OpenMP Application Programming Interface

Examples

Version 4.0.2 – March 2015

Source codes for OpenMP 4.0.2 Examples can be downloaded from github.

This page intentionally left blank

Contents

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 first release of the OpenMP Examples reflects the OpenMP Version 4.0 specifications. Additional examples are being developed and will be published in future releases of this document.

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.

The latest source codes for OpenMP Examples can be downloaded from the **sources** directory at https://github.com/OpenMP/Examples. The codes for this OpenMP 4.0.2 Examples document have the tag *v4.0.2*.

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

Examples

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

C / C++

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

C / C++

CHAPTER 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.

C / C++

Example ploop.1c

```
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;
}
```

C / C++

Fortran

Example ploop.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
```

Fortran

CHAPTER 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.

C / C++

Example mem_model.1c

```
#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;
}
```

C / C++

Fortran

Example mem_model.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
```

Fortran

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.

C / C++

Example mem_model.2c

```
#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;
}
```

C / C++

Fortran

Example mem_model.2f

```
      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)

           !Values DATA will be 42, value of FLAG still undefined */
           PRINT *, 'FLAG=', FLAG, ' DATA=', DATA
        ENDIF
      !$OMP END PARALLEL
      END
```

Fortran

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.

C / C++

1 *Example mem_model.3c*

```
#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;
}
```

C / C++

Fortran

1 *Example mem_model.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
```

Fortran

CHAPTER 3

Conditional Compilation

C / C++

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

Example cond_comp.1c

```
#include <stdio.h>

int main()
{

# ifdef _OPENMP
    printf("Compiled by an OpenMP-compliant implementation.\n");
# endif

    return 0;
}
```

C / C++

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 cond_comp.1f

```
      PROGRAM EXAMPLE

C234567890
!$    PRINT *, "Compiled by an OpenMP-compliant implementation."

      END PROGRAM EXAMPLE
```

Fortran

CHAPTER 4

Internal Control Variables (ICVs)

According to Section 2.3 of the OpenMP 4.0 specification, 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.

C / C++

Example icv.1c

```
#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;
   }
```

C / C++

Fortran

Example icv.1f

```
      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
```

Fortran

CHAPTER 5

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++

Example parallel.1c

```
#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;
}
```

C / C++

Fortran

1 *Example parallel.1f*

```
      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
```

Fortran

CHAPTER 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 nthrs_nesting.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());
           }
        }
        #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;
  }
```

C / C++

Fortran

1 *Example nthrs_nesting.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
```

Fortran

CHAPTER 7

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 nthrs_dynamic.1c

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

C / C++

Fortran

Example nthrs_dynamic.1f

```
      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
```

Fortran

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.

C / C++

Example nthrs_dynamic.2c

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

C / C++

Fortran

Example nthrs_dynamic.2f

```
      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
```

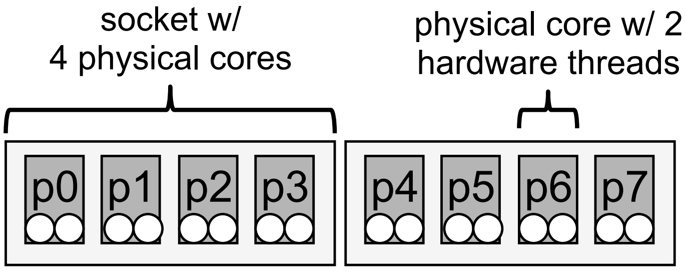
Fortran

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.

CHAPTER 8

The `proc_bind` Clause

The following examples demonstrate how to use the **`proc_bind`** clause to control the thread binding for a team of threads in a **`parallel`** region. The machine architecture is depicted in the figure below. It consists of two sockets, each equipped with a quad-core processor and configured to execute two hardware threads simultaneously on each core. These examples assume a contiguous core numbering starting from 0, such that the hardware threads 0,1 form the first physical core.



The following equivalent place list declarations consist of eight places (which we designate as p0 to p7):

`OMP_PLACES="{0,1},{2,3},{4,5},{6,7},{8,9},{10,11},{12,13},{14,15}"`

or

`OMP_PLACES="{0:2}:8:2"`

8.1 Spread Affinity Policy

The following example shows the result of the **`spread`** affinity policy on the partition list when the number of threads is less than or equal to the number of places in the parent's place partition, for the machine architecture depicted above. Note that the threads are bound to the first place of each subpartition.

C / C++

Example affinity.1c

```
void work();
int main()
{
#pragma omp parallel proc_bind(spread) num_threads(4)
   {
      work();
   }
   return 0;
}
```

C / C++

Fortran

Example affinity.1f

```
      PROGRAM EXAMPLE
!$OMP PARALLEL PROC_BIND(SPREAD) NUM_THREADS(4)
      CALL WORK()
!$OMP END PARALLEL
      END PROGRAM EXAMPLE
```

Fortran

It is unspecified on which place the master thread is initially started. If the master thread is initially started on p0, the following placement of threads will be applied in the parallel region:

- thread 0 executes on p0 with the place partition p0,p1
- thread 1 executes on p2 with the place partition p2,p3
- thread 2 executes on p4 with the place partition p4,p5
- thread 3 executes on p6 with the place partition p6,p7

If the master thread would initially be started on p2, the placement of threads and distribution of the place partition would be as follows:

- thread 0 executes on p2 with the place partition p2,p3
- thread 1 executes on p4 with the place partition p4,p5
- thread 2 executes on p6 with the place partition p6,p7
- thread 3 executes on p0 with the place partition p0,p1

The following example illustrates the **spread** thread affinity policy when the number of threads is greater than the number of places in the parent's place partition.

Let *T* be the number of threads in the team, and *P* be the number of places in the parent's place partition. The first *T/P* threads of the team (including the master thread) execute on the parent's place. The next *T/P* threads execute on the next place in the place partition, and so on, with wrap around.

C / C++

Example affinity.2c

