
            }
        }
    }
}
```
```c
#include "omp.h"

int main()
{
    #pragma omp barrier
    #pragma omp single
    {
        /*
        * If OMP_NUM_THREADS=2,3 was set, the following should print:
        * Outer: num_thds=2
        */
        printf("Outer: num_thds=%d\n", omp_get_num_threads());
    }
    return 0;
}
```
Example 6.1f

program icv
    use omp_lib
    call omp_set_nested(.true.)
    call omp_set_dynamic(.false.)
    !$omp parallel
    !$omp parallel
    !$omp single
    ! If OMP_NUM_THREADS=2,3 was set, the following should print:
    ! Inner: num_thds= 3
    ! Inner: num_thds= 3
    ! If nesting is not supported, the following should print:
    ! Inner: num_thds= 1
    ! Inner: num_thds= 1
    print *, "Inner: num_thds=", omp_get_num_threads()
    !$omp end single
    !$omp end parallel
    !$omp barrier
    call omp_set_nested(.false.)
    !$omp parallel
    !$omp single
    ! Even if OMP_NUM_THREADS=2,3 was set, the following should print,
    ! because nesting is disabled:
    ! Inner: num_thds= 1
    ! Inner: num_thds= 1
    print *, "Inner: num_thds=", omp_get_num_threads()
    !$omp end single
    !$omp end parallel
    !$omp barrier
    !$omp single
    ! If OMP_NUM_THREADS=2,3 was set, the following should print:
    ! Outer: num_thds= 2
    print *, "Outer: num_thds=", omp_get_num_threads()
    !$omp end single
    !$omp end parallel
    end
Interaction Between the num_threads Clause and omp_set_dynamic

The following example demonstrates the num_threads clause and the effect of the omp_set_dynamic routine on it.

The call to the omp_set_dynamic routine with argument 0 in C/C++, or .FALSE. in Fortran, disables the dynamic adjustment of the number of threads in OpenMP implementations that support it. In this case, 10 threads are provided. Note that in case of an error the OpenMP implementation is free to abort the program or to supply any number of threads available.

C/C++

Example 7.1c

```c
#include <omp.h>
int main()
{
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(10)
    {
        /* do work here */
    }
    return 0;
}
```
**Example 7.1f**

```fortran
PROGRAM EXAMPLE
    INCLUDE "omp_lib.h" ! or USE OMP_LIB
    CALL OMP_SET_DYNAMIC(.FALSE.)
    !$OMP PARALLEL NUM_THREADS(10)
        ! do work here
    !$OMP END PARALLEL
END PROGRAM EXAMPLE
```

The call to the `omp_set_dynamic` routine with a non-zero argument in C/C++, or `.TRUE.` in Fortran, allows the OpenMP implementation to choose any number of threads between 1 and 10.

**Example 7.2c**

```c/c++
#include <omp.h>
int main()
{
    omp_set_dynamic(1);
    #pragma omp parallel num_threads(10)
    {
        /* do work here */
    }
    return 0;
}
```
Example 7.2f

```fortran
PROGRAM EXAMPLE
    INCLUDE "omp_lib.h" ! or USE OMP_LIB
    CALL OMP_SET_DYNAMIC(.TRUE.)
    !$OMP PARALLEL NUM_THREADS(10)
        ! do work here
    !$OMP END PARALLEL
END PROGRAM EXAMPLE
```

It is good practice to set the dyn-var ICV explicitly by calling the `omp_set_dynamic` routine, as its default setting is implementation defined.
8 Fortran Restrictions on the do Construct

If an `end do` directive follows a `do-construct` in which several DO statements share a DO termination statement, then a do directive can only be specified for the outermost of these DO statements. The following example contains correct usages of loop constructs:

Example 8.1f

```fortran
SUBROUTINE WORK(I, J)
  INTEGER I, J
END SUBROUTINE WORK

SUBROUTINE DO_GOOD()
  INTEGER I, J
  REAL A(1000)

  DO 100 I = 1, 10
     !$OMP DO
       DO 100 J = 1, 10
       CALL WORK(I, J)
     100 CONTINUE ! !$OMP ENDDO implied here

  !$OMP DO
    DO 200 J = 1, 10
    A(I) = I + 1
  200 !$OMP ENDDO

  !$OMP DO
    DO 300 I = 1, 10
      DO 300 J = 1, 10
      CALL WORK(I, J)
    300 CONTINUE
  300 !$OMP ENDDO
END SUBROUTINE DO_GOOD
```

The following example is non-conforming because the matching do directive for the end do does not precede the outermost loop:
Example 8.2f

```fortran
SUBROUTINE WORK(I, J)
  INTEGER I, J
END SUBROUTINE WORK

SUBROUTINE DO_WRONG
  INTEGER I, J

  DO 100 I = 1, 10
    !$OMP DO
      DO 100 J = 1, 10
        CALL WORK(I, J)
    100 CONTINUE
    !$OMP ENDDO
END SUBROUTINE DO_WRONG
```

Fortran
Fortran Private Loop Iteration Variables

In general loop iteration variables will be private, when used in the do-loop of a do and parallel do construct or in sequential loops in a parallel construct (see $ and $). In the following example of a sequential loop in a parallel construct the loop iteration variable $ will be private.

Example 9.1f

```fortran
SUBROUTINE PLOOP_1(A,N) 
  INCLUDE "omp_lib.h" ! or USE OMP_LIB 
  REAL A(*) 
  INTEGER I, MYOFFSET, N 
  !$OMP PARALLEL PRIVATE(MYOFFSET) 
    MYOFFSET = OMP_GET_THREAD_NUM()*N 
    DO I = 1, N 
      A(MYOFFSET+I) = FLOAT(I) 
    ENDDO 
  !$OMP END PARALLEL 
END SUBROUTINE PLOOP_1 
```

In exceptional cases, loop iteration variables can be made shared, as in the following example:

Example 9.2f

```fortran
SUBROUTINE PLOOP_2(A,B,N,I1,I2) 
  REAL A(*), B(*) 
  INTEGER I1, I2, N 
  !$OMP PARALLEL SHARED(A,B,I1,I2) 
  !$OMP SECTIONS 
  !$OMP SECTION 
    DO I1 = I1, N 
      IF (A(I1).NE.0.0) EXIT 
    ENDDO 
  !$OMP SECTION 
    DO I2 = I2, N 
      IF (B(I2).NE.0.0) EXIT 
    ENDDO 
  !$OMP END SECTIONS 
  !$OMP SINGLE 
  IF (I1.LE.N) PRINT *, 'ITEMS IN A UP TO ', I1, 'ARE ALL ZERO.' 
  IF (I2.LE.N) PRINT *, 'ITEMS IN B UP TO ', I2, 'ARE ALL ZERO.' 
  !$OMP END SINGLE 
```
Note however that the use of shared loop iteration variables can easily lead to race conditions.
10 The `nowait` clause

If there are multiple independent loops within a `parallel` region, you can use the `nowait` clause to avoid the implied barrier at the end of the loop construct, as follows:

```
#include <math.h>

void nowait_example(int n, int m, float *a, float *b, float *y, float *z)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (i=1; i<n; i++)
            b[i] = (a[i] + a[i-1]) / 2.0;

        #pragma omp for nowait
        for (i=0; i<m; i++)
            y[i] = sqrt(z[i]);
    }
}
```

Example 10.1c
Example 10.1f

```fortran
SUBROUTINE NOWAIT_EXAMPLE(N, M, A, B, Y, Z)

   INTEGER N, M
   REAL A(*), B(*), Y(*), Z(*)
   INTEGER I

   !$OMP PARALLEL
   !$OMP DO
   DO I=2,N
      B(I) = (A(I) + A(I-1)) / 2.0
   ENDDO
   !$OMP END DO NOWAIT
   !$OMP DO
   DO I=1,M
      Y(I) = SQRT(Z(I))
   ENDDO
   !$OMP END DO NOWAIT
   !$OMP END PARALLEL

END SUBROUTINE NOWAIT_EXAMPLE
```

In the following example, static scheduling distributes the same logical iteration numbers to the threads that execute the three loop regions. This allows the `nowait` clause to be used, even though there is a data dependence between the loops. The dependence is satisfied as long the same thread executes the same logical iteration numbers in each loop.

Note that the iteration count of the loops must be the same. The example satisfies this requirement, since the iteration space of the first two loops is from 0 to n-1 (from 1 to N in the Fortran version), while the iteration space of the last loop is from 1 to n (2 to N+1 in the Fortran version).

Example 10.2c

```c
#include <math.h>
void nowait_example2(int n, float *a, float *b, float *c, float *y, float *z)
{
   int i;
   #pragma omp parallel
   {
      #pragma omp for schedule(static) nowait
```
for (i=0; i<n; i++)
c[i] = (a[i] + b[i]) / 2.0f;
#pragma omp for schedule(static) nowait
for (i=0; i<n; i++)
z[i] = sqrtf(c[i]);
#pragma omp for schedule(static) nowait
for (i=1; i<=n; i++)
y[i] = z[i-1] + a[i];
}

Example 10.2f

SUBROUTINE NOWAIT_EXAMPLE2(N, A, B, C, Y, Z)
INTEGER N
REAL A(*), B(*) , C(*), Y(*), Z(*)
INTEGER I
!
|$OMP PARALLEL
|$OMP DO SCHEDULE(STATIC)
DO I=1,N
  C(I) = (A(I) + B(I)) / 2.0
ENDDO
|$OMP END DO NOWAIT
|$OMP DO SCHEDULE(STATIC)
DO I=1,N
  Z(I) = SQRT(C(I))
ENDDO
|$OMP END DO NOWAIT
|$OMP DO SCHEDULE(STATIC)
DO I=2,N+1
  Y(I) = Z(I-1) + A(I)
ENDDO
|$OMP END DO NOWAIT
|$OMP END PARALLEL
END SUBROUTINE NOWAIT_EXAMPLE2
The collapse clause

In the following example, the k and j loops are associated with the loop construct. So the iterations of the k and j loops are collapsed into one loop with a larger iteration space, and that loop is then divided among the threads in the current team. Since the i loop is not associated with the loop construct, it is not collapsed, and the i loop is executed sequentially in its entirety in every iteration of the collapsed k and j loop.

Example 11.1c

```c
void bar(float *a, int i, int j, int k);
int kl, ku, ks, jl, ju, js, il, iu,is;
void sub(float *a)
{
    int i, j, k;
    #pragma omp for collapse(2) private(i, k, j)
    for (k=kl; k<=ku; k+=ks)
        for (j=jl; j<=ju; j+=js)
            for (i=il; i<=iu; i+=is)
                bar(a,i,j,k);
}
```
Example 11.1f

subroutine sub(a)
    real a(*)
    integer kl, ku, ks, jl, ju, js, il, iu, is
    common /csub/ kl, ku, ks, jl, ju, js, il, iu, is
    integer i, j, k

    !$omp do collapse(2) private(i,j,k)
    do k = kl, ku, ks
        do j = jl, ju, js
            do i = il, iu, is
                call bar(a,i,j,k)
            enddo
        enddo
    enddo
    !$omp end do
end subroutine

In the next example, the $k$ and $j$ loops are associated with the loop construct. So the iterations of the $k$ and $j$ loops are collapsed into one loop with a larger iteration space, and that loop is then divided among the threads in the current team.

The sequential execution of the iterations in the $k$ and $j$ loops determines the order of the iterations in the collapsed iteration space. This implies that in the sequentially last iteration of the collapsed iteration space, $k$ will have the value 2 and $j$ will have the value 3. Since $k_{last}$ and $j_{last}$ are lastprivate, their values are assigned by the sequentially last iteration of the collapsed $k$ and $j$ loop. This example prints: 2 3.
Example 11.2c

```c
#include <stdio.h>
void test()
{
    int j, k, jlast, klast;
    #pragma omp parallel
    {
        #pragma omp for collapse(2) lastprivate(jlast, klast)
        for (k=1; k<=2; k++)
            for (j=1; j<=3; j++)
                {
                    jlast=j;
                    klast=k;
                }
        #pragma omp single
        printf("%d %d\n", klast, jlast);
    }
}
```
**Example 11.2f**

```fortran
program test
!$omp parallel
!$omp do private(j,k) collapse(2) lastprivate(jlast, klast)
   do k = 1,2
      do j = 1,3
         jlast=j
         klast=k
      enddo
   enddo
!$omp end do
!$omp single
   print *, klast, jlast
!$omp end single
!$omp end parallel
end program test
```

The next example illustrates the interaction of the `collapse` and `ordered` clauses. In the example, the loop construct has both a `collapse` clause and an `ordered` clause. The `collapse` clause causes the iterations of the `k` and `j` loops to be collapsed into one loop with a larger iteration space, and that loop is divided among the threads in the current team. An `ordered` clause is added to the loop construct, because an ordered region binds to the loop region arising from the loop construct.

According to $\$, a thread must not execute more than one ordered region that binds to the same loop region. So the `collapse` clause is required for the example to be conforming. With the `collapse` clause, the iterations of the `k` and `j` loops are collapsed into one loop, and therefore only one ordered region will bind to the collapsed `k` and `j` loop. Without the `collapse` clause, there would be two ordered regions that bind to each iteration of the `k` loop (one arising from the first iteration of the `j` loop, and the other arising from the second iteration of the `j` loop).

The code prints

```
0 1 1
0 1 2
0 2 1
1 2 2
1 3 1
1 3 2
```
Example 11.3c

```c
#include <omp.h>
#include <stdio.h>
void work(int a, int j, int k);
void sub()
{
    int j, k, a;
    #pragma omp parallel num_threads(2)
    {
        #pragma omp for collapse(2) ordered private(j,k) schedule(static,3)
        for (k=1; k<=3; k++)
            for (j=1; j<=2; j++)
                {
                    #pragma omp ordered
                    printf("%d %d %d
", omp_get_thread_num(), k, j);
                    /* end ordered */
                    work(a,j,k);
                }
    }
}
```

Example 11.3f

```fortran
program test
    include 'omp_lib.h'
    !$omp parallel num_threads(2)
    !$omp do collapse(2) ordered private(j,k) schedule(static,3)
    do k = 1,3
        do j = 1,2
            !$omp ordered
            print *, omp_get_thread_num(), k, j
            !$omp end ordered
            call work(a,j,k)
        enddo
    enddo
    !$omp end do
    !$omp end parallel
end program test
```
The parallel sections Construct

In the following example routines XAXIS, YAXIS, and ZAXIS can be executed concurrently. The first section directive is optional. Note that all section directives need to appear in the parallel sections construct.

**Example 12.1c**

```c/c++

void XAXIS();
void YAXIS();
void ZAXIS();

void sect_example()
{
  #pragma omp parallel sections
  {
    #pragma omp section
    XAXIS();
    #pragma omp section
    YAXIS();
    #pragma omp section
    ZAXIS();
  }
}
```

**Example 12.1f**

```fortran

SUBROUTINE SECT_EXAMPLE()