```
void work();
void foo()
{
  #pragma omp parallel num_threads(16) proc_bind(spread)
  {
    work();
  }
}
```

C / C++

Fortran

Example affinity.2f

```
subroutine foo
!$omp parallel num_threads(16) proc_bind(spread)
      call work()
!$omp end parallel
end subroutine
```

Fortran

It is unspecified on which place the master thread is initially started. If the master thread is initially started on p0, the following placement of threads will be applied in the parallel region:

- threads 0,1 execute on p0 with the place partition p0
- threads 2,3 execute on p1 with the place partition p1
- threads 4,5 execute on p2 with the place partition p2
- threads 6,7 execute on p3 with the place partition p3
- threads 8,9 execute on p4 with the place partition p4
- threads 10,11 execute on p5 with the place partition p5
- threads 12,13 execute on p6 with the place partition p6
- threads 14,15 execute on p7 with the place partition p7

If the master thread would initially be started on p2, the placement of threads and distribution of the place partition would be as follows:

- threads 0,1 execute on p2 with the place partition p2

- threads 2,3 execute on p3 with the place partition p3
- threads 4,5 execute on p4 with the place partition p4
- threads 6,7 execute on p5 with the place partition p5
- threads 8,9 execute on p6 with the place partition p6
- threads 10,11 execute on p7 with the place partition p7
- threads 12,13 execute on p0 with the place partition p0
- threads 14,15 execute on p1 with the place partition p1

8.2 Close Affinity Policy

The following example shows the result of the **close** affinity policy on the partition list when the number of threads is less than or equal to the number of places in parent's place partition, for the machine architecture depicted above. The place partition is not changed by the **close** policy.

C / C++

Example affinity.3c

```
void work();
int main()
{
#pragma omp parallel proc_bind(close) num_threads(4)
   {
      work();
   }
   return 0;
}
```

C / C++

Fortran

Example affinity.3f

```
      PROGRAM EXAMPLE
!$OMP PARALLEL PROC_BIND(CLOSE) NUM_THREADS(4)
      CALL WORK()
!$OMP END PARALLEL
      END PROGRAM EXAMPLE
```

Fortran

It is unspecified on which place the master thread is initially started. If the master thread is initially started on p0, the following placement of threads will be applied in the **parallel** region:

- thread 0 executes on p0 with the place partition p0-p7
- thread 1 executes on p1 with the place partition p0-p7
- thread 2 executes on p2 with the place partition p0-p7
- thread 3 executes on p3 with the place partition p0-p7

If the master thread would initially be started on p2, the placement of threads and distribution of the place partition would be as follows:

- thread 0 executes on p2 with the place partition p0-p7
- thread 1 executes on p3 with the place partition p0-p7
- thread 2 executes on p4 with the place partition p0-p7
- thread 3 executes on p5 with the place partition p0-p7

The following example illustrates the **close** thread affinity policy when the number of threads is greater than the number of places in the parent's place partition.

Let *T* be the number of threads in the team, and *P* be the number of places in the parent's place partition. The first *T/P* threads of the team (including the master thread) execute on the parent's place. The next *T/P* threads execute on the next place in the place partition, and so on, with wrap around. The place partition is not changed by the **close** policy.

C / C++

Example affinity.4c

```
void work();
void foo()
{
  #pragma omp parallel num_threads(16) proc_bind(close)
  {
    work();
  }
}
```

C / C++

Fortran

Example affinity.4f

```
subroutine foo
!$omp parallel num_threads(16) proc_bind(close)
      call work()
!$omp end parallel
end subroutine
```

Fortran

It is unspecified on which place the master thread is initially started. If the master thread is initially running on p0, the following placement of threads will be applied in the parallel region:

- threads 0,1 execute on p0 with the place partition p0-p7
- threads 2,3 execute on p1 with the place partition p0-p7
- threads 4,5 execute on p2 with the place partition p0-p7
- threads 6,7 execute on p3 with the place partition p0-p7
- threads 8,9 execute on p4 with the place partition p0-p7
- threads 10,11 execute on p5 with the place partition p0-p7
- threads 12,13 execute on p6 with the place partition p0-p7
- threads 14,15 execute on p7 with the place partition p0-p7

If the master thread would initially be started on p2, the placement of threads and distribution of the place partition would be as follows:

- threads 0,1 execute on p2 with the place partition p0-p7
- threads 2,3 execute on p3 with the place partition p0-p7
- threads 4,5 execute on p4 with the place partition p0-p7
- threads 6,7 execute on p5 with the place partition p0-p7
- threads 8,9 execute on p6 with the place partition p0-p7
- threads 10,11 execute on p7 with the place partition p0-p7
- threads 12,13 execute on p0 with the place partition p0-p7
- threads 14,15 execute on p1 with the place partition p0-p7

8.3 Master Affinity Policy

The following example shows the result of the **`master`** affinity policy on the partition list for the machine architecture depicted above. The place partition is not changed by the master policy.

C / C++

Example affinity.5c