!$OMP PARALLEL SECTIONS
!$OMP SECTION
CALL XAXIS()
!$OMP SECTION
CALL YAXIS()
!$OMP SECTION
CALL ZAXIS()
!$OMP END PARALLEL SECTIONS
END SUBROUTINE SECT_EXAMPLE
```
13 The firstprivate Clause and the sections Construct

In the following example of the sections construct the firstprivate clause is used to initialize the private copy of section_count of each thread. The problem is that the section constructs modify section_count, which breaks the independence of the section constructs. When different threads execute each section, both sections will print the value 1. When the same thread executes the two sections, one section will print the value 1 and the other will print the value 2. Since the order of execution of the two sections in this case is unspecified, it is unspecified which section prints which value.

Example 13.1c

```c
#include <omp.h>
#include <stdio.h>
#define NT 4
int main( ) {
    int section_count = 0;
    omp_set_dynamic(0);
    omp_set_num_threads(NT);
    #pragma omp parallel
    #pragma omp sections firstprivate( section_count )
    {
        #pragma omp section
        {
            section_count++;
            /* may print the number one or two */
            printf( "section_count %d\n", section_count );
        }
        #pragma omp section
        {
            section_count++;
            /* may print the number one or two */
            printf( "section_count %d\n", section_count );
        }
        return 1;
    }
}
```
Example 13.1f

```fortran
program section
  use omp_lib
  integer :: section_count = 0
  integer, parameter :: NT = 4
  call omp_set_dynamic(.false.)
  call omp_set_num_threads(NT)
  !$omp parallel
  !$omp sections firstprivate ( section_count )
  !$omp section
    section_count = section_count + 1
    ! may print the number one or two
    print *, 'section_count', section_count
  !$omp section
    section_count = section_count + 1
    ! may print the number one or two
    print *, 'section_count', section_count
  !$omp end sections
  !$omp end parallel
end program section
```
The single Construct

The following example demonstrates the single construct. In the example, only one thread prints each of the progress messages. All other threads will skip the single region and stop at the barrier at the end of the single construct until all threads in the team have reached the barrier. If other threads can proceed without waiting for the thread executing the single region, a nowait clause can be specified, as is done in the third single construct in this example. The user must not make any assumptions as to which thread will execute a single region.

Example 14.1c

```c
#include <stdio.h>

void work1() {}
void work2() {}

void single_example()
{
    #pragma omp parallel
    {
        #pragma omp single
        printf("Beginning work1.\n");
        work1();

        #pragma omp single
        printf("Finishing work1.\n");

        #pragma omp single nowait
        printf("Finished work1 and beginning work2.\n");
        work2();
    }
}
```
Example 14.1f

```fortran
SUBROUTINE WORK1()
END SUBROUTINE WORK1

SUBROUTINE WORK2()
END SUBROUTINE WORK2

PROGRAM SINGLE_EXAMPLE
!$OMP PARALLEL

!$OMP SINGLE
print *, "Beginning work1."
!$OMP END SINGLE

CALL WORK1()

!$OMP SINGLE
print *, "Finishing work1."
!$OMP END SINGLE

!$OMP SINGLE
print *, "Finished work1 and beginning work2."
!$OMP END SINGLE NOWAIT

CALL WORK2()

!$OMP END PARALLEL

END PROGRAM SINGLE_EXAMPLE
```
15 Tasking Constructs

The following example shows how to traverse a tree-like structure using explicit tasks. Note that the traverse function should be called from within a parallel region for the different specified tasks to be executed in parallel. Also note that the tasks will be executed in no specified order because there are no synchronization directives. Thus, assuming that the traversal will be done in post order, as in the sequential code, is wrong.

**Example 15.1c**

```c
struct node {
    struct node *left;
    struct node *right;
};
extern void process(struct node *);

void traverse(struct node *p) {
    if (p->left)
        #pragma omp task // p is firstprivate by default
        traverse(p->left);
    if (p->right)
        #pragma omp task // p is firstprivate by default
        traverse(p->right);
    process(p);
}
```

**Example 15.1f**

```fortran
RECURSIVE SUBROUTINE traverse(P)
    TYPE(Node), POINTER :: left, right
END TYPE Node
    TYPE(Node) :: P
    IF (associated(P%left)) THEN
        !$OMP TASK // P is firstprivate by default
        call traverse(P%left)
        !$OMP END TASK
    ENDIF
    IF (associated(P%right)) THEN
        !$OMP TASK // P is firstprivate by default
        call traverse(P%right)
        !$OMP END TASK
ENDIF
```
In the next example, we force a postorder traversal of the tree by adding a `taskwait` directive. Now, we can safely assume that the left and right sons have been executed before we process the current node.

**Example 15.2c**

```c
struct node {
    struct node *left;
    struct node *right;
};
extern void process(struct node *);
void postorder_traverse( struct node *p ) {
    if (p->left)
        #pragma omp task    // p is firstprivate by default
        postorder_traverse(p->left);
    if (p->right)
        #pragma omp task    // p is firstprivate by default
        postorder_traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

**Example 15.2f**

```fortran
RECURSIVE SUBROUTINE traverse ( P )
    TYPE Node
        TYPE(Node), POINTER :: left, right
    END TYPE Node
    TYPE(Node) :: P
    IF (associated(P%left)) THEN
        !$OMP TASK   ! P is firstprivate by default
        call traverse(P%left)
        !$OMP END TASK
    ENDIF
    IF (associated(P%right)) THEN
        !$OMP TASK   ! P is firstprivate by default
        call traverse(P%right)
        !$OMP END TASK
    ENDIF
    !$OMP TASKWAIT
```

OpenMP  Examples  43
The following example demonstrates how to use the `task` construct to process elements of a linked list in parallel. The thread executing the `single` region generates all of the explicit tasks, which are then executed by the threads in the current team. The pointer `p` is `firstprivate` by default on the `task` construct so it is not necessary to specify it in a `firstprivate` clause.

```c
typedef struct node node;
struct node {
    int data;
    node * next;
};

void process(node * p)
{
    /* do work here */
}
void increment_list_items(node * head)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            node * p = head;
            while (p) {
                #pragma omp task
                // p is firstprivate by default
                process(p);
                p = p->next;
            }
        }
    }
}
```
Example 15.3f

Fortran

```fortran
MODULE LIST
  TYPE NODE
    INTEGER :: PAYLOAD
    TYPE (NODE), POINTER :: NEXT
  END TYPE NODE
CONTAINS
  SUBROUTINE PROCESS(p)
    TYPE (NODE), POINTER :: P
    ! do work here
  END SUBROUTINE
  SUBROUTINE INCREMENT_LIST_ITEMS (HEAD)
    TYPE (NODE), POINTER :: HEAD
    TYPE (NODE), POINTER :: P
    !$OMP PARALLEL PRIVATE(P)
    !$OMP SINGLE
    P => HEAD
    DO
      !$OMP TASK
      ! P is firstprivate by default
      CALL PROCESS(P)
      !$OMP END TASK
      P => P%NEXT
      IF (.NOT. ASSOCIATED (P) ) EXIT
    END DO
    !$OMP END SINGLE
    !$OMP END PARALLEL
  END SUBROUTINE
END MODULE
```

The `fib()` function should be called from within a `parallel` region for the different specified tasks to be executed in parallel. Also, only one thread of the `parallel` region should call `fib()` unless multiple concurrent Fibonacci computations are desired.

Example 15.4c

C/C++

```c
int fib(int n) {
  int i, j;
  if (n<2)
    return n;
  else {
    #pragma omp task shared(i)
    i=fib(n-1);
    #pragma omp task shared(j)
    // further code...
  }
}
```
Example 15.4f

RECURSIVE INTEGER FUNCTION fib(n) RESULT(res)
INTEGER n, i, j
IF ( n .LT. 2) THEN
  res = n
ELSE
  !$OMP TASK SHARED(i)
  i = fib( n-1 )
  !$OMP END TASK
  !$OMP TASK SHARED(j)
  j = fib( n-2 )
  !$OMP END TASK
  !$OMP TASKWAIT
  res = i+j
END IF
END FUNCTION

Note: There are more efficient algorithms for computing Fibonacci numbers. This classic recursion algorithm is for illustrative purposes.

The following example demonstrates a way to generate a large number of tasks with one thread and execute them with the threads in the team. While generating these tasks, the implementation may reach its limit on unassigned tasks. If it does, the implementation is allowed to cause the thread executing the task generating loop to suspend its task at the task scheduling point in the task directive, and start executing unassigned tasks. Once the number of unassigned tasks is sufficiently low, the thread may resume execution of the task generating loop.

Example 15.5c

#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];
extern void process(double);

int main() {
  #pragma omp parallel
  \}
\begin{verbatim}

define LARGENumber

{ #pragma omp single
  
  int i;
  for (i=0; i<LARGENumber; i++)
    
    #pragma omp task // i is firstprivate, item is shared
    process(item[i]);

  
}

C/C++

Fortran

Example 15.5f

real*8 item(10000000)
integer i

!$omp parallel
!$omp single ! loop iteration variable i is private
  do i=1,10000000
  !$omp task
    ! i is firstprivate, item is shared
    call process(item(i))
  !$omp end task
  end do
!$omp end single
!$omp end parallel
end

Fortran

The following example is the same as the previous one, except that the tasks are
generated in an untied task. While generating the tasks, the implementation may reach
its limit on unassigned tasks. If it does, the implementation is allowed to cause the
thread executing the task generating loop to suspend its task at the task scheduling point
in the task directive, and start executing unassigned tasks. If that thread begins
execution of a task that takes a long time to complete, the other threads may complete
all the other tasks before it is finished.

In this case, since the loop is in an untied task, any other thread is eligible to resume the
task generating loop. In the previous examples, the other threads would be forced to idle
until the generating thread finishes its long task, since the task generating loop was in a
tied task.
\end{verbatim}
The following two examples demonstrate how the scheduling rules illustrated in $ affect the usage of threadprivate variables in tasks. A threadprivate variable can be modified by another task that is executed by the same thread. Thus, the value of a
threadprivate variable cannot be assumed to be unchanged across a task scheduling point. In untied tasks, task scheduling points may be added in any place by the implementation.

A task switch may occur at a task scheduling point. A single thread may execute both of the task regions that modify tp. The parts of these task regions in which tp is modified may be executed in any order so the resulting value of var can be either 1 or 2.

---

**Example 15.7c**

```
int tp;
#pragma omp threadprivate(tp)
int var;
void work()
{
#pragma omp task
{
   /* do work here */
#include omp task
{
   tp = 1;
   /* do work here */
#include omp task
{
   /* no modification of tp */

   var = tp; //value of tp can be 1 or 2
}
   tp = 2;
}
}
```

---

**Example 15.7f**

```
module example
integer tp
!$omp threadprivate(tp)
integer var
contains
subroutine work
use globals
!$omp task
  ! do work here
!$omp task
  tp = 1
  ! do work here
!$omp task
  ! no modification of tp
!$omp end task
```
In this example, scheduling constraints prohibit a thread in the team from executing a new task that modifies `tp` while another such task region tied to the same thread is suspended. Therefore, the value written will persist across the task scheduling point.

**Example 15.8c**

```c
int tp;
#pragma omp threadprivate(tp)
int var;
void work()
{
    #pragma omp parallel
    {
        /* do work here */
        #pragma omp task
        {
            tp++;
            /* do work here */
        }
        #pragma omp task
        {
            /* do work here but don't modify tp */
            var = tp; //Value does not change after write above
        }
    }
}
```

**Example 15.8f**

```fortran
module example
integer tp
!$omp threadprivate(tp)
integer var
contains
subroutine work
!$omp parallel
! do work here
!$omp task
    tp = tp + 1
!$omp task
```

---

50 OpenMP API • Version 4.0.0 - July 2013
! do work here but don't modify tp
!$omp end task
  var = tp    ! value does not change after write above
!$omp end task
!$omp end parallel
end subroutine
end module

Fortran

The following two examples demonstrate how the scheduling rules illustrated in $ affect
the usage of locks and critical sections in tasks. If a lock is held across a task
scheduling point, no attempt should be made to acquire the same lock in any code that
may be interleaved. Otherwise, a deadlock is possible.

In the example below, suppose the thread executing task 1 defers task 2. When it
encounters the task scheduling point at task 3, it could suspend task 1 and begin task 2
which will result in a deadlock when it tries to enter critical region 1.

C/C++

Example 15.9c

void work()
{
  #pragma omp task
  { //Task 1
    #pragma omp task
    { //Task 2
      #pragma omp critical //Critical region 1
      { /*do work here*/ }
    }
    #pragma omp critical //Critical Region 2
    {
      //Capture data for the following task
      #pragma omp task
      { /* do work here */ } //Task 3
    }
  }
}
module example
contains
subroutine work
!$omp task
! Task 1
!$omp task
! Task 2
!$omp critical
! Critical region 1
! do work here
!$omp end critical
!$omp end task
!$omp critical
! Critical region 2
! Capture data for the following task
!$omp task
! Task 3
! do work here
!$omp end task
!$omp end critical
!$omp end task
end subroutine
end module
In the following example, `lock` is held across a task scheduling point. However, according to the scheduling restrictions, the executing thread can't begin executing one of the non-descendant tasks that also acquires `lock` before the task region is complete. Therefore, no deadlock is possible.

```
#include <omp.h>

void work() {
    omp_lock_t lock;
    omp_init_lock(&lock);
    #pragma omp parallel
    {
        int i;
        #pragma omp for
        for (i = 0; i < 100; i++) {
            #pragma omp task
            {
                // lock is shared by default in the task
                omp_set_lock(&lock);
                // Capture data for the following task
                #pragma omp task
                { /* do work here */ }
                omp_unset_lock(&lock);
            }
        }
        #pragma omp task
        { // Task Scheduling Point 1
            /* do work here */
        }
        omp_unset_lock(&lock);
    }
    omp_destroy_lock(&lock);
}
```
The following examples illustrate the use of the `mergeable` clause in the `task` construct. In this first example, the `task` construct has been annotated with the `mergeable` clause. The addition of this clause allows the implementation to reuse the data environment (including the ICVs) of the parent task for the task inside `foo` if the task is included or undeferred. Thus, the result of the execution may differ depending on whether the task is merged or not. Therefore the `mergeable` clause needs to be used with caution. In this example, the use of the `mergeable` clause is safe. As `x` is a shared variable the outcome does not depend on whether or not the task is merged (that is, the task will always increment the same variable and will always compute the same value for `x`).

```
module example
include 'omp_lib.h'
integer (kind=omp_lock_kind) lock
integer i
contains
subroutine work
  call omp_init_lock(lock)
!$omp parallel
  !$omp do
  do i=1,100
    !$omp task
      ! Outer task
      call omp_set_lock(lock) ! lock is shared by
      ! default in the task
      ! Capture data for the following task
      !$omp task ! Task Scheduling Point 1
          ! do work here
      !$omp end task
      !$omp end task
  end do
!$omp end parallel
  call omp_unset_lock(lock)
end subroutine
end module
```
#pragma omp task shared(x) mergeable
{
    x++;
}
#pragma omp taskwait
printf("%d\n",x);  // prints 3

C/C++

Example 15.11f

subroutine foo()
    integer :: x
    x = 2
    !$omp task shared(x) mergeable
    x = x + 1
    !$omp end task
    !$omp taskwait
    print *, x    ! prints 3
end subroutine

Fortran

This second example shows an incorrect use of the mergeable clause. In this example, the created task will access different instances of the variable x if the task is not merged, as x is firstprivate, but it will access the same variable x if the task is merged. As a result, the behavior of the program is unspecified and it can print two different values for x depending on the decisions taken by the implementation.

Example 15.12c

#include <stdio.h>
void foo ()
{
    int x = 2;
    #pragma omp task mergeable
    {
        x++;
    }
    #pragma omp taskwait
    printf("%d\n",x);  // prints 2 or 3
}

C/C++
Example 15.12f

```fortran
subroutine foo()
    integer :: x
    x = 2
    !$omp task mergeable
    x = x + 1
    !$omp end task
    !$omp taskwait
    print *, x ! prints 2 or 3
end subroutine
```

The following example shows the use of the **final** clause and the **omp_in_final** API call in a recursive binary search program. To reduce overhead, once a certain depth of recursion is reached the program uses the **final** clause to create only included tasks, which allow additional optimizations.

The use of the **omp_in_final** API call allows programmers to optimize their code by specifying which parts of the program are not necessary when a task can create only included tasks (that is, the code is inside a **final** task). In this example, the use of a different state variable is not necessary so once the program reaches the part of the computation that is finalized and copying from the parent state to the new state is eliminated. The allocation of **new_state** in the stack could also be avoided but it would make this example less clear. The **final** clause is most effective when used in conjunction with the **mergeable** clause since all tasks created in a **final** task region are included tasks that can be merged if the **mergeable** clause is present.

Example 15.13c

```c
#include <string.h>
#include <omp.h>
#define LIMIT 3 /* arbitrary limit on recursion depth */
void check_solution(char *);
void bin_search (int pos, int n, char *state)
{
    if ( pos == n ) {
        check_solution(state);
        return;
    }
    #pragma omp task final( pos > LIMIT ) mergeable
    {
        char new_state[n];
        if (!omp_in_final() ) {
            memcpy(new_state, state, pos);
```
state = new_state;
}
state[pos] = 0;
bin_search(pos+1, n, state );
}
#pragma omp task final( pos > LIMIT ) mergeable
{
    char new_state[n];
    if (!omp_in_final() ) {
        memcpy(new_state, state, pos );
        state = new_state;
    }
    state[pos] = 1;
    bin_search(pos+1, n, state );
}
#pragma omp taskwait
Fortran

Example 15.13f

recursive subroutine bin_search(pos, n, state)
    use omp_lib
    integer :: pos, n
    character, pointer :: state(:)
    character, target, dimension(n) :: new_state1, new_state2
    integer, parameter :: LIMIT = 3
    if (pos .eq. n) then
        call check_solution(state)
        return
    endif
    !$omp task final(pos > LIMIT) mergeable
    if (.not. omp_in_final()) then
        new_state1(1:pos) = state(1:pos)
        state => new_state1
    endif
    state(pos+1) = 'z'
    call bin_search(pos+1, n, state)
    !$omp end task
    !$omp task final(pos > LIMIT) mergeable
    if (.not. omp_in_final()) then
        new_state2(1:pos) = state(1:pos)
        state => new_state2
    endif
    state(pos+1) = 'y'
    call bin_search(pos+1, n, state)
    !$omp end task
    !$omp taskwait
end subroutine

The following example illustrates the difference between the if and the final clauses. The if clause has a local effect. In the first nest of tasks, the one that has the if clause will be undeferred but the task nested inside that task will not be affected by the if clause and will be created as usual. Alternatively, the final clause affects all task constructs in the final task region but not the final task itself. In the second nest of tasks, the nested tasks will be created as included tasks. Note also that the conditions for the if and final clauses are usually the opposite.

C/C++

Example 15.14c

void foo ( )
{
    int i;
    #pragma omp task if(0) // This task is undeferred
{ 
    #pragma omp task  // This task is a regular task
    for (i = 0; i < 3; i++) {
        #pragma omp task  // This task is a regular task
        bar();
    }
}

#pragma omp task final(1) // This task is a regular task
{
    #pragma omp task  // This task is included
    for (i = 0; i < 3; i++) {
        #pragma omp task  // This task is also included
        bar();
    }
}

Example 15.14f

subroutine foo()
    integer i
    !$omp task if(.FALSE.) ! This task is undeferred
    !$omp task             ! This task is a regular task
    do i = 1, 3
        !$omp task             ! This task is a regular task
        call bar()
        !$omp end task
    enddo
    !$omp end task
    !$omp end task
    !$omp task final(.TRUE.) ! This task is a regular task
    !$omp task             ! This task is included
    do i = 1, 3
        !$omp task             ! This task is also included
        call bar()
        !$omp end task
    enddo
    !$omp end task
    !$omp end task
    !$omp end task
end subroutine
The taskyield Directive

The following example illustrates the use of the `taskyield` directive. The tasks in the example compute something useful and then do some computation that must be done in a critical region. By using `taskyield` when a task cannot get access to the critical region the implementation can suspend the current task and schedule some other task that can do something useful.

```c
#include <omp.h>

void something_useful ( void );
void something_critical ( void );
void foo ( omp_lock_t * lock, int n )
{
    int i;

    for ( i = 0; i < n; i++ )
    {
        #pragma omp task
        {
            something_useful();
            while ( !omp_test_lock(lock) ) {
                #pragma omp taskyield
            }
            something_critical();
            omp_unset_lock(lock);
        }
    }
}
```

Example 16.1f

```fortran
subroutine foo ( lock, n )
    use omp_lib
    integer (kind=omp_lock_kind) :: lock
    integer n
    integer i

    do i = 1, n
        !$omp task
        call something_useful()
        do while ( .not. omp_test_lock(lock) )
            !$omp taskyield
        end do
        call something_critical()
        call omp_unset_lock(lock)
        !$omp end task
    end do

end subroutine
```

The workshare Construct

The following are examples of the workshare construct.

In the following example, workshare spreads work across the threads executing the parallel region, and there is a barrier after the last statement. Implementations must enforce Fortran execution rules inside of the workshare block.

Example 17.1f

```fortran
SUBROUTINE WSHARE1(AA, BB, CC, DD, EE, FF, N)
  INTEGER N
  REAL  AA(N,N), BB(N,N), CC(N,N), DD(N,N), EE(N,N), FF(N,N)
  !$OMP PARALLEL
  !$OMP WORKSHARE
      AA = BB
      CC = DD
      EE = FF
  !$OMP END WORKSHARE
  !$OMP END PARALLEL
END SUBROUTINE WSHARE1
```

In the following example, the barrier at the end of the first workshare region is eliminated with a nowait clause. Threads doing CC = DD immediately begin work on EE = FF when they are done with CC = DD.
Example 17.2f

```
SUBROUTINE WSHARE2(AA, BB, CC, DD, EE, FF, N)
INTEGER N
REAL AA(N,N), BB(N,N), CC(N,N)
REAL DD(N,N), EE(N,N), FF(N,N)
!
$OMP   PARALLEL
$OMP     WORKSHARE
    AA = BB
    CC = DD
$OMP     END WORKSHARE NOWAIT
$OMP     WORKSHARE
    EE = FF
$OMP     END WORKSHARE
$OMP   END PARALLEL
END SUBROUTINE WSHARE2
```

The following example shows the use of an atomic directive inside a workshare construct. The computation of \( \text{SUM(AA)} \) is workshared, but the update to \( R \) is atomic.

Example 17.3f

```
SUBROUTINE WSHARE3(AA, BB, CC, DD, N)
INTEGER N
REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
REAL R
R=0
!
$OMP   PARALLEL
$OMP     WORKSHARE
    AA = BB
$OMP       ATOMIC UPDATE
    R = R + \text{SUM(AA)}
    CC = DD
$OMP     END WORKSHARE
$OMP   END PARALLEL
END SUBROUTINE WSHARE3
```

Fortran WHERE and P0RALL statements are compound statements, made up of a control part and a statement part. When workshare is applied to one of these compound statements, both the control and the statement parts are workshared. The following example shows the use of a WHERE statement in a workshare construct.

```
Each task gets worked on in order by the threads:
AA = BB then
CC = DD then
EE .ne. 0 then
FF = 1 / EE then
GG = HH
```
Example 17.4f

```fortran
SUBROUTINE WSHARE4(AA, BB, CC, DD, EE, FF, GG, HH, N)
  INTEGER N
  REAL AA(N,N), BB(N,N), CC(N,N)
  REAL DD(N,N), EE(N,N), FF(N,N)
  REAL GG(N,N), HH(N,N)
  !$OMP PARALLEL
  !$OMP WORKSHARE
  AA = BB
  CC = DD
  WHERE (EE .ne. 0) FF = 1 / EE
  GG = HH
  !$OMP END WORKSHARE
  !$OMP END PARALLEL
END SUBROUTINE WSHARE4
```

In the following example, an assignment to a shared scalar variable is performed by one thread in a **workshare** while all other threads in the team wait.

Example 17.5f

```fortran
SUBROUTINE WSHARE5(AA, BB, CC, DD, N)
  INTEGER N
  REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
  INTEGER SHR
  !$OMP PARALLEL SHARED(SHR)
  !$OMP WORKSHARE
  AA = BB
  SHR = 1
  CC = DD * SHR
  !$OMP END WORKSHARE
  !$OMP END PARALLEL
END SUBROUTINE WSHARE5
```

The following example contains an assignment to a private scalar variable, which is performed by one thread in a **workshare** while all other threads wait. It is non-conforming because the private scalar variable is undefined after the assignment statement.
Example 17.6f

```fortran
SUBROUTINE WSHARE6_WRONG(AA, BB, CC, DD, N)
  INTEGER N
  REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
  INTEGER PRI
  !$OMP PARALLEL PRIVATE(PRI)
  !$OMP WORKSHARE
    AA = BB
    PRI = 1
    CC = DD * PRI
  !$OMP END WORKSHARE
  !$OMP END PARALLEL
END SUBROUTINE WSHARE6_WRONG
```

Fortran execution rules must be enforced inside a *workshare* construct. In the following example, the same result is produced in the following program fragment regardless of whether the code is executed sequentially or inside an OpenMP program with multiple threads:

Example 17.7f

```fortran
SUBROUTINE WSHARE7(AA, BB, CC, N)
  INTEGER N
  REAL AA(N), BB(N), CC(N)
  !$OMP PARALLEL
  !$OMP WORKSHARE
    AA(1:50) = BB(11:60)
    CC(11:20) = AA(1:10)
  !$OMP END WORKSHARE
  !$OMP END PARALLEL
END SUBROUTINE WSHARE7
```
The master Construct

The following example demonstrates the master construct. In the example, the master keeps track of how many iterations have been executed and prints out a progress report. The other threads skip the master region without waiting.

Example 18.1c

```c
#include <stdio.h>
extern float average(float, float, float);

void master_example( float* x, float* xold, int n, float tol )
{
    int c, i, toobig;
    float error, y;
    c = 0;
    #pragma omp parallel
    {
        do{
            #pragma omp for private(i)
            for( i = 1; i < n-1; ++i ){
                xold[i] = x[i];
            }
            #pragma omp single
            {
                toobig = 0;
            }
            #pragma omp for private(i, y, error) reduction(+:toobig)
            for( i = 1; i < n-1; ++i ){
                y = x[i];
                x[i] = average( xold[i-1], x[i], xold[i+1] );
                error = y - x[i];
                if( error > tol || error < -tol ) ++toobig;
            }
            #pragma omp master
            {
                ++c;
                printf( "iteration %d, toobig=%d\n", c, toobig );
            }
        }while( toobig > 0 );
    }
}
```
Example 18.1f

SUBROUTINE MASTER_EXAMPLE( X, XOLD, N, TOL )
REAL X(*), XOLD(*), TOL
INTEGER N
INTEGER C, I, TOOBIG
REAL ERROR, Y, AVERAGE
EXTERNAL AVERAGE
C = 0
TOOBIG = 1
!$OMP PARALLEL
   DO WHILE( TOOBIG > 0 )
      !$OMP DO PRIVATE(I)
         DO I = 2, N-1
            XOLD(I) = X(I)
         ENDDO
      !$OMP SINGLE
         TOOBIG = 0
      !$OMP END SINGLE
      !$OMP DO PRIVATE(I,Y,ERROR), REDUCTION(+:TOOBIG)
         DO I = 2, N-1
            Y = X(I)
            X(I) = AVERAGE( XOLD(I-1), X(I), XOLD(I+1) )
            ERROR = Y-X(I)
            IF( ERROR > TOL .OR. ERROR < -TOL ) TOOBIG = TOOBIG+1
         ENDDO
      !$OMP MASTER
         C = C + 1
      PRINT *, 'Iteration ', C, 'TOOBIG=', TOOBIG
   !$OMP END MASTER
ENDDO
!$OMP END PARALLEL
END SUBROUTINE MASTER_EXAMPLE
The critical Construct

The following example includes several `critical` constructs. The example illustrates a queuing model in which a task is dequeued and worked on. To guard against multiple threads dequeuing the same task, the dequeuing operation must be in a `critical` region. Because the two queues in this example are independent, they are protected by `critical` constructs with different names, `xaxis` and `yaxis`.

```c
void critical_example(float *x, float *y)
{
    int ix_next, iy_next;

    #pragma omp parallel shared(x, y) private(ix_next, iy_next)
    {
        #pragma omp critical (xaxis)
        ix_next = dequeue(x);
        work(ix_next, x);

        #pragma omp critical (yaxis)
        iy_next = dequeue(y);
        work(iy_next, y);
    }
}
```
Example 19.1f

```fortran
SUBROUTINE CRITICAL_EXAMPLE(X, Y)
  REAL X(*), Y(*)
  INTEGER IX_NEXT, IY_NEXT

  !$OMP PARALLEL SHARED(X, Y) PRIVATE(IX_NEXT, IY_NEXT)
  !$OMP CRITICAL(XAXIS)
  CALL DEQUEUE(IX_NEXT, X)
  !$OMP END CRITICAL(XAXIS)
  CALL WORK(IX_NEXT, X)

  !$OMP CRITICAL(YAXIS)
  CALL DEQUEUE(IY_NEXT, Y)
  !$OMP END CRITICAL(YAXIS)
  CALL WORK(IY_NEXT, Y)

  !$OMP END PARALLEL
END SUBROUTINE CRITICAL_EXAMPLE
```
The following example demonstrates using a worksharing construct inside a critical construct. This example is conforming because the worksharing single region is not closely nested inside the critical region. A single thread executes the one and only section in the sections region, and executes the critical region. The same thread encounters the nested parallel region, creates a new team of threads, and becomes the master of the new team. One of the threads in the new team enters the single region and increments \( i \) by 1. At the end of this example \( i \) is equal to 2.

```c
void critical_work()
{
    int i = 1;
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            #pragma omp critical (name)
            {
                #pragma omp parallel
                {
                    #pragma omp single
                    {
                        i++;
                    }
                }
            }
        }
    }
}
```
Example 20.1f

```fortran
SUBROUTINE CRITICAL_WORK()

    INTEGER I
    I = 1

    !$OMP PARALLEL SECTIONS
    !$OMP SECTION
    !$OMP CRITICAL (NAME)
    !$OMP PARALLEL
    !$OMP SINGLE
    I = I + 1
    !$OMP END SINGLE
    !$OMP END PARALLEL
    !$OMP END CRITICAL (NAME)
    !$OMP END PARALLEL SECTIONS

END SUBROUTINE CRITICAL_WORK
```
21 Binding of barrier Regions

The binding rules call for a barrier region to bind to the closest enclosing parallel region.