```c
void work();
int main()
{
#pragma omp parallel proc_bind(master) num_threads(4)
   {
      work();
   }
   return 0;
}
```

C / C++

Fortran

Example affinity.5f

```fortran
      PROGRAM EXAMPLE
!$OMP PARALLEL PROC_BIND(MASTER) NUM_THREADS(4)
      CALL WORK()
!$OMP END PARALLEL
      END PROGRAM EXAMPLE
```

Fortran

It is unspecified on which place the master thread is initially started. If the master thread is initially running on p0, the following placement of threads will be applied in the parallel region:

- threads 0-3 execute on p0 with the place partition p0-p7

If the master thread would initially be started on p2, the placement of threads and distribution of the place partition would be as follows:

- threads 0-3 execute on p2 with the place partition p0-p7

CHAPTER 9

Fortran Restrictions on the do Construct

Fortran

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 fort_do.1f

```
      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

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

!$OMP   DO
        DO 300 I = 1,10
          DO 300 J = 1,10
            CALL WORK(I,J)
```

```
300     CONTINUE
!$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 fort_do.2f

```
      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

CHAPTER 10

Fortran Private Loop Iteration Variables

Fortran

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 Section 2.7.1 and Section 2.14.1 of the OpenMP 4.0 specification). In the following example of a sequential loop in a **parallel** construct the loop iteration variable *I* will be private.

Example fort_loopvar.1f

```
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 fort_loopvar.2f

```
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.'
!$OMP END SINGLE
!$OMP END PARALLEL

END SUBROUTINE PLOOP_2
```

Note however that the use of shared loop iteration variables can easily lead to race conditions.

Fortran

CHAPTER 11

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:

C / C++

Example nowait.1c

```
#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]);
  }
}
```

C / C++

Fortran

Example nowait.1f

```
      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
```

Fortran

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).

C / C++

1 *Example nowait.2c*

```

#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];
   }
}
```

C / C++

Fortran

2 *Example nowait.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
```

Fortran

CHAPTER 12

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.

The variable **`j`** can be omitted from the **`private`** clause when the **`collapse`** clause is used since it is implicitly private. However, if the **`collapse`** clause is omitted then **`j`** will be shared if it is omitted from the **`private`** clause. In either case, **`k`** is implicitly private and could be omitted from the **`private`** clause.

C / C++

Example collapse.1c

```
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);
}
```

C / C++

Fortran

Example collapse.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
```

Fortran

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 **klast** and **jlast** are **lastprivate**, their values are assigned by the sequentially last iteration of the collapsed **k** and **j** loop. This example prints: **2 3**.

C / C++

Example collapse.2c

```
#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);
    }
}
```

C / C++

Fortran

Example collapse.2f

```
      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
```

Fortran

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 Section 2.12.8 of the OpenMP 4.0 specification, 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
```

C / C++

Example collapse.3c

```
#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\n", omp_get_thread_num(), k, j);
            /* end ordered */
            work(a,j,k);
         }
   }
}
```

C / C++

Fortran

Example collapse.3f

```
      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
```

Fortran

CHAPTER 13

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.

C / C++

Example psections.1c

```
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();
  }
}
```

C / C++

Fortran

1 *Example psections.1f*

```
      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
```

Fortran

CHAPTER 14

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.

C / C++

Example fpriv_sections.1c

```
#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 0;
}
```

C / C++

Fortran

Example fpriv_sections.1f

```
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
```

Fortran

CHAPTER 15

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.

C / C++

Example single.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();
  }
}
```

C / C++

Fortran

1 *Example single.1f*

```
      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
```

Fortran

CHAPTER 16

The `task` and `taskwait` 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.

C / C++

Example tasking.1c

```
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);
}
```

C / C++

Fortran

Example tasking.1f

```
       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
          CALL process ( P )
        END SUBROUTINE
```

Fortran

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.

C / C++

Example tasking.2c

```
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);
}
```

C / C++

Fortran

Example tasking.2f

```
       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
           CALL process ( P )
        END SUBROUTINE
```

Fortran

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 / C++

Example tasking.3c

```
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;
                   }
               }
        }
}
```

C / C++

Fortran

1 *Example tasking.3f*

```
      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
```

Fortran

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.

C / C++

Example tasking.4c

```
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)
         j=fib(n-2);
      #pragma omp taskwait
         return i+j;
   }
}
```

C / C++

Fortran

Example tasking.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
```

Fortran

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.

C / C++

Example tasking.5c

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

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

C / C++

Fortran

Example tasking.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.

C / C++

Example tasking.6c

```
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];
extern void process(double);
int main() {
#pragma omp parallel
  {
    #pragma omp single
    {
      int i;
      #pragma omp task untied
      // i is firstprivate, item is shared
      {
         for (i=0; i<LARGE_NUMBER; i++)
             #pragma omp task
```

```
                    process(item[i]);
          }
        }
      }
    return 0;
  }
```

C / C++

Fortran

Example tasking.6f

```
      real*8 item(10000000)
!$omp parallel
!$omp single
!$omp task untied
      ! loop iteration variable i is private
      do i=1,10000000
!$omp task ! i is firstprivate, item is shared
         call process(item(i))
!$omp end task
      end do
!$omp end task
!$omp end single
!$omp end parallel
      end
```

Fortran

The following two examples demonstrate how the scheduling rules illustrated in Section 2.11.3 of the OpenMP 4.0 specification 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.

C / C++

Example tasking.7c

```

int tp;
#pragma omp threadprivate(tp)
int var;
void work()
{
#pragma omp task
    {
        /* do work here */
#pragma omp task
        {
            tp = 1;
            /* do work here */
#pragma omp task
            {
                /* no modification of tp */
            }
            var = tp; //value of tp can be 1 or 2
        }
        tp = 2;
    }
}
```

C / C++

Fortran

Example tasking.7f

```
      module example
      integer tp
!$omp threadprivate(tp)
      integer var
      contains
      subroutine work
!$omp task
         ! do work here
!$omp task
         tp = 1
         ! do work here
!$omp task
           ! no modification of tp
!$omp end task
         var = tp    ! value of var can be 1 or 2
!$omp end task
        tp = 2
```

```
!$omp end task
      end subroutine
      end module
```

Fortran

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.

C / C++

Example tasking.8c

```

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
        }
    }
}
```

C / C++

Fortran

Example tasking.8f

```
       module example
       integer tp
!$omp threadprivate(tp)
       integer var
       contains
       subroutine work
!$omp parallel
           ! do work here
!$omp task
           tp = tp + 1
           ! do work here
!$omp task
             ! 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 Section 2.11.3 of the OpenMP 4.0 specification 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 tasking.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
        }
    }
}
```

C / C++

Fortran

Example tasking.9f

```
       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
```

Fortran

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.

C / C++

1 *Example tasking.10c*

```
#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
        // Task Scheduling Point 1
                { /* do work here */ }
                omp_unset_lock(&lock);
            }
        }
    }
    omp_destroy_lock(&lock);
}
```

C / C++

Fortran

2 *Example tasking.10f*

```
       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
                            !$omp end task
                      call omp_unset_lock(lock)
                !$omp end task
      end do
!$omp end parallel
      call omp_destroy_lock(lock)
      end subroutine
      end module
```

Fortran

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**).

C / C++

Example tasking.11c

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

C / C++

Fortran

Example tasking.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.

C / C++

Example tasking.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++

Fortran

Example tasking.12f

```
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
```

Fortran

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.

C / C++

Example tasking.13c

```
#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
    }
```

C / C++

Fortran

1 *Example tasking.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
```

Fortran

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 tasking.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();
      }
   }
}
```

C / C++

Fortran

Example tasking.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
end subroutine
```

Fortran

CHAPTER 17

Task Dependences

17.1 Flow Dependence

In this example we show a simple flow dependence expressed using the **depend** clause on the **task** construct.

C / C++

Example task_dep.1c

```
#include <stdio.h>
int main()
{
   int x = 1;
   #pragma omp parallel
   #pragma omp single
   {
      #pragma omp task shared(x) depend(out: x)
         x = 2;
      #pragma omp task shared(x) depend(in: x)
         printf("x = %d\n", x);
   }
   return 0;
}
```

C / C++

Fortran

Example task_dep.1f

```
program example
   integer :: x
   x = 1
   !$omp parallel
   !$omp single
      !$omp task shared(x) depend(out: x)
         x = 2
      !$omp end task
      !$omp task shared(x) depend(in: x)
         print*, "x = ", x
      !$omp end task
   !$omp end single
   !$omp end parallel
end program
```

Fortran

The program will always print `"x = 2"`, because the **`depend`** clauses enforce the ordering of the tasks. If the **`depend`** clauses had been omitted, then the tasks could execute in any order and the program and the program would have a race condition.

17.2 Anti-dependence

In this example we show an anti-dependence expressed using the **`depend`** clause on the **`task`** construct.

C / C++

Example task_dep.2c

```
#include <stdio.h>
int main()
{
   int x = 1;
   #pragma omp parallel
   #pragma omp single
   {
      #pragma omp task shared(x) depend(in: x)
         printf("x = %d\n", x);
      #pragma omp task shared(x) depend(out: x)
         x = 2;
   }
   return 0;
}
```

C / C++

Fortran

Example task_dep.2f

```
program example
   integer :: x
   x = 1
   !$omp parallel
   !$omp single
      !$omp task shared(x) depend(in: x)
         print*, "x = ", x
      !$omp end task
      !$omp task shared(x) depend(out: x)
         x = 2
      !$omp end task
   !$omp end single
   !$omp end parallel
end program
```

Fortran

The program will always print `"x = 1"`, because the **`depend`** clauses enforce the ordering of the tasks. If the **`depend`** clauses had been omitted, then the tasks could execute in any order and the program would have a race condition.

17.3 Output Dependence

In this example we show an output dependence expressed using the **`depend`** clause on the **`task`** construct.