In the following example, the call from the main program to sub2 is conforming because the barrier region (in sub3) binds to the parallel region in sub2. The call from the main program to sub1 is conforming because the barrier region binds to the parallel region in subroutine sub2.

The call from the main program to sub3 is conforming because the barrier region binds to the implicit inactive parallel region enclosing the sequential part. Also note that the barrier region in sub3 when called from sub2 only synchronizes the team of threads in the enclosing parallel region and not all the threads created in sub1.
void work(int n) {}

void sub3(int n)
{
    work(n);
    #pragma omp barrier
    work(n);
}

void sub2(int k)
{
    #pragma omp parallel shared(k)
    sub3(k);
}

void sub1(int n)
{
    int i;
    #pragma omp parallel private(i) shared(n)
    {
        #pragma omp for
        for (i=0; i<n; i++)
            sub2(i);
    }
}

int main()
{
    sub1(2);
    sub2(2);
    sub3(2);
    return 0;
}
Example 21.1f

SUBROUTINE WORK(N)
    INTEGER N
END SUBROUTINE WORK

SUBROUTINE SUB3(N)
    INTEGER N
    CALL WORK(N)
    !$OMP BARRIER
    CALL WORK(N)
END SUBROUTINE SUB3

SUBROUTINE SUB2(K)
    INTEGER K
    !$OMP PARALLEL SHARED(K)
    CALL SUB3(K)
    !$OMP END PARALLEL
END SUBROUTINE SUB2

SUBROUTINE SUB1(N)
    INTEGER N
    INTEGER I
    !$OMP PARALLEL PRIVATE(I) SHARED(N)
    !$OMP DO
    DO I = 1, N
        CALL SUB2(I)
    END DO
    !$OMP END PARALLEL
END SUBROUTINE SUB1

PROGRAM EXAMPLE
    CALL SUB1(2)
    CALL SUB2(2)
    CALL SUB3(2)
END PROGRAM EXAMPLE
The atomic Construct

The following example avoids race conditions (simultaneous updates of an element of $x$ by multiple threads) by using the `atomic` construct.

The advantage of using the `atomic` construct in this example is that it allows updates of two different elements of $x$ to occur in parallel. If a `critical` construct were used instead, then all updates to elements of $x$ would be executed serially (though not in any guaranteed order).

Note that the `atomic` directive applies only to the statement immediately following it. As a result, elements of $y$ are not updated atomically in this example.

```c
float work1(int i)
{
    return 1.0 * i;
}

float work2(int i)
{
    return 2.0 * i;
}

void atomic_example(float *x, float *y, int *index, int n)
{
    int i;

    #pragma omp parallel for shared(x, y, index, n)
    for (i=0; i<n; i++) {
        #pragma omp atomic update
        x[index[i]] += work1(i);
        y[i] += work2(i);
    }
}
```

```c
int main()
{
    float x[1000];
    float y[10000];
    int index[10000];
    int i;

    for (i = 0; i < 10000; i++) {
        index[i] = i % 1000;
        y[i] = 0.0;
```
for (i = 0; i < 1000; i++)
    x[i] = 0.0;
atomic_example(x, y, index, 10000);
return 0;
Example 22.1f

```fortran
REAL FUNCTION WORK1(I)
    INTEGER I
    WORK1 = 1.0 * I
    RETURN
END FUNCTION WORK1

REAL FUNCTION WORK2(I)
    INTEGER I
    WORK2 = 2.0 * I
    RETURN
END FUNCTION WORK2

SUBROUTINE SUB(X, Y, INDEX, N)
    REAL X(*), Y(*)
    INTEGER INDEX(*), N
    INTEGER I

    !$OMP PARALLEL DO SHARED(X, Y, INDEX, N)
    DO I=1,N
        !$OMP ATOMIC UPDATE
        X(INDEX(I)) = X(INDEX(I)) + WORK1(I)
        Y(I) = Y(I) + WORK2(I)
    ENDDO

END SUBROUTINE SUB

PROGRAM ATOMIC_EXAMPLE
    REAL X(1000), Y(10000)
    INTEGER INDEX(10000)
    INTEGER I

    DO I=1,10000
        INDEX(I) = MOD(I, 1000) + 1
        Y(I) = 0.0
    ENDDO

    DO I = 1,1000
        X(I) = 0.0
    ENDDO

    CALL SUB(X, Y, INDEX, 10000)

END PROGRAM ATOMIC_EXAMPLE
```
The following example illustrates the `read` and `write` clauses for the `atomic` directive. These clauses ensure that the given variable is read or written, respectively, as a whole. Otherwise, some other thread might read or write part of the variable while the current thread was reading or writing another part of the variable. Note that most hardware provides atomic reads and writes for some set of properly aligned variables of specific sizes, but not necessarily for all the variable types supported by the OpenMP API.

```
Example 22.2c

C/C++

int atomic_read(const int *p)
{
    int value;
    /* Guarantee that the entire value of *p is read atomically. No part of *p can change during the read operation. */
    #pragma omp atomic read
    value = *p;
    return value;
}

void atomic_write(int *p, int value)
{
    /* Guarantee that value is stored atomically into *p. No part of *p can change * until after the entire write operation is completed. */
    #pragma omp atomic write
    *p = value;
}
```

C/C++
Fortran

Example 22.2f

function atomic_read(p)
    integer :: atomic_read
    integer, intent(in) :: p
! Guarantee that the entire value of p is read atomically. No part of
! p can change during the read operation.

!$omp atomic read
    atomic_read = p
    return
end function atomic_read

subroutine atomic_write(p, value)
    integer, intent(out) :: p
    integer, intent(in) :: value
! Guarantee that value is stored atomically into p. No part of p can change
! until after the entire write operation is completed.

!$omp atomic write
    p = value
end subroutine atomic_write

The following example illustrates the capture clause for the atomic directive. In
this case the value of a variable is captured, and then the variable is incremented. These
operations occur atomically. This particular example could be implemented using the
fetch-and-add instruction available on many kinds of hardware. The example also shows
a way to implement a spin lock using the capture and read clauses.

C/C++

Example 22.3c

int fetch_and_add(int *p)
{
    /* Atomically read the value of *p and then increment it. The previous value is
    * returned. This can be used to implement a simple lock as shown below.
    */
    int old;
    #pragma omp atomic capture
    { old = *p; (*p)++; }
    return old;
}

/*
 * Use fetch_and_add to implement a lock
 */
struct locktype {
    int ticketnumber;
    int turn;
};

void do_locked_work(struct locktype *lock) {

    int atomic_read(const int *p);
    void work();

    // Obtain the lock
    int myturn = fetch_and_add(&lock->ticketnumber);
    while (atomic_read(&lock->turn) != myturn);

    // Do some work. The flush is needed to ensure visibility of 
    // variables not involved in atomic directives

    #pragma omp flush
    work();
    #pragma omp flush

    // Release the lock
    fetch_and_add(&lock->turn);
}
Example 22.3f

function fetch_and_add(p)
    integer:: fetch_and_add
    integer, intent(inout) :: p

! Atomically read the value of p and then increment it. The previous value is returned. This can be used to implement a simple lock as shown below.
 !$omp atomic capture
     fetch_and_add = p
     p = p + 1
 !$omp end atomic
 end function fetch_and_add

! Use fetch_and_add to implement a lock
module m
    interface
        function fetch_and_add(p)
            integer :: fetch_and_add
            integer, intent(inout) :: p
        end function

        function atomic_read(p)
            integer :: atomic_read
            integer, intent(in) :: p
        end function
    end interface

    type locktype
        integer ticketnumber
        integer turn
    end type

    contains
        subroutine do_locked_work(lock)
            type(locktype), intent(inout) :: lock
            integer myturn
            integer junk

            ! obtain the lock
            myturn = fetch_and_add(lock%ticketnumber)
            do while (atomic_read(lock%turn) .ne. myturn)
                continue
            enddo

            ! Do some work. The flush is needed to ensure visibility of variables not involved in atomic directives
            !$omp flush
                call work
            !$omp flush
            ! Release the lock
            junk = fetch_and_add(lock%turn)
        end subroutine
    end module m
23 Restrictions on the atomic Construct

The following non-conforming examples illustrate the restrictions on the atomic construct.

```c
Example 23.1c

void atomic_wrong ()
{
    union {int n; float x;} u;

    #pragma omp parallel
    {
        #pragma omp atomic update
        u.n++;

        #pragma omp atomic update
        u.x += 1.0;

        /* Incorrect because the atomic constructs reference the same location
         * through incompatible types */
    }
}
```
Example 23.1f

```fortran
SUBROUTINE ATOMIC_WRONG()
   INTEGER:: I
   REAL:: R
   EQUIVALENCE(I,R)

   !$OMP PARALLEL
   !$OMP ATOMIC UPDATE
   I = I + 1
   !$OMP ATOMIC UPDATE
   R = R + 1.0
   !$OMP END PARALLEL
END SUBROUTINE ATOMIC_WRONG
```

Example 23.2c

```c
void atomic_wrong2 ()
{
   int  x;
   int *i;
   float  *r;

   i = &x;
   r = (float *)&x;

   #pragma omp parallel
   {
      #pragma omp atomic update
      *i += 1;

      #pragma omp atomic update
      *r += 1.0;
   
      /* Incorrect because the atomic constructs reference the same location
         through incompatible types */
   }
}
```
The following example is non-conforming because \( I \) and \( R \) reference the same location but have different types.

**Example 23.2f**

```fortran
SUBROUTINE SUB()
    COMMON /BLK/ R
    REAL R

 !$OMP ATOMIC UPDATE
    R = R + 1.0
END SUBROUTINE SUB

SUBROUTINE ATOMIC_WRONG2()
    COMMON /BLK/ I
    INTEGER I

 !$OMP PARALLEL
 !$OMP ATOMIC UPDATE
    I = I + 1
CALL SUB()
 !$OMP END PARALLEL
END SUBROUTINE ATOMIC_WRONG2
```

Although the following example might work on some implementations, this is also non-conforming:
Example 23.3f

```fortran
SUBROUTINE ATOMIC_WRONG3
    INTEGER:: I
    REAL:: R
    EQUIVALENCE(I,R)

!$OMP PARALLEL
!$OMP ATOMIC UPDATE
    I = I + 1
! incorrect because I and R reference the same location
! but have different types
!$OMP END PARALLEL

!$OMP PARALLEL
!$OMP ATOMIC UPDATE
    R = R + 1.0
! incorrect because I and R reference the same location
! but have different types
!$OMP END PARALLEL

END SUBROUTINE ATOMIC_WRONG3
```
The flush Construct without a List

The following example distinguishes the shared variables affected by a flush construct with no list from the shared objects that are not affected:

```
Example 24.1c

int x, *p = &x;

void f1(int *q)
{
  *q = 1;
  #pragma omp flush
  /* x, p, and *q are flushed */
  /* because they are shared and accessible */
  /* q is not flushed because it is not shared. */
}

void f2(int *q)
{
  #pragma omp barrier
  *q = 2;
  #pragma omp barrier
  /* a barrier implies a flush */
  /* x, p, and *q are flushed */
  /* because they are shared and accessible */
  /* q is not flushed because it is not shared. */
}

int g(int n)
{
  int i = 1, j, sum = 0;
  *p = 1;
  #pragma omp parallel reduction(+: sum) num_threads(10)
  {
    f1(&j);
    /* i, n and sum were not flushed */
    /* because they were not accessible in f1 */
    /* j was flushed because it was accessible */
    sum += j;
    f2(&j);
    /* i, n, and sum were not flushed */
    /* because they were not accessible in f2 */
    /* j was flushed because it was accessible */
```
int main()
{
    int result = g(7);
    return result;
}
Example 24.1f

```
SUBROUTINE F1(Q)
   COMMON /DATA/ X, P
   INTEGER, TARGET  :: X
   INTEGER, POINTER :: P
   INTEGER Q

   Q = 1
   !$OMP   FLUSH
   ! X, P and Q are flushed
   ! because they are shared and accessible
END SUBROUTINE F1

SUBROUTINE F2(Q)
   COMMON /DATA/ X, P
   INTEGER, TARGET  :: X
   INTEGER, POINTER :: P
   INTEGER Q

   !$OMP   BARRIER
   Q = 2
   !$OMP   BARRIER
   ! a barrier implies a flush
   ! X, P and Q are flushed
   ! because they are shared and accessible
END SUBROUTINE F2

INTEGER FUNCTION G(N)
   COMMON /DATA/ X, P
   INTEGER, TARGET  :: X
   INTEGER, POINTER :: P
   INTEGER N
   INTEGER I, J, SUM

   I = 1
   SUM = 0
   P = 1
   !$OMP   PARALLEL REDUCTION(+: SUM) NUM_THREADS(10)
   CALL F1(J)
   ! I, N and SUM were not flushed
   ! because they were not accessible in F1
   ! J was flushed because it was accessible
   SUM = SUM + J

   CALL F2(J)
   ! I, N, and SUM were not flushed
   ! because they were not accessible in F2
   ! J was flushed because it was accessible
   SUM = SUM + I + J + P + N

   !$OMP   END PARALLEL
```
G = SUM
END FUNCTION G

PROGRAM FLUSH_NOLIST
COMMON /DATA/ X, P
INTEGER, TARGET :: X
INTEGER, POINTER :: P
INTEGER RESULT, G

P => X
RESULT = G(7)
PRINT *, RESULT
END PROGRAM FLUSH_NOLIST
Placement of `flush`, `barrier`, `taskwait` and `taskyield` Directives

The following example is non-conforming, because the `flush`, `barrier`, `taskwait`, and `taskyield` directives are stand-alone directives and cannot be the immediate substatement of an `if` statement.

C/C++

```c
Example 25.1c

void standalone_wrong()
{
    int a = 1;

    if (a != 0)
        #pragma omp flush(a)
        /* incorrect as flush cannot be immediate substatement of if statement */

    if (a != 0)
        #pragma omp barrier
        /* incorrect as barrier cannot be immediate substatement of if statement */

    if (a != 0)
        #pragma omp taskyield
        /* incorrect as taskyield cannot be immediate substatement of if statement */

    if (a != 0)
        #pragma omp taskwait
        /* incorrect as taskwait cannot be immediate substatement of if statement */
}
```

C/C++

The following example is non-conforming, because the `flush`, `barrier`, `taskwait`, and `taskyield` directives are stand-alone directives and cannot be the action statement of an `if` statement or a labeled branch target.
Example 25.1f

```fortran
SUBROUTINE STANDALONE_WRONG()
   INTEGER  A
   A = 1
   ! the FLUSH directive must not be the action statement in an IF statement
   IF (A .NE. 0) !$OMP FLUSH(A)

   ! the BARRIER directive must not be the action statement in an IF statement
   IF (A .NE. 0) !$OMP BARRIER

   ! the TASKWAIT directive must not be the action statement in an IF statement
   IF (A .NE. 0) !$OMP TASKWAIT

   ! the TASKYIELD directive must not be the action statement in an IF statement
   IF (A .NE. 0) !$OMP TASKYIELD

   GOTO 100

   ! the FLUSH directive must not be a labeled branch target statement
   100 !$OMP FLUSH(A)
   GOTO 200

   ! the BARRIER directive must not be a labeled branch target statement
   200 !$OMP BARRIER
   GOTO 300

   ! the TASKWAIT directive must not be a labeled branch target statement
   300 !$OMP TASKWAIT
   GOTO 400

   ! the TASKYIELD directive must not be a labeled branch target statement
   400 !$OMP TASKYIELD

END SUBROUTINE
```

The following version of the above example is conforming because the `flush`, `barrier`, `taskwait`, and `taskyield` directives are enclosed in a compound statement.
Example 25.2c

```c
void standalone_ok()
{
    int a = 1;

    #pragma omp parallel
    {
        if (a != 0) {
            #pragma omp flush(a)
        }
        if (a != 0) {
            #pragma omp barrier
        }
        if (a != 0) {
            #pragma omp taskwait
        }
        if (a != 0) {
            #pragma omp taskyield
        }
    }
}
```

The following example is conforming because the `flush`, `barrier`, `taskwait`, and `taskyield` directives are enclosed in an if construct or follow the labeled branch target.
Example 25.2f

SUBROUTINE STANDALONE_OK()
  INTEGER A
  A = 1
  IF (A .NE. 0) THEN
    !$OMP FLUSH(A)
  ENDIF
  IF (A .NE. 0) THEN
    !$OMP BARRIER
  ENDIF
  IF (A .NE. 0) THEN
    !$OMP TASKWAIT
  ENDIF
  IF (A .NE. 0) THEN
    !$OMP TASKYIELD
  ENDIF
  GOTO 100
100 CONTINUE
  !$OMP FLUSH(A)
  GOTO 200
200 CONTINUE
  !$OMP BARRIER
  GOTO 300
300 CONTINUE
  !$OMP TASKWAIT
  GOTO 400
400 CONTINUE
  !$OMP TASKYIELD
END SUBROUTINE
26 The ordered Clause and the ordered Construct

Ordered constructs are useful for sequentially ordering the output from work that is done in parallel. The following program prints out the indices in sequential order:

```
#include <stdio.h>

void work(int k)
{
   #pragma omp ordered
   printf(" %d\n", k);
}

void ordered_example(int lb, int ub, int stride)
{
   int i;

   #pragma omp parallel for ordered schedule(dynamic)
   for (i=lb; i<ub; i+=stride)
      work(i);

   int main()
   {
      ordered_example(0, 100, 5);
      return 0;
   }
```
Example 26.1f

```fortran
SUBROUTINE WORK(K)
   INTEGER k

   !$OMP ORDERED
   WRITE(*,*) K
   !$OMP END ORDERED
END SUBROUTINE WORK

SUBROUTINE SUB(LB, UB, STRIDE)
   INTEGER LB, UB, STRIDE
   INTEGER I

   !$OMP PARALLEL DO ORDERED SCHEDULE(DYNAMIC)
   DO I=LB,UB,STRIDE
      CALL WORK(I)
   END DO
   !$OMP END PARALLEL DO
END SUBROUTINE SUB

PROGRAM ORDERED_EXAMPLE
   CALL SUB(1,100,5)
END PROGRAM ORDERED_EXAMPLE
```

It is possible to have multiple ordered constructs within a loop region with the ordered clause specified. The first example is non-conforming because all iterations execute two ordered regions. An iteration of a loop must not execute more than one ordered region:
Example 26.2c

```c
void work(int i) {}

void ordered_wrong(int n)
{
    int i;
    #pragma omp for ordered
    for (i=0; i<n; i++) {
        /* incorrect because an iteration may not execute more than one
           ordered region */
        #pragma omp ordered
        work(i);
        #pragma omp ordered
        work(i+1);
    }
}
```
Example 26.2f

```fortran
SUBROUTINE WORK(I)
INTEGER I
END SUBROUTINE WORK

SUBROUTINE ORDERED_WRONG(N)
INTEGER N

INTEGER I
!$OMP DO ORDERED
DO I = 1, N
! incorrect because an iteration may not execute more than one
! ordered region
!$OMP ORDERED
CALL WORK(I)
!$OMP END ORDERED
!$OMP ORDERED
CALL WORK(I+1)
!$OMP END ORDERED
END DO
END SUBROUTINE ORDERED_WRONG
```

The following is a conforming example with more than one `ordered` construct. Each iteration will execute only one `ordered` region:

Example 26.3c

```c
void work(int i) {}
void ordered_good(int n)
{
    int i;
    #pragma omp for ordered
    for (i=0; i<n; i++) {
        if (i <= 10) {
            #pragma omp ordered
            work(i);
        }
        if (i > 10) {
            #pragma omp ordered
            work(i+1);
        }
    }
}
```
Example 26.3f

SUBROUTINE ORDERED_GOOD(N)
  INTEGER N

  !$OMP  DO ORDERED
  DO I = 1,N
    IF (I <= 10) THEN
      !$OMP ORDERED
      CALL WORK(I)
      !$OMP END ORDERED
    ENDIF
  !$OMP END ORDERED
  ENDDO
  !$OMP ORDERED
  IF (I > 10) THEN
    CALL WORK(I+1)
    !$OMP END ORDERED
    ENDDO
  ENDIF
END SUBROUTINE ORDERED_GOOD
The threadprivate Directive

The following examples demonstrate how to use the threadprivate directive to give each thread a separate counter.

Example 27.1c

```c
int counter = 0;
#pragma omp threadprivate(counter)
int increment_counter()
{
    counter++;
    return(counter);
}
```

Example 27.1f

```fortran
INTEGER FUNCTION INCREMENT_COUNTER()
COMMON/INC_COMMON/COUNTER
!$OMP THREADPRIVATE(/INC_COMMON/)
COUNTER = COUNTER +1
INCREMENT_COUNTER = COUNTER
RETURN
END FUNCTION INCREMENT_COUNTER
```

The following example uses threadprivate on a static variable:

Example 27.2c

```c
int increment_counter_2()
{
    static int counter = 0;
    #pragma omp threadprivate(counter)
    counter++;
    return(counter);
}
```
The following example demonstrates unspecified behavior for the initialization of a
\texttt{threadprivate} variable. A \texttt{threadprivate} variable is initialized once at an
unspecified point before its first reference. Because \texttt{a} is constructed using the value of \texttt{x}
(which is modified by the statement \texttt{x++}), the value of \texttt{a.val} at the start of the
\texttt{parallel} region could be either 1 or 2. This problem is avoided for \texttt{b}, which uses an
auxiliary \texttt{const} variable and a copy-constructor.

\textit{Example 27.3c}

```c
class T {
    public:
        int val;
        T (int);
        T (const T&);
    }

T :: T (int v){
    val = v;
}

T :: T (const T& t) {
    val = t.val;
}

void g(T a, T b){
    a.val += b.val;
}

int x = 1;
T a(x);
const T b_aux(x); /* Capture value of x = 1 */
T b(b_aux);
#pragma omp threadprivate(a, b)

void f(int n) {
    x++;  
    //pragma omp parallel for
    /* In each thread:
    * a is constructed from x (with value 1 or 2?)
    * b is copy-constructed from b_aux
    */

    for (int i=0; i<n; i++) {
        g(a, b); /* Value of a is unspecified. */
    }
}
```

C/C++
The following examples show non-conforming uses and correct uses of the
\texttt{threadprivate} directive.

The following example is non-conforming because the common block is not declared
local to the subroutine that refers to it:

\textbf{Example 27.2f}

\begin{verbatim}
MODULE INC_MODULE
    COMMON /T/ A
END MODULE INC_MODULE

SUBROUTINE INC_MODULE_WRONG()
    USE INC_MODULE
    !$OMP THREADPRIVATE(/T/)
    !non-conforming because /T/ not declared in INC_MODULE_WRONG
END SUBROUTINE INC_MODULE_WRONG
\end{verbatim}

The following example is also non-conforming because the common block is not
declared local to the subroutine that refers to it:

\textbf{Example 27.3f}

\begin{verbatim}
SUBROUTINE INC_WRONG()
    COMMON /T/ A
    !$OMP THREADPRIVATE(/T/)
    CONTAINS
        SUBROUTINE INC_WRONG_SUB()
        !$OMP PARALLEL COPYIN(/T/)
        !non-conforming because /T/ not declared in INC_WRONG_SUB
        !$OMP END PARALLEL
        END SUBROUTINE INC_WRONG_SUB
    END SUBROUTINE INC_WRONG
\end{verbatim}
The following example is a correct rewrite of the previous example:

**Example 27.4f**

```fortran
SUBROUTINE INC_GOOD()
  COMMON /T/ A
  !$OMP THREADPRIVATE(/T/)
  
  CONTAINS
  SUBROUTINE INC_GOOD_SUB()
    COMMON /T/ A
    !$OMP THREADPRIVATE(/T/)
    !$OMP PARALLEL COPYIN(/T/)
    !$OMP END PARALLEL
    END SUBROUTINE INC_GOOD_SUB
  END SUBROUTINE INC_GOOD

  !$OMP THREADPRIVATE(A)
  ALLOCATE (A(3))
  A = (/1,2,3/)
  PTR => TARG
  I = 5

  !$OMP PARALLEL COPYIN(I, PTR)
  CRITICAL
  IF (FIRSTIN) THEN
    TARG = 4           ! Update target of ptr
    I = I + 10
    IF (ALLOCATED(A)) A = A + 10
    FIRSTIN = .FALSE.
  END IF
  IF (ALLOCATED(A)) THEN
    PRINT *, 'a = ', A
  ELSE
    PRINT *, 'a is not allocated'
  END IF
END SUBROUTINE INC_GOOD
```

The following is an example of the use of `threadprivate` for local variables:

**Example 27.5f**

```fortran
PROGRAM INC_GOOD2
  INTEGER, ALLOCATABLE, SAVE :: A(:)
  INTEGER, POINTER, SAVE :: PTR
  INTEGER, SAVE :: I
  INTEGER, TARGET :: TARG
  LOGICAL :: FIRSTIN = .TRUE.
  !$OMP THREADPRIVATE(A, I, PTR)
  ALLOCATE (A(3))
  A = (/1,2,3/)
  PTR => TARG
  I = 5

  !$OMP PARALLEL COPYIN(I, PTR)
  CRITICAL
  IF (FIRSTIN) THEN
    TARG = 4           ! Update target of ptr
    I = I + 10
    IF (ALLOCATED(A)) A = A + 10
    FIRSTIN = .FALSE.
  END IF
  IF (ALLOCATED(A)) THEN
    PRINT *, 'a = ', A
  ELSE
    PRINT *, 'a is not allocated'
  END IF
END PROGRAM INC_GOOD2
```
The above program, if executed by two threads, will print one of the following two sets of output:

```
a = 11 12 13
ptr = 4
i = 15

A is not allocated
ptr = 4
i = 5
```
or

```
A is not allocated
ptr = 4
i = 5
```

The following is an example of the use of `threadprivate` for module variables:
Example 27.6f

```fortran
MODULE INC_MODULE_GOOD3
  REAL, POINTER :: WORK(:)
  SAVE WORK
  !$OMP THREADPRIVATE(WORK)
END MODULE INC_MODULE_GOOD3

SUBROUTINE SUB1(N)
  USE INC_MODULE_GOOD3
  !$OMP PARALLEL PRIVATE(THE_SUM)
  ALLOCATE(WORK(N))
  CALL SUB2(THE_SUM)
  WRITE(*,*) THE_SUM
  !$OMP END PARALLEL
END SUBROUTINE SUB1

SUBROUTINE SUB2(THE_SUM)
  USE INC_MODULE_GOOD3
  WORK(:) = 10
  THE_SUM=SUM(WORK)
END SUBROUTINE SUB2

PROGRAM INC_GOOD3
  N = 10
  CALL SUB1(N)
END PROGRAM INC_GOOD3
```

The following example illustrates initialization of `threadprivate` variables for class-type `T`. `t1` is default constructed, `t2` is constructed taking a constructor accepting one argument of integer type, `t3` is copy constructed with argument `f()`:

Example 27.4c

```c
static T t1;
#pragma omp threadprivate(t1)
static T t2( 23 );
#pragma omp threadprivate(t2)
static T t3 = f();
#pragma omp threadprivate(t3)
```

The following example illustrates the use of `threadprivate` for static class members. The `threadprivate` directive for a static class member must be placed inside the class definition.
Example 27.5c

class T {
public:
    static int i;
#pragma omp threadprivate(i)
};
Parallel Random Access Iterator Loop

The following example shows a parallel random access iterator loop.

Example 28.1c

```c++
#include <vector>
void iterator_example()
{
    std::vector<int> vec(23);
    std::vector<int>::iterator it;
    #pragma omp parallel for default(none) shared(vec)
    for (it = vec.begin(); it < vec.end(); it++)
    {
        // do work with *it //
    }
}
```
Fortran Restrictions on shared and private Clauses with Common Blocks

When a named common block is specified in a **private**, **firstprivate**, or **lastprivate** clause of a construct, none of its members may be declared in another data-sharing attribute clause on that construct. The following examples illustrate this point.