C / C++

Example task_dep.3c

```
#include <stdio.h>
int main()
{
   int x;
   #pragma omp parallel
   #pragma omp single
   {
      #pragma omp task shared(x) depend(out: x)
         x = 1;
      #pragma omp task shared(x) depend(out: x)
         x = 2;
      #pragma omp taskwait
      printf("x = %d\n", x);
   }
   return 0;
}
```

C / C++

Fortran

Example task_dep.3f

```
program example
   integer :: x
   !$omp parallel
   !$omp single
      !$omp task shared(x) depend(out: x)
         x = 1
      !$omp end task
      !$omp task shared(x) depend(out: x)
         x = 2
      !$omp end task
      !$omp taskwait
      print*, "x = ", x
   !$omp end single
   !$omp end parallel
end program
```

Fortran

The program will always print `"x = 2"`, because the **`depend`** clauses enforce the ordering of the tasks. If the **`depend`** clauses had been omitted, then the tasks could execute in any order and the program would have a race condition.

17.4 Concurrent Execution with Dependences

In this example we show potentially concurrent execution of tasks using multiple flow dependences expressed using the **depend** clause on the **task** construct.

C / C++

Example task_dep.4c

```
#include <stdio.h>
int main()
{
   int x = 1;
   #pragma omp parallel
   #pragma omp single
   {
      #pragma omp task shared(x) depend(out: x)
         x = 2;
      #pragma omp task shared(x) depend(in: x)
         printf("x + 1 = %d. ", x+1);
      #pragma omp task shared(x) depend(in: x)
         printf("x + 2 = %d\n", x+2);
   }
   return 0;
}
```

C / C++

Fortran

Example task_dep.4f

```
program example
   integer :: x
   x = 1
   !$omp parallel
   !$omp single
      !$omp task shared(x) depend(out: x)
         x = 2
      !$omp end task
      !$omp task shared(x) depend(in: x)
         print*, "x + 1 = ", x+1, "."
      !$omp end task
      !$omp task shared(x) depend(in: x)
         print*, "x + 2 = ", x+2, "."
      !$omp end task
   !$omp end single
   !$omp end parallel
end program
```

Fortran

The last two tasks are dependent on the first task. However there is no dependence between the last two tasks, which may execute in any order (or concurrently if more than one thread is available). Thus, the possible outputs are "x + 1 = 3. x + 2 = 4. " and "x + 2 = 4. x + 1 = 3. ". If the **depend** clauses had been omitted, then all of the tasks could execute in any order and the program would have a race condition.

17.5 Matrix multiplication

This example shows a task-based blocked matrix multiplication. Matrices are of NxN elements, and the multiplication is implemented using blocks of BSxBS elements.

C / C++

Example task_dep.5c

```
// Assume BS divides N perfectly
void matmul_depend(int N, int BS, float A[N][N], float B[N][N], float
C[N][N] )
{
   int i, j, k, ii, jj, kk;
   for (i = 0; i < N; i+=BS) {
      for (j = 0; j < N; j+=BS) {
         for (k = 0; k < N; k+=BS) {
// Note 1: i, j, k, A, B, C are firstprivate by default
// Note 2: A, B and C are just pointers
#pragma omp task private(ii, jj, kk) \
            depend ( in: A[i:BS][k:BS], B[k:BS][j:BS] ) \
            depend ( inout: C[i:BS][j:BS] )
            for (ii = i; ii < i+BS; ii++ )
               for (jj = j; jj < j+BS; jj++ )
                  for (kk = k; kk < k+BS; kk++ )
                     C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
         }
      }
   }
}
```

C / C++

Fortran

Example task_dep.5f

```
! Assume BS divides N perfectly
subroutine matmul_depend (N, BS, A, B, C)
   implicit none
   integer :: N, BS, BM
   real, dimension(N, N) :: A, B, C
   integer :: i, j, k, ii, jj, kk
   BM = BS - 1
   do i = 1, N, BS
      do j = 1, N, BS
         do k = 1, N, BS
!$omp task shared(A,B,C) private(ii,jj,kk) & ! I,J,K are firstprivate by default
!$omp depend ( in: A(i:i+BM, k:k+BM), B(k:k+BM, j:j+BM) ) &
!$omp depend ( inout: C(i:i+BM, j:j+BM) )
            do ii = i, i+BM
               do jj = j, j+BM
                  do kk = k, k+BM
                     C(jj,ii) = C(jj,ii) + A(kk,ii) * B(jj,kk)
                  end do
               end do
            end do
!$omp end task
         end do
      end do
   end do
end subroutine
```

Fortran

CHAPTER 18

The `taskgroup` Construct

In this example, tasks are grouped and synchronized using the **taskgroup** construct.

Initially, one task (the task executing the **start_background_work()** call) is created in the **parallel** region, and later a parallel tree traversal is started (the task executing the root of the recursive **compute_tree()** calls). While synchronizing tasks at the end of each tree traversal, using the **taskgroup** construct ensures that the formerly started background task does not participate in the synchronization, and is left free to execute in parallel. This is opposed to the behaviour of the **taskwait** construct, which would include the background tasks in the synchronization.