The following example is conforming:

**Example 29.1f**

```fortran
SUBROUTINE COMMON_GOOD()
    COMMON /C/ X, Y
    REAL X, Y

    !$OMP PARALLEL PRIVATE (/C/)
        ! do work here
    !$OMP END PARALLEL
    !$OMP PARALLEL SHARED (X,Y)
        ! do work here
    !$OMP END PARALLEL
END SUBROUTINE COMMON_GOOD
```

The following example is also conforming:

**Example 29.2f**

```fortran
SUBROUTINE COMMON_GOOD2()
    COMMON /C/ X, Y
    REAL X, Y
    INTEGER I

    !$OMP PARALLEL
    !$OMP DO PRIVATE(/C/)
        DO I=1,1000
            ! do work here
        ENDDO
    !$OMP END DO
    !$OMP DO PRIVATE(X)
        DO I=1,1000
            ! do work here
        ENDDO
    !$OMP END PARALLEL
END SUBROUTINE COMMON_GOOD2
```
The following example is conforming:

**Example 29.3f**

```fortran
SUBROUTINE COMMON_GOOD3()
    COMMON /C/ X,Y
    !$OMP PARALLEL PRIVATE (/C/)
    ! do work here
    !$OMP END PARALLEL
    !$OMP PARALLEL SHARED (/C/)
    ! do work here
    !$OMP END PARALLEL
END SUBROUTINE COMMON_GOOD3
```

The following example is non-conforming because \( x \) is a constituent element of \( C \):

**Example 29.4f**

```fortran
SUBROUTINE COMMON_WRONG()
    COMMON /C/ X,Y
    ! Incorrect because X is a constituent element of C
    !$OMP PARALLEL PRIVATE(/C/), SHARED(X)
    ! do work here
    !$OMP END PARALLEL
END SUBROUTINE COMMON_WRONG
```

The following example is non-conforming because a common block may not be declared both shared and private:

**Example 29.5f**

```fortran
SUBROUTINE COMMON_WRONG2()
    COMMON /C/ X,Y
    ! Incorrect: common block C cannot be declared both
    ! shared and private
    !$OMP PARALLEL PRIVATE (/C/), SHARED(/C/)
    ! do work here
    !$OMP END PARALLEL
END SUBROUTINE COMMON_WRONG2
```
The **default(none)** Clause

The following example distinguishes the variables that are affected by the **default(none)** clause from those that are not.

```
#include <omp.h>
int x, y, z[1000];
#pragma omp threadprivate(x)

void default_none(int a) {
    const int c = 1;
    int i = 0;

    #pragma omp parallel default(none) private(a) shared(z)
    {
        int j = omp_get_num_threads();
        /* O.K. - j is declared within parallel region */
        a = z[j]; /* O.K. - a is listed in private clause */
        /* - z is listed in shared clause */
        x = c; /* O.K. - x is threadprivate */
        /* - c has const-qualified type */
        z[i] = y; /* Error - cannot reference i or y here */

        #pragma omp for firstprivate(y)
        /* Error - Cannot reference y in the firstprivate clause */
        for (i=0; i<10 ; i++) {
            z[i] = i; /* O.K. - i is the loop iteration variable */
        }

        z[i] = y; /* Error - cannot reference i or y here */
    }
}
```
Example 30.1f

```fortran
SUBROUTINE DEFAULT_NONE(A)
   INCLUDE "omp_lib.h" ! or USE OMP_LIB

   INTEGER A

   INTEGER X, Y, Z(1000)
   COMMON/BLOCKX/X
   COMMON/BLOCKY/Y
   COMMON/BLOCKZ/Z

   !$OMP THREADPRIVATE(/BLOCKX/)

   INTEGER I, J
   i = 1

   !$OMP PARALLEL DEFAULT(NONE) PRIVATE(A) SHARED(Z) PRIVATE(J)
   J = OMP_GET_NUM_THREADS();
   ! O.K. - J is listed in PRIVATE clause
   A = Z(J) ! O.K. - A is listed in PRIVATE clause
   ! - Z is listed in SHARED clause
   X = 1 ! O.K. - X is THREADPRIVATE
   Z(I) = Y ! Error - cannot reference I or Y here

   !$OMP DO firstprivate(y)
   ! Error - Cannot reference y in the firstprivate clause
   DO I = 1,10
      Z(I) = I ! O.K. - I is the loop iteration variable
   END DO

   Z(I) = Y ! Error - cannot reference I or Y here

   !$OMP END PARALLEL

END SUBROUTINE DEFAULT_NONE
```
Race Conditions Caused by Implied Copies of Shared Variables in Fortran

The following example contains a race condition, because the shared variable, which is an array section, is passed as an actual argument to a routine that has an assumed-size array as its dummy argument. The subroutine call passing an array section argument may cause the compiler to copy the argument into a temporary location prior to the call and copy from the temporary location into the original variable when the subroutine returns. This copying would cause races in the parallel region.

Example 31.1f

```fortran
SUBROUTINE SHARED_RACE
  INCLUDE "omp_lib.h" ! or USE OMP_LIB
  REAL A(20)
  INTEGER MYTHREAD

  !$OMP PARALLEL SHARED(A) PRIVATE(MYTHREAD)

  MYTHREAD = OMP_GET_THREAD_NUM()
  IF (MYTHREAD .EQ. 0) THEN
    CALL SUB(A(1:10)) ! compiler may introduce writes to A(6:10)
  ELSE
    A(6:10) = 12
  ENDIF

  !$OMP END PARALLEL
END SUBROUTINE SHARED_RACE

SUBROUTINE SUB(X)
  REAL X(*)
  X(1:5) = 4
END SUBROUTINE SUB
```
The private Clause

In the following example, the values of original list items $i$ and $j$ are retained on exit from the parallel region, while the private list items $i$ and $j$ are modified within the parallel construct.

```c
#include <stdio.h>
#include <assert.h>

int main()
{
    int i, j;
    int *ptr_i, *ptr_j;

    i = 1;
    j = 2;

    ptr_i = &i;
    ptr_j = &j;

    #pragma omp parallel private(i) firstprivate(j)
    {
        i = 3;
        j = j + 2;
        assert (*ptr_i == 1 && *ptr_j == 2);
    }

    assert(i == 1 && j == 2);
    return 0;
}
```
Example 32.1f

```fortran
PROGRAM PRIV_EXAMPLE
    INTEGER I, J

    I = 1
    J = 2

    !$OMP PARALLEL PRIVATE(I) FIRSTPRIVATE(J)
    I = 3
    J = J + 2
    !$OMP END PARALLEL

    PRINT *, I, J  ! I .eq. 1 .and. J .eq. 2
END PROGRAM PRIV_EXAMPLE
```

In the following example, all uses of the variable $a$ within the loop construct in the routine $f$ refer to a private list item $a$, while it is unspecified whether references to $a$ in the routine $g$ are to a private list item or the original list item.

Example 32.2c

```c
int a;

void g(int k) {
    a = k; /* Accessed in the region but outside of the construct;
             * therefore unspecified whether original or private list
             * item is modified. */
}

void f(int n) {
    int a = 0;

    #pragma omp parallel for private(a)
    for (int i=1; i<n; i++) {
        a = i;
        g(a*2);  /* Private copy of "a" */
    }
}
```
Example 32.2f

```fortran
MODULE PRIV_EXAMPLE2
  REAL A

  CONTAINS

  SUBROUTINE G(K)
    REAL K
    A = K  ! Accessed in the region but outside of the
    ! construct; therefore unspecified whether
    ! original or private list item is modified.
  END SUBROUTINE G

  SUBROUTINE F(N)
    INTEGER N
    REAL A
    INTEGER I

    !$OMP PARALLEL DO PRIVATE(A)
    DO I = 1,N
      A = I
      CALL G(A*2)
    ENDDO
    !$OMP END PARALLEL DO
  END SUBROUTINE F

END MODULE PRIV_EXAMPLE2
```

The following example demonstrates that a list item that appears in a `private` clause in a `parallel` construct may also appear in a `private` clause in an enclosed worksharing construct, which results in an additional private copy.
Example 32.3c

```c
#include <assert.h>
void priv_example3()
{
    int i, a;

    #pragma omp parallel private(a)
    {
        a = 1;
        #pragma omp parallel for private(a)
        for (i=0; i<10; i++)
        {
            a = 2;
        }
        assert(a == 1);
    }
}
```

Example 32.3f

```fortran
SUBROUTINE PRIV_EXAMPLE3()
    INTEGER I, A

    !$OMP PARALLEL PRIVATE(A)
    A = 1
    !$OMP PARALLEL DO PRIVATE(A)
    DO I = 1, 10
        A = 2
    END DO
    !$OMP END PARALLEL DO
    PRINT *, A ! Outer A still has value 1
    !$OMP END PARALLEL
END SUBROUTINE PRIV_EXAMPLE3
```
Fortran Restrictions on Storage Association with the private Clause

The following non-conforming examples illustrate the implications of the private clause rules with regard to storage association.

Example 33.1f

```fortran
SUBROUTINE SUB()
  COMMON /BLOCK/ X
  PRINT *,X ! X is undefined
END SUBROUTINE SUB

PROGRAM PRIV_RESTRICT
  COMMON /BLOCK/ X
  X = 1.0
  !$OMP PARALLEL PRIVATE (X)
  X = 2.0
  CALL SUB()
  !$OMP END PARALLEL
END PROGRAM PRIV_RESTRICT
```

Example 33.2f

```fortran
PROGRAM PRIV_RESTRICT2
  COMMON /BLOCK2/ X
  X = 1.0
  !$OMP PARALLEL PRIVATE (X)
  X = 2.0
  CALL SUB()
  !$OMP END PARALLEL
  CONTAINS
  SUBROUTINE SUB()
    COMMON /BLOCK2/ Y
    PRINT *,X ! X is undefined
    PRINT *,Y ! Y is undefined
    END SUBROUTINE SUB
  END SUBROUTINE SUB
```

END PROGRAM PRIV_RESTRICT2
Example 33.3f

```fortran
PROGRAM PRIV_RESTRICT3
    EQUIVALENCE (X,Y)
    X = 1.0

    !$OMP PARALLEL PRIVATE(X)
    PRINT *,Y                       ! Y is undefined
    Y = 10
    PRINT *,X                       ! X is undefined
    !$OMP END PARALLEL
END PROGRAM PRIV_RESTRICT3
```

Example 33.4f

```fortran
PROGRAM PRIV_RESTRICT4
    INTEGER I, J
    INTEGER A(100), B(100)
    EQUIVALENCE (A(51), B(1))

    !$OMP PARALLEL DO DEFAULT(PRIVATE) PRIVATE(I,J) LASTPRIVATE(A)
    DO I=1,100
        DO J=1,100
            B(J) = J - 1
        ENDDO
    DO J=1,100
        A(J) = J       ! B becomes undefined at this point
    ENDDO
    DO J=1,50
        B(J) = B(J) + 1  ! B is undefined
    ENDDO
    !$OMP END PARALLEL DO       ! The LASTPRIVATE write for A has
    PRINT *, B                  ! undefined results
    ! A becomes undefined at this point
END PROGRAM PRIV_RESTRICT4
```
Example 33.5f

SUBROUTINE SUB1(X)
  DIMENSION X(10)
  ! This use of X does not conform to the
  ! specification. It would be legal Fortran 90,
  ! but the OpenMP private directive allows the
  ! compiler to break the sequence association that
  ! A had with the rest of the common block.
  FORALL (I = 1:10) X(I) = I
END SUBROUTINE SUB1

PROGRAM PRIV_RESTRICT5
  COMMON /BLOCK5/ A
  DIMENSION B(10)
  EQUIVALENCE (A,B(1))
  ! the common block has to be at least 10 words
  A = 0

  !$OMP PARALLEL PRIVATE(/BLOCK5/)
  ! Without the private clause,
  ! we would be passing a member of a sequence
  ! that is at least ten elements long.
  ! With the private clause, A may no longer be
  ! sequence-associated.
  CALL SUB1(A)
  !$OMP MASTER
    PRINT *, A
  !$OMP END MASTER
  !$OMP END PARALLEL
END PROGRAM PRIV_RESTRICT5

Fortran
C/C++

C/C++ Arrays in a firstprivate Clause

The following example illustrates the size and value of list items of array or pointer type in a firstprivate clause. The size of new list items is based on the type of the corresponding original list item, as determined by the base language.

In this example:

• The type of A is array of two arrays of two ints.
• The type of B is adjusted to pointer to array of n ints, because it is a function parameter.
• The type of C is adjusted to pointer to int, because it is a function parameter.
• The type of D is array of two arrays of two ints.
• The type of E is array of n arrays of n ints.

Note that B and E involve variable length array types.

The new items of array type are initialized as if each integer element of the original array is assigned to the corresponding element of the new array. Those of pointer type are initialized as if by assignment from the original item to the new item.
Example 34.1c

```c
#include <assert.h>

int A[2][2] = {1, 2, 3, 4};

void f(int n, int B[n][n], int C[])
{
    int D[2][2] = {1, 2, 3, 4};
    int E[n][n];

    assert(n >= 2);
    E[1][1] = 4;

    #pragma omp parallel firstprivate(B, C, D, E)
    {
        assert(sizeof(B) == sizeof(int (*)[n]));
        assert(sizeof(C) == sizeof(int*));
        assert(sizeof(D) == 4 * sizeof(int));
        assert(sizeof(E) == n * n * sizeof(int));

        /* Private B and C have values of original B and C. */
        assert(&B[1][1] == &A[1][1]);
        assert(&C[3] == &A[1][1]);
        assert(D[1][1] == 4);
        assert(E[1][1] == 4);
    }
}

int main() {
    f(2, A, A[0]);
    return 0;
}
```

C/C++
The lastprivate Clause

Correct execution sometimes depends on the value that the last iteration of a loop assigns to a variable. Such programs must list all such variables in a lastprivate clause so that the values of the variables are the same as when the loop is executed sequentially.