C / C++

Example taskgroup.1c

```
extern void start_background_work(void);
extern void check_step(void);
extern void print_results(void);
struct tree_node
{
   struct tree_node *left;
   struct tree_node *right;
};
typedef struct tree_node* tree_type;
extern void init_tree(tree_type);
#define max_steps 100
void compute_something(tree_type tree)
{
   // some computation
}
void compute_tree(tree_type tree)
{
   if (tree->left)
   {
      #pragma omp task
        compute_tree(tree->left);
    }
    if (tree->right)
    {
      #pragma omp task
        compute_tree(tree->right);
    }
    #pragma omp task
    compute_something(tree);
}
int main()
{
  int i;
  tree_type tree;
  init_tree(tree);
  #pragma omp parallel
  #pragma omp single
  {
    #pragma omp task
      start_background_work();
    for (i = 0; i < max_steps; i++)
    {
        #pragma omp taskgroup
        {
           #pragma omp task
             compute_tree(tree);
        } // wait on tree traversal in this step
        check_step();
    }
  } // only now is background work required to be complete
  print_results();
  return 0;
}
```

C / C++

Fortran

1 *Example taskgroup.1f*

```
module tree_type_mod
  integer, parameter :: max_steps=100
  type tree_type
    type(tree_type), pointer :: left, right
  end type
  contains
    subroutine compute_something(tree)
      type(tree_type), pointer :: tree
```

```
! some computation
    end subroutine
    recursive subroutine compute_tree(tree)
      type(tree_type), pointer :: tree
      if (associated(tree%left)) then
!$omp task
        call compute_tree(tree%left)
!$omp end task
      endif
      if (associated(tree%right)) then
!$omp task
        call compute_tree(tree%right)
!$omp end task
      endif
!$omp task
      call compute_something(tree)
!$omp end task
    end subroutine
end module
program main
  use tree_type_mod
  type(tree_type), pointer :: tree
  call init_tree(tree);
!$omp parallel
!$omp single
!$omp task
  call start_background_work()
!$omp end task
  do i=1, max_steps
!$omp taskgroup
!$omp task
    call compute_tree(tree)
!$omp end task
!$omp end taskgroup ! wait on tree traversal in this step
    call check_step()
  enddo
!$omp end single
!$omp end parallel    ! only now is background work required to be complete
  call print_results()
end program
```

Fortran

CHAPTER 19

The **`taskyield`** Construct

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 / C++

Example taskyield.1c

```
#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);
      }
}
```

C / C++

Fortran

1 *Example taskyield.1f*

```
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
```

Fortran

CHAPTER 20

The **workshare** Construct

Fortran

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 workshare.1f

```
      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 workshare.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 **SUM(AA)** is workshared, but the update to **R** is atomic.

Example workshare.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
!$OMP       ATOMIC UPDATE
              R = R + SUM(AA)
            CC = DD
!$OMP     END WORKSHARE
!$OMP   END PARALLEL
      END SUBROUTINE WSHARE3
```

Fortran **WHERE** and **FORALL** 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 workshare.4f

```
      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 workshare.5f

```
      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 workshare.6f

```
      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 workshare.7f

```
      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
```

Fortran

CHAPTER 21

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.

C / C++

Example master.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 );
  }
}
```

C / C++

Fortran

Example master.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
```

Fortran

CHAPTER 22

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 / C++

Example critical.1c

```c
int dequeue(float *a);
void work(int i, float *a);

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);
  }

}
```

C / C++

Fortran

Example critical.1f

```
      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
```

Fortran

CHAPTER 23

Worksharing Constructs Inside a critical Construct

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 / C++

Example worksharing_critical.1c

```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++;
          }
        }
      }
    }
  }
}
```

C / C++

Fortran

1 *Example worksharing_critical.1f*

```
      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
```

Fortran

CHAPTER 24

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*.

C / C++

Example barrier_regions.1c

```
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;
}
```

C / C++

Fortran

1 *Example barrier_regions.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
```

Fortran

CHAPTER 25

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 / C++

Example atomic.1c

```
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);
    }
}

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;
}
```

C / C++

Fortran

Example atomic.1f

```
      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
```

Fortran

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.

C / C++

Example atomic.2c

```
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 atomic.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
```

Fortran

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 atomic.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);
}
```

C / C++

Fortran

Example atomic.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
       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
```

Fortran

CHAPTER 26

Restrictions on the `atomic` Construct

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

C / C++

Example atomic_restrict.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 */
  }
}
```

C / C++

Fortran

1 *Example atomic_restrict.1f*

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

!$OMP   PARALLEL
!$OMP     ATOMIC UPDATE
            I = I + 1
!$OMP     ATOMIC UPDATE
            R = R + 1.0
! incorrect because I and R reference the same location
! but have different types
!$OMP   END PARALLEL
      END SUBROUTINE ATOMIC_WRONG
```

Fortran

C / C++

2 *Example atomic_restrict.2c*

```
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 */

  }
}
```

C / C++

The following example is non-conforming because **I** and **R** reference the same location but have different types.

Example atomic_restrict.2f

```
      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 atomic_restrict.3f

```
      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
```

Fortran

CHAPTER 27

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:

C / C++

Example flush_nolist.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 */
    sum += i + j + *p + n;
  }
  return sum;
}

int main()
{
  int result = g(7);
  return result;
}
```

C / C++

Fortran

Example flush_nolist.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
```

```
S-63        END PROGRAM FLUSH_NOLIST
```

Fortran

CHAPTER 28

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++

Example standalone.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.

Fortran

Example standalone.1f

```
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
```

Fortran

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

C / C++

Example standalone.2c

```
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
        }
  }
}
```

C / C++

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.

Fortran

Example standalone.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
```

Fortran

CHAPTER 29

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:

C / C++

Example ordered.1c

```
#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;
}
```

C / C++

Fortran

Example ordered.1f

```
      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
```

Fortran

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:

C / C++

Example ordered.2c

```
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);
  }
}
```

C / C++

Fortran

Example ordered.2f

```
      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
```

Fortran

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

C / C++

1 *Example ordered.3c*

```
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);
    }
  }
}
```

C / C++

Fortran

2 *Example ordered.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

          IF (I > 10) THEN
!$OMP       ORDERED
              CALL WORK(I+1)
!$OMP       END ORDERED
          ENDIF
        ENDDO
      END SUBROUTINE ORDERED_GOOD
```

Fortran

CHAPTER 30

Cancellation Constructs

The following example shows how the **cancel** directive can be used to terminate an OpenMP region. Although the **cancel** construct terminates the OpenMP worksharing region, programmers must still track the exception through the pointer ex and issue a cancellation for the **parallel** region if an exception has been raised. The master thread checks the exception pointer to make sure that the exception is properly handled in the sequential part. If cancellation of the **parallel** region has been requested, some threads might have executed **phase_1()**. However, it is guaranteed that none of the threads executed **phase_2()**.

C / C++

Example cancellation.1c

```
#include <iostream>
#include <exception>

#define N 10000

extern void causes_an_exception();
extern void phase_1();
extern void phase_2();

void example() {
    std::exception *ex = NULL;
#pragma omp parallel shared(ex)
    {
#pragma omp for
        for (int i = 0; i < N; i++) {
            // no 'if' that prevents compiler optimizations
            try {
                causes_an_exception();
            }
            catch (std::exception *e) {
                    // still must remember exception for later handling
#pragma omp atomic write
                    ex = e;
                                    // cancel worksharing construct
#pragma omp cancel for
                  }
              }
   // if an exception has been raised, cancel parallel region
          if (ex) {
#pragma omp cancel parallel
          }
          phase_1();
#pragma omp barrier
          phase_2();
    }
    // continue here if an exception has been thrown in the worksharing loop
    if (ex) {
        // handle exception stored in ex
    }
}
```

C / C++

The following example illustrates the use of the **cancel** construct in error handling. If there is an error condition from the **allocate** statement, the cancellation is activated. The encountering thread sets the shared variable **err** and other threads of the binding thread set proceed to the end of the worksharing construct after the cancellation has been activated.

Fortran

Example cancellation.1f

```
subroutine example(n, dim)
  integer, intent(in) :: n, dim(n)
  integer :: i, s, err
  real, allocatable :: B(:)
  err = 0
!$omp parallel shared(err)
! ...
!$omp do private(s, B)
  do i=1, n
!$omp cancellation point do
    allocate(B(dim(i)), stat=s)
    if (s .gt. 0) then
!$omp atomic write
      err = s
!$omp cancel do
    endif
!   ...
! deallocate private array B
    if (allocated(B)) then
      deallocate(B)
    endif
  enddo
!$omp end parallel
end subroutine
```

Fortran

The following example shows how to cancel a parallel search on a binary tree as soon as the search value has been detected. The code creates a task to descend into the child nodes of the current tree node. If the search value has been found, the code remembers the tree node with the found value through an **atomic** write to the result variable and then cancels execution of all search tasks. The function **search_tree_parallel** groups all search tasks into a single task group to control the effect of the **cancel taskgroup** directive. The *level* argument is used to create undeferred tasks after the first ten levels of the tree.

C / C++

Example cancellation.2c

```
typedef struct binary_tree_s {
   int value;
   struct binary_tree_s *left, *right;
} binary_tree_t;

binary_tree_t *search_tree(binary_tree_t *tree, int value, int level) {
    binary_tree_t *found = NULL;
    if (tree) {
        if (tree->value == value) {
            found = tree;
        }
        else {
#pragma omp task shared(found) if(level < 10)
            {
                binary_tree_t *found_left = NULL;
                found_left = search_tree(tree->left, value, level + 1);
                if (found_left) {
#pragma omp atomic write
                    found = found_left;
#pragma omp cancel taskgroup
                }
            }
#pragma omp task shared(found) if(level < 10)
            {
                binary_tree_t *found_right = NULL;
                found_right = search_tree(tree->right, value, level + 1);
                if (found_right) {
```

```
#pragma omp atomic write
                    found = found_right;
#pragma omp cancel taskgroup
                }
            }
#pragma omp taskwait
        }
    }
    return found;
}
binary_tree_t *search_tree_parallel(binary_tree_t *tree, int value) {
    binary_tree_t *found = NULL;
#pragma omp parallel shared(found, tree, value)
    {
#pragma omp master
        {
#pragma omp taskgroup
            {
                found = search_tree(tree, value, 0);
            }
        }
    }
    return found;
}
```

C / C++

The following is the equivalent parallel search example in Fortran.

Fortran

Example cancellation.2f

```
module parallel_search
  type binary_tree
    integer :: value
    type(binary_tree), pointer :: right
    type(binary_tree), pointer :: left
  end type

contains
  recursive subroutine search_tree(tree, value, level, found)
    type(binary_tree), intent(in), pointer :: tree
    integer, intent(in) :: value, level
    type(binary_tree), pointer :: found
    type(binary_tree), pointer :: found_left => NULL(), found_right => NULL()

    if (associated(tree)) then
      if (tree%value .eq. value) then
```

```
          found => tree
        else
!$omp task shared(found) if(level<10)
          call search_tree(tree%left, value, level+1, found_left)
          if (associated(found_left)) then
!$omp critical
            found => found_left
!$omp end critical