C/C++

Example 35.1c

```c
void lastpriv (int n, float *a, float *b)
{
    int i;

    #pragma omp parallel
    {
        #pragma omp for lastprivate(i)
        for (i=0; i<n-1; i++)
            a[i] = b[i] + b[i+1];
    }

    a[i]=b[i]; /* i == n-1 here */
}
```

Fortran

Example 35.1f

```fortran
SUBROUTINE LASTPRIV(N, A, B)
    INTEGER N
    REAL A(*), B(*)
    INTEGER I
!$OMP PARALLEL
!$OMP DO LASTPRIVATE(I)
    DO I=1,N-1
        A(I) = B(I) + B(I+1)
    ENDDO
!$OMP END PARALLEL
    A(I) = B(I)  ! I has the value of N here
END SUBROUTINE LASTPRIV
```
The reduction Clause

The following example demonstrates the `reduction` clause; note that some reductions can be expressed in the loop in several ways, as shown for the `max` and `min` reductions below:

```c
#include <math.h>
void reduction1(float *x, int *y, int n)
{
    int i, b, c;
    float a, d;
    a = 0.0;
    b = 0;
    c = y[0];
    d = x[0];
    #pragma omp parallel for private(i) shared(x, y, n) \ 
            reduction(+:a) reduction(^:b) \ 
            reduction(min:c) reduction(max:d)
    for (i=0; i<n; i++) {
        a += x[i];
        b ^= y[i];
        if (c > y[i]) c = y[i];
        d = fmaxf(d, x[i]);
    }
}
```
Example 36.1f

```fortran
SUBROUTINE REDUCTION1(A, B, C, D, X, Y, N)
    REAL :: X(*), A, D
    INTEGER :: Y(*), N, B, C
    INTEGER :: I
    A = 0
    B = 0
    C = Y(1)
    D = X(1)
    !$OMP PARALLEL DO PRIVATE(I) SHARED(X, Y, N) REDUCTION(+:A) &
    !$OMP& REDUCTION(IEOR:B) REDUCTION(MIN:C)  REDUCTION(MAX:D)
    DO I=1,N
        A = A + X(I)
        B = IEOR(B, Y(I))
        C = MIN(C, Y(I))
        IF (D < X(I)) D = X(I)
    END DO
END SUBROUTINE REDUCTION1
```

A common implementation of the preceding example is to treat it as if it had been written as follows:

Example 36.2c

```c
#include <limits.h>
#include <math.h>
void reduction2(float *x, int *y, int n)
{
    int i, b, b_p, c, c_p;
    float a, a_p, d, d_p;
    a = 0.0f;
    b = 0;
    c = y[0];
    d = x[0];
    #pragma omp parallel shared(a, b, c, d, x, y, n) \ 
    private(a_p, b_p, c_p, d_p)
    {
        a_p = 0.0f;
        b_p = 0;
        c_p = INT_MAX;
        d_p = -HUGE_VALF;
        #pragma omp for private(i)
        for (i=0; i<n; i++) {
            a_p += x[i];
```
b_p ^= y[i];
if (c_p > y[i]) c_p = y[i];
d_p = fmaxf(d_p, x[i]);
}
#pragma omp critical
{
    a += a_p;
    b ^= b_p;
    if (c > c_p) c = c_p;
    d = fmaxf(d, d_p);
}
}
Example 36.2f

SUBROUTINE REDUCTION2(A, B, C, D, X, Y, N)
  REAL :: X(*), A, D
  INTEGER :: Y(*), N, B, C
  REAL :: A_P, D_P
  INTEGER :: I, B_P, C_P
  A = 0
  B = 0
  C = Y(1)
  D = X(1)
  !$OMP PARALLEL SHARED(X, Y, A, B, C, D, N) &
  A_P = 0.0
  B_P = 0
  C_P = HUGE(C_P)
  D_P = -HUGE(D_P)
  !$OMP DO PRIVATE(I)
  DO I=1,N
    A_P = A_P + X(I)
    B_P = IEOR(B_P, Y(I))
    C_P = MIN(C_P, Y(I))
    IF (D_P < X(I)) D_P = X(I)
  END DO
  !$OMP CRITICAL
  A = A + A_P
  B = IEOR(B, B_P)
  C = MIN(C, C_P)
  D = MAX(D, D_P)
  !$OMP END CRITICAL
  !$OMP END PARALLEL
END SUBROUTINE REDUCTION2

The following program is non-conforming because the reduction is on the intrinsic procedure name MAX but that name has been redefined to be the variable named MAX.
Example 36.3f

```fortran
PROGRAM REDUCTION_WRONG
MAX = HUGE(0)
M = 0

!$OMP PARALLEL DO REDUCTION(MAX: M)
! MAX is no longer the intrinsic so this is non-conforming
DO I = 1, 100
   CALL SUB(M,I)
END DO

END PROGRAM REDUCTION_WRONG

SUBROUTINE SUB(M,I)
   M = MAX(M,I)
END SUBROUTINE SUB

The following conforming program performs the reduction using the intrinsic procedure name MAX even though the intrinsic MAX has been renamed to REN.

Example 36.4f

```fortran
MODULE M
   INTRINSIC MAX
END MODULE M

PROGRAM REDUCTION3
   USE M, REN => MAX
   N = 0
   !$OMP PARALLEL DO REDUCTION(REN: N) ! still does MAX
   DO I = 1, 100
      N = MAX(N,I)
   END DO
END PROGRAM REDUCTION3
```

The following conforming program performs the reduction using intrinsic procedure name MAX even though the intrinsic MAX has been renamed to MIN.
**Example 36.5f**

```fortran
MODULE MOD
    INTRINSIC MAX, MIN
END MODULE MOD

PROGRAM REDUCTION4
    USE MOD, MIN=>MAX, MAX=>MIN
    REAL :: R
    R = -HUGE(0.0)
    !$OMP PARALLEL DO REDUCTION(MIN: R) ! still does MAX
    DO I = 1, 1000
        R = MIN(R, SIN(REAL(I)))
    END DO
    PRINT *, R
END PROGRAM REDUCTION4
```

The following example is non-conforming because the initialization (\(a = 0\)) of the original list item \(a\) is not synchronized with the update of \(a\) as a result of the reduction computation in the `for` loop. Therefore, the example may print an incorrect value for \(a\).

To avoid this problem, the initialization of the original list item \(a\) should complete before any update of \(a\) as a result of the `reduction` clause. This can be achieved by adding an explicit barrier after the assignment \(a = 0\), or by enclosing the assignment \(a = 0\) in a `single` directive (which has an implied barrier), or by initializing \(a\) before the start of the `parallel` region.

**Example 36.3c**

```c
#include <stdio.h>

int main (void)
{
    int a, i;

    #pragma omp parallel shared(a) private(i)
    {
        #pragma omp master
        a = 0;

        // To avoid race conditions, add a barrier here.
        #pragma omp for reduction(+:a)
        for (i = 0; i < 10; i++) {
```
a += i;
}

#pragma omp single
printf ("Sum is %d\n", a);
}

C/C++

Example 36.6f

INTEGER A, I

!$OMP PARALLEL SHARED(A) PRIVATE(I)

!$OMP MASTER
A = 0
!$OMP END MASTER

! To avoid race conditions, add a barrier here.

!$OMP DO REDUCTION(+:A)
DO I= 0, 9
   A = A + I
END DO

!$OMP SINGLE
PRINT *, "Sum is ", A
!$OMP END SINGLE

!$OMP END PARALLEL
END

Fortran
The *copyin* Clause

The *copyin* clause is used to initialize threadprivate data upon entry to a *parallel* region. The value of the threadprivate variable in the master thread is copied to the threadprivate variable of each other team member.

```c
#include <stdlib.h>

float* work;
int size;
float tol;

#pragma omp threadprivate(work,size,tol)

void build()
{
    int i;
    work = (float*)malloc( sizeof(float)*size );
    for( i = 0; i < size; ++i ) work[i] = tol;
}

void copyin_example( float t, int n )
{
    tol = t;
    size = n;
    #pragma omp parallel copyin(tol,size)
    {
        build();
    }
}
```

```c++
```
Example 37.1f

MODULE M
  REAL, POINTER, SAVE :: WORK(:)
  INTEGER :: SIZE
  REAL :: TOL
  !$OMP THREADPRIVATE(WORK,SIZE,TOL)
END MODULE M

SUBROUTINE COPYIN_EXAMPLE( T, N )
  USE M
  REAL :: T
  INTEGER :: N
  TOL = T
  SIZE = N
  !$OMP PARALLEL COPYIN(TOL,SIZE)
  CALL BUILD
  !$OMP END PARALLEL
END SUBROUTINE COPYIN_EXAMPLE

SUBROUTINE BUILD
  USE M
  ALLOCATE(WORK(SIZE))
  WORK = TOL
END SUBROUTINE BUILD
The copyprivate Clause

The `copyprivate` clause can be used to broadcast values acquired by a single thread directly to all instances of the private variables in the other threads. In this example, if the routine is called from the sequential part, its behavior is not affected by the presence of the directives. If it is called from a `parallel` region, then the actual arguments with which `a` and `b` are associated must be private.

The thread that executes the structured block associated with the `single` construct broadcasts the values of the private variables `a`, `b`, `x`, and `y` from its implicit task's data environment to the data environments of the other implicit tasks in the thread team. The broadcast completes before any of the threads have left the barrier at the end of the construct.

**Example 38.1c**

```c
#include <stdio.h>
float x, y;
#pragma omp threadprivate(x, y)
void init(float a, float b ) {
    #pragma omp single copyprivate(a,b,x,y)
    {
        scanf("%f %f %f %f", &a, &b, &x, &y);
    }
}
```

**Example 38.1f**

```fortran
SUBROUTINE INIT(A,B)
REAL A, B
COMMON /XY/ X,Y
!$OMP THREADPRIVATE (/XY/)
!$OMP SINGLE
    READ (11) A,B,X,Y
!$OMP END SINGLE COPYPRIVATE (A,B,/XY/)
END SUBROUTINE INIT
```
In this example, assume that the input must be performed by the master thread. Since the 
**master** construct does not support the **copyprivate** clause, it cannot broadcast the 
input value that is read. However, **copyprivate** is used to broadcast an address where 
the input value is stored.

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

float read_next( ) {
    float * tmp;
    float return_val;

    #pragma omp single copyprivate(tmp)
    { 
        tmp = (float *) malloc(sizeof(float));
    } /* copies the pointer only */

    #pragma omp master
    { 
        scanf("%f", tmp);
    }

    #pragma omp barrier return_val = *tmp;
    #pragma omp barrier

    #pragma omp single nowait
    { 
        free(tmp);
    }

    return return_val;
}  
```

---

**Example 38.2c**
Example 38.2f

```fortran
REAL FUNCTION READ_NEXT()
    REAL, POINTER :: TMP

!$OMP SINGLE
    ALLOCATE (TMP)
!$OMP END SINGLE COPYPRIVATE (TMP)  ! copies the pointer only

!$OMP MASTER
    READ (11) TMP
!$OMP END MASTER

!$OMP BARRIER
    READ_NEXT = TMP

!$OMP BARRIER
!$OMP SINGLE
    DEALLOCATE (TMP)
!$OMP END SINGLE NOWAIT

END FUNCTION READ_NEXT
```

Suppose that the number of lock variables required within a parallel region cannot easily be determined prior to entering it. The copyprivate clause can be used to provide access to shared lock variables that are allocated within that parallel region.

Example 38.3c

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

omp_lock_t *new_lock()
{
    omp_lock_t *lock_ptr;

    #pragma omp single copyprivate(lock_ptr)
    {
        lock_ptr = (omp_lock_t *) malloc(sizeof(omp_lock_t));
        omp_init_lock( lock_ptr );
    }

    return lock_ptr;
}
```
Example 38.3f

```fortran
FUNCTION NEW_LOCK()
    USE OMP_LIB       ! or INCLUDE "omp_lib.h"
    INTEGER(OMP_LOCK_KIND), POINTER :: NEW_LOCK

    !$OMP SINGLE
    ALLOCATE(NEW_LOCK)
    CALL OMP_INIT_LOCK(NEW_LOCK)
    !$OMP END SINGLE COPYPRIVATE(NEW_LOCK)
END FUNCTION NEW_LOCK
```

Note that the effect of the `copyprivate` clause on a variable with the `allocatable` attribute is different than on a variable with the `pointer` attribute. The value of A is copied (as if by intrinsic assignment) and the pointer B is copied (as if by pointer assignment) to the corresponding list items in the other implicit tasks belonging to the parallel region.

Example 38.4f

```fortran
SUBROUTINE S(N)
    INTEGER N

    REAL, DIMENSION(:), ALLOCATABLE :: A
    REAL, DIMENSION(:), POINTER :: B

    ALLOCATE (A(N))
    !$OMP SINGLE
    ALLOCATE (B(N))
    READ (11) A,B
    !$OMP END SINGLE COPYPRIVATE(A,B)

    !$OMP BARRIER
    !$OMP SINGLE
    DEALLOCATE (B)
    !$OMP END SINGLE NOWAIT
END SUBROUTINE S
```
The following example of loop construct nesting is conforming because the inner and outer loop regions bind to different parallel regions:

```c/c++
void work(int i, int j) {}

void good_nesting(int n)
{
    int i, j;
    #pragma omp parallel default(shared)
    {
        #pragma omp for
        for (i=0; i<n; i++) {
            #pragma omp parallel shared(i, n)
            {
                #pragma omp for
                for (j=0; j < n; j++)
                    work(i, j);
            }
        }
    }
}
```

Example 39.1c
Example 39.1f

```fortran
SUBROUTINE WORK(I, J)
INTEGER I, J
END SUBROUTINE WORK

SUBROUTINE GOOD_NESTING(N)
INTEGER N

INTEGER I
!
$OMP PARALLEL DEFAULT(SHARED)
!
$OMP DO
!
$OMP PARALLEL SHARED(I,N)
!
$OMP DO
!
$OMP END PARALLEL
!
$OMP END PARALLEL
END SUBROUTINE GOOD_NESTING
```
The following variation of the preceding example is also conforming:

```
Example 39.2c

```
Example 39.2f

```fortran
SUBROUTINE WORK(I, J)
INTEGER I, J
END SUBROUTINE WORK

SUBROUTINE WORK1(I, N)
INTEGER J
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
DO J = 1, N
   CALL WORK(I,J)
END DO
!$OMP END PARALLEL
END SUBROUTINE WORK1

SUBROUTINE GOOD_NESTING2(N)
INTEGER N
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
DO I = 1, N
   CALL WORK1(I, N)
END DO
!$OMP END PARALLEL
END SUBROUTINE GOOD_NESTING2
```
Restrictions on Nesting of Regions

The examples in this section illustrate the region nesting rules.

The following example is non-conforming because the inner and outer loop regions are closely nested:

```c
void work(int i, int j) {}
void wrong1(int n)
{
    #pragma omp parallel default(shared)
    {
        int i, j;
        #pragma omp for
        for (i=0; i<n; i++) {
            /* incorrect nesting of loop regions */
            #pragma omp for
            for (j=0; j<n; j++)
                work(i, j);
        }
    }
}
```
Example 40.1f

```fortran
SUBROUTINE WORK(I, J)
INTEGER I, J
END SUBROUTINE WORK

SUBROUTINE WRONG1(N)
INTEGER N
INTEGER I,J
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
DO I = 1, N
!$OMP DO  ! incorrect nesting of loop regions
DO J = 1, N
CALL WORK(I,J)
END DO
END DO
!$OMP END PARALLEL
END SUBROUTINE WRONG1
```

The following orphaned version of the preceding example is also non-conforming:

Example 40.2c

```c
void work(int i, int j) {}
void work1(int i, int n)
{
    int j;
    /* incorrect nesting of loop regions */
    #pragma omp for
    for (j=0; j<n; j++)
    work(i, j);
}

void wrong2(int n)
{
    #pragma omp parallel default(shared)
    {
        int i;
        #pragma omp for
        for (i=0; i<n; i++)
        work1(i, n);
    }
}
Example 40.2f

```fortran
SUBROUTINE WORK1(I,N)
  INTEGER I, N
  INTEGER J
  !$OMP DO ! incorrect nesting of loop regions
  DO J = 1, N
    CALL WORK(I,J)
  END DO
END SUBROUTINE WORK1

SUBROUTINE WRONG2(N)
  INTEGER N
  INTEGER I
  !$OMP PARALLEL DEFAULT(SHARED)
  !$OMP DO
  DO I = 1, N
    CALL WORK1(I,N)
  END DO
  !$OMP END PARALLEL
END SUBROUTINE WRONG2
```

The following example is non-conforming because the loop and `single` regions are closely nested:

Example 40.3c

```c
void work(int i, int j) {} 
void wrong3(int n) 
{ 
  #pragma omp parallel default(shared)
  { 
    int i;
    #pragma omp for 
    for (i=0; i<n; i++) {
      /* incorrect nesting of regions */
      #pragma omp single 
      work(i, 0);
    }
  }
}
```
Example 40.3f

```fortran
SUBROUTINE WRONG3(N)
 INTEGER N
 INTEGER I
 !$OMP PARALLEL DEFAULT(SHARED)
 !$OMP DO
 DO I = 1, N
 !$OMP SINGLE ! incorrect nesting of regions
 CALL WORK(I, 1)
 !$OMP END SINGLE
 END DO
 !$OMP END PARALLEL
 END SUBROUTINE WRONG3
```

The following example is non-conforming because a `barrier` region cannot be closely nested inside a loop region:

Example 40.4c

```c
void work(int i, int j) {}
void wrong4(int n)
{
    #pragma omp parallel default(shared)
    {
        int i;
        #pragma omp for
        for (i=0; i<n; i++) {
            work(i, 0);
            /* incorrect nesting of barrier region in a loop region */
            #pragma omp barrier
            work(i, 1);
        }
    }
}
```
Example 40.4f

```fortran
SUBROUTINE WRONG4(N)
  INTEGER N
  INTEGER I
  !$OMP PARALLEL DEFAULT(SHARED)
  !$OMP DO
  DO I = 1, N
    CALL WORK(I, 1)
    ! incorrect nesting of barrier region in a loop region
    !$OMP BARRIER
    CALL WORK(I, 2)
  END DO
  !$OMP END PARALLEL
END SUBROUTINE WRONG4
```

The following example is non-conforming because the `barrier` region cannot be closely nested inside the `critical` region. If this were permitted, it would result in deadlock due to the fact that only one thread at a time can enter the `critical` region:

Example 40.5c

```c
void work(int i, int j) {}
void wrong5(int n)
{
  #pragma omp parallel
  {
    #pragma omp critical
    {
      work(n, 0);
      /* incorrect nesting of barrier region in a critical region */
      #pragma omp barrier
      work(n, 1);
    }
  }
}
```
Example 40.5f

```fortran
SUBROUTINE WRONG5(N)
    INTEGER N
    !$OMP PARALLEL DEFAULT(SHARED)
    !$OMP CRITICAL
    CALL WORK(N, 1)
    ! incorrect nesting of barrier region in a critical region
    !$OMP BARRIER
    CALL WORK(N, 2)
    !$OMP END CRITICAL
    !$OMP END PARALLEL
END SUBROUTINE WRONG5
```

The following example is non-conforming because the `barrier` region cannot be closely nested inside the `single` region. If this were permitted, it would result in deadlock due to the fact that only one thread executes the `single` region:

Example 40.6c

```c++
void work(int i, int j) {}
void wrong6(int n)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            work(n, 0);
            /* incorrect nesting of barrier region in a single region */
            #pragma omp barrier
            work(n, 1);
        }
    }
}
```
Example 40.6f

SUBROUTINE WRONG6(N)
  INTEGER N

  !$OMP PARALLEL DEFAULT(SHARED)
  !$OMP SINGLE
    CALL WORK(N,1)
  !$OMP BARRIER
  CALL WORK(N,2)
  !$OMP END SINGLE
  !$OMP END PARALLEL
END SUBROUTINE WRONG6
The **omp_set_dynamic** and **omp_set_num_threads** Routines

Some programs rely on a fixed, prespecified number of threads to execute correctly. Because the default setting for the dynamic adjustment of the number of threads is implementation defined, such programs can choose to turn off the dynamic threads capability and set the number of threads explicitly to ensure portability. The following example shows how to do this using **omp_set_dynamic**, and **omp_set_num_threads**.

In this example, the program executes correctly only if it is executed by 16 threads. If the implementation is not capable of supporting 16 threads, the behavior of this example is implementation defined. Note that the number of threads executing a **parallel** region remains constant during the region, regardless of the dynamic threads setting. The dynamic threads mechanism determines the number of threads to use at the start of the **parallel** region and keeps it constant for the duration of the region.

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

void do_by_16(float *x, int iam, int ipoints) {}

void dynthreads(float *x, int npoints)
{
    int iam, ipoints;

    omp_set_dynamic(0);
    omp_set_num_threads(16);

    #pragma omp parallel shared(x, npoints) private(iam, ipoints)
    {
        if (omp_get_num_threads() != 16)
            abort();

        iam = omp_get_thread_num();
        ipoints = npoints/16;
        do_by_16(x, iam, ipoints);
    }
}
```
Example 41.1f

**Fortran**

```fortran
SUBROUTINE DO_BY_16(X, IAM, IPOINTS)
  REAL X(*)
  INTEGER IAM, IPOINTS
END SUBROUTINE DO_BY_16

SUBROUTINE DYNTHREADS(X, NPOINTS)
  INCLUDE "omp_lib.h" ! or USE OMP_LIB
  INTEGER NPOINTS
  REAL X(NPOINTS)
  INTEGER IAM, IPOINTS
  CALL OMP_SET_DYNAMIC(.FALSE.)
  CALL OMP_SET_NUM_THREADS(16)
  !$OMP PARALLEL SHARED(X,NPOINTS) PRIVATE(IAM, IPOINTS)
  IF (OMP_GET_NUM_THREADS() .NE. 16) THEN
    STOP
  ENDIF
  IAM = OMP_GET_THREAD_NUM()
  IPOINTS = NPOINTS/16
  CALL DO_BY_16(X,IAM,IPOINTS)
  !$OMP END PARALLEL
END SUBROUTINE DYNTHREADS
```
The `omp_get_num_threads` Routine

In the following example, the `omp_get_num_threads` call returns 1 in the sequential part of the code, so `np` will always be equal to 1. To determine the number of threads that will be deployed for the `parallel` region, the call should be inside the `parallel` region.

```
#include <omp.h>
void work(int i);

void incorrect()
{
    int np, i;

    np = omp_get_num_threads(); /* misplaced */
    #pragma omp parallel for schedule(static)
        for (i=0; i < np; i++)
            work(i);
}
```

Example 42.1c
Example 42.1f

```fortran
SUBROUTINE WORK(I)
  INTEGER I
  I = I + 1
END SUBROUTINE WORK

SUBROUTINE INCORRECT()
  INCLUDE "omp_lib.h" ! or USE OMP_LIB
  INTEGER I, NP
  NP = OMP_GET_NUM_THREADS()   ! misplaced: will return 1
  !$OMP PARALLEL DO SCHEDULE(STATIC)
  DO I = 0, NP-1
    CALL WORK(I)
  ENDDO
  !$OMP END PARALLEL DO
END SUBROUTINE INCORRECT
```

The following example shows how to rewrite this program without including a query for the number of threads:

Example 42.2c

```c++
#include <omp.h>
void work(int i);

void correct()
{
  int i;
  #pragma omp parallel private(i)
  {
    i = omp_get_thread_num();
    work(i);
  }
}
```
Example 42.2f

```fortran
SUBROUTINE WORK(I)
    INTEGER I
    I = I + 1
END SUBROUTINE WORK

SUBROUTINE CORRECT()
    INCLUDE "omp_lib.h"    ! or USE OMP_LIB
    INTEGER I

    !$OMP PARALLEL PRIVATE(I)
        I = OMP_GET_THREAD_NUM()
        CALL WORK(I)
    !$OMP END PARALLEL

END SUBROUTINE CORRECT
```
The `omp_init_lock` Routine

The following example demonstrates how to initialize an array of locks in a parallel region by using `omp_init_lock`.

**Example 43.1c**

```c
#include <omp.h>

omp_lock_t *new_locks()
{
    int i;
    omp_lock_t *lock = new omp_lock_t[1000];

    #pragma omp parallel for private(i)
    for (i=0; i<1000; i++)
    {
        omp_init_lock(&lock[i]);
    }
    return lock;
}
```

**Example 43.1f**

```fortran
FUNCTION NEW_LOCKS()
    USE OMP_LIB ! or INCLUDE "omp_lib.h"
    INTEGER(OMP_LOCK_KIND), DIMENSION(1000) :: NEW_LOCKS

    INTEGER I

    !$OMP PARALLEL DO PRIVATE(I)
    DO I=1,1000
        CALL OMP_INIT_LOCK(NEW_LOCKS(I))
    END DO
    !$OMP END PARALLEL DO

END FUNCTION NEW_LOCKS
```
Ownership of Locks

Ownership of locks has changed since OpenMP 2.5. In OpenMP 2.5, locks are owned by threads; so a lock released by the `omp_unset_lock` routine must be owned by the same thread executing the routine. With OpenMP 3.0, locks are owned by task regions; so a lock released by the `omp_unset_lock` routine in a task region must be owned by the same task region.

This change in ownership requires extra care when using locks. The following program is conforming in OpenMP 2.5 because the thread that releases the lock `lck` in the parallel region is the same thread that acquired the lock in the sequential part of the program (master thread of parallel region and the initial thread are the same). However, it is not conforming in OpenMP 3.0 and 3.1, because the task region that releases the lock `lck` is different from the task region that acquires the lock.

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

int main()
{
  int x;
  omp_lock_t lck;

  omp_init_lock (&lck);
  omp_set_lock (&lck);
  x = 0;

  #pragma omp parallel shared (x)
  {
    #pragma omp master
    {
      x = x + 1;
      omp_unset_lock (&lck);
    }

    /* Some more stuff. */
  }
  omp_destroy_lock (&lck);
  return 0;
}
```
**Example 44.1f**

```fortran
program lock
  use omp_lib
  integer :: x
  integer(kind=omp_lock_kind) :: lck
  call omp_init_lock(lck)
  call omp_set_lock(lck)
  x = 0

!$omp parallel shared (x)
!$omp master
  x = x + 1
  call omp_unset_lock(lck)
!$omp end master

! Some more stuff.
!$omp end parallel
  call omp_destroy_lock(lck)
end
```
45 Simple Lock Routines

In the following example, the lock routines cause the threads to be idle while waiting for
entry to the first critical section, but to do other work while waiting for entry to the
second. The `omp_set_lock` function blocks, but the `omp_test_lock` function does
not, allowing the work in `skip` to be done.

```c
#include <stdio.h>
#include <omp.h>
void skip(int i) {}
void work(int i) {}
int main()
{
  omp_lock_t lck;
  int id;
  omp_init_lock(&lck);

  #pragma omp parallel shared(lck) private(id)
  {
    id = omp_get_thread_num();

    omp_set_lock(&lck);
    /* only one thread at a time can execute this printf */
    printf("My thread id is \%d.\n", id);
    omp_unset_lock(&lck);

    while (! omp_test_lock(&lck)) {
      skip(id); /* we do not yet have the lock,
                  so we must do something else */
    }

    work(id); /* we now have the lock
               and can do the work */
    omp_unset_lock(&lck);
  }
  omp_destroy_lock(&lck);
  return 0;
}  
```
Note that there is no need to flush the lock variable.

**Example 45.1f**

```fortran
SUBROUTINE SKIP(ID)
END SUBROUTINE SKIP

SUBROUTINE WORK(ID)
END SUBROUTINE WORK

PROGRAM SIMPLELOCK

INCLUDE "omp_lib.h" ! or USE OMP_LIB

INTEGER(OMP_LOCK_KIND) LCK
INTEGER ID

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 PROGRAM SIMPLELOCK
```
46 Nestable Lock Routines

The following example demonstrates how a nestable lock can be used to synchronize updates both to a whole structure and to one of its members.

```
Example 46.1c

#include <omp.h>
typedef struct {
    int a,b;
    omp_nest_lock_t lck; } pair;

int work1();
int work2();
int work3();
void incr_a(pair *p, int a)
{
    /* Called only from incr_pair, no need to lock. */
    p->a += a;
}
void incr_b(pair *p, int b)
{
    /* Called both from incr_pair and elsewhere, */
    /* so need a nestable lock. */
    omp_set_nest_lock(&p->lck);
    p->b += b;
    omp_unset_nest_lock(&p->lck);
}
void incr_pair(pair *p, int a, int b)
{
    omp_set_nest_lock(&p->lck);
    incr_a(p, a);
    incr_b(p, b);
    omp_unset_nest_lock(&p->lck);
}
void nestlock(pair *p)
{
    #pragma omp parallel sections
    {
        #pragma omp section
        incr_pair(p, work1(), work2());
        #pragma omp section
        incr_b(p, work3());
    }
}
```
Example 46.1f

```fortran
MODULE DATA
  USE OMP_LIB, ONLY: OMP_NEST_LOCK_KIND
  TYPE LOCKED_PAIR
    INTEGER A
    INTEGER B
    INTEGER (OMP_NEST_LOCK_KIND) LCK
  END TYPE
END MODULE DATA

SUBROUTINE INCR_A(P, A)
  ! called only from INCR_PAIR, no need to lock
  USE DATA
  TYPE(LOCKED_PAIR) :: P
  INTEGER A
  P%A = P%A + A
END SUBROUTINE INCR_A

SUBROUTINE INCR_B(P, B)
  ! called from both INCR_PAIR and elsewhere,
  ! so we need a nestable lock
  USE OMP_LIB       ! or INCLUDE "omp_lib.h"
  USE DATA
  TYPE(LOCKED_PAIR) :: P
  INTEGER B
  CALL OMP_SET_NEST_LOCK(P%LCK)
  P%B = P%B + B
  CALL OMP_UNSET_NEST_LOCK(P%LCK)
END SUBROUTINE INCR_B

SUBROUTINE INCR_PAIR(P, A, B)
  USE OMP_LIB        ! or INCLUDE "omp_lib.h"
  USE DATA
  TYPE(LOCKED_PAIR) :: P
  INTEGER A
  INTEGER B
  CALL OMP_SET_NEST_LOCK(P%LCK)
  CALL INCR_A(P, A)
  CALL INCR_B(P, B)
  CALL OMP_UNSET_NEST_LOCK(P%LCK)
END SUBROUTINE INCR_PAIR

SUBROUTINE NESTLOCK(P)
  USE OMP_LIB        ! or INCLUDE "omp_lib.h"
  USE DATA
  TYPE(LOCKED_PAIR) :: P
  INTEGER WORK1, WORK2, WORK3
  EXTERNAL WORK1, WORK2, WORK3
END SUBROUTINE NESTLOCK
```
!$OMP PARALLEL SECTIONS
    !$OMP SECTION
        CALL INCR_PAIR(P, WORK1(), WORK2())
    !$OMP SECTION
        CALL INCR_B(P, WORK3())
    !$OMP END PARALLEL SECTIONS

END SUBROUTINE NESTLOCK