!$omp cancel taskgroup
          endif
!$omp end task

!$omp task shared(found) if(level<10)
          call search_tree(tree%right, value, level+1, found_right)
          if (associated(found_right)) then
!$omp critical
            found => found_right
!$omp end critical

!$omp cancel taskgroup
          endif
!$omp end task

!$omp taskwait
        endif
      endif
    end subroutine

    subroutine search_tree_parallel(tree, value, found)
      type(binary_tree), intent(in), pointer :: tree
      integer, intent(in) :: value
      type(binary_tree), pointer :: found

      found => NULL()
!$omp parallel shared(found, tree, value)
!$omp master
!$omp taskgroup
      call search_tree(tree, value, 0, found)
!$omp end taskgroup
!$omp end master
!$omp end parallel
    end subroutine

end module parallel_search
```

Fortran

CHAPTER 31

The `threadprivate` Directive

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

C / C++

Example threadprivate.1c

```
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
  counter++;
  return(counter);
}
```

C / C++

Fortran

Example threadprivate.1f

```
      INTEGER FUNCTION INCREMENT_COUNTER()
        COMMON/INC_COMMON/COUNTER
!$OMP   THREADPRIVATE(/INC_COMMON/)

        COUNTER = COUNTER +1
        INCREMENT_COUNTER = COUNTER
        RETURN
      END FUNCTION INCREMENT_COUNTER
```

Fortran

C / C++

The following example uses **threadprivate** on a static variable:

Example threadprivate.2c

```
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 **threadprivate** variable. A **threadprivate** variable is initialized once at an unspecified point before its first reference. Because **a** is constructed using the value of **x** (which is modified by the statement **x++**), the value of **a.val** at the start of the **parallel** region could be either 1 or 2. This problem is avoided for **b**, which uses an auxiliary **const** variable and a copy-constructor.

Example threadprivate.3c

```
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 **threadprivate** directive.

Fortran

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

Example threadprivate.2f

```
      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
```

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

Example threadprivate.3f

```
      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
```

The following example is a correct rewrite of the previous example:

Example threadprivate.4f

```
      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
```

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

Example threadprivate.5f

```
      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

!$OMP   PARALLEL COPYIN(I, PTR)
!$OMP     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

                PRINT *, 'ptr = ', PTR
                PRINT *, 'i = ', I
                PRINT *

!$OMP      END CRITICAL
!$OMP    END PARALLEL
      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 = 15

a = 1 2 3
ptr = 4
i = 5
```

The following is an example of the use of **threadprivate** for module variables:

Example threadprivate.6f

```
      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
```

Fortran

C++

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 threadprivate.4c

```
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 threadprivate.5c

```
class T {
 public:
  static int i;
#pragma omp threadprivate(i)
};
```

C++

CHAPTER 32

Parallel Random Access Iterator Loop

C++

The following example shows a parallel random access iterator loop.

Example pra_iterator.1c

```
#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 //
  }
}
```

C++

CHAPTER 33

Fortran Restrictions on `shared` and `private` Clauses with Common Blocks

Fortran

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 fort_sp_common.1f

```
      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 fort_sp_common.2f

```
      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
!$OMP     END DO
!$OMP   END PARALLEL
      END SUBROUTINE COMMON_GOOD2
```

The following example is conforming:

Example fort_sp_common.3f

```
      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 fort_sp_common.4f

```
      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 fort_sp_common.5f

```
      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
```

Fortran

CHAPTER 34

The **default(none)** Clause

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

C / C++

Beginning with OpenMP 4.0, variables with **const**-qualified type and no mutable member are no longer predetermined shared. Thus, these variables (variable *c* in the example) need to be explicitly listed in data-sharing attribute clauses when the **default(none)** clause is specified.

Example default_none.1c

```
#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, c)
  {
     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 and
                              is listed in shared clause */
     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 */
  }
}
```

C / C++

Fortran

Example default_none.1f

```
      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
```

Fortran

CHAPTER 35

Race Conditions Caused by Implied Copies of Shared Variables in Fortran

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 fort_race.1f

```
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
```

Fortran

CHAPTER 36

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 / C++

Example private.1c

```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;
}
```

C / C++

Fortran

Example private.1f

```
      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
```

Fortran

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.

C / C++

Example private.2c

```
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" */
    }
}
```

C / C++

Fortran

Example private.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
```

Fortran

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.

C / C++

Example private.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);
  }
}
```

C / C++

Fortran

1 *Example private.3f*

```
      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

CHAPTER 37

Fortran Restrictions on Storage Association with the `private` Clause

Fortran

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

Example fort_sa_private.1f

```
       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 fort_sa_private.2f

```
      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 PROGRAM PRIV_RESTRICT2
```

1 *Example fort_sa_private.3f*

```
      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
```

2 *Example fort_sa_private.4f*

```
      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
                       ! A becomes undefined at this point
            ENDDO
```

```
          ENDDO
!$OMP END PARALLEL DO       ! The LASTPRIVATE write for A has
                            ! undefined results

         PRINT *, B     ! B is undefined since the LASTPRIVATE
                        ! write of A was not defined
       END PROGRAM PRIV_RESTRICT4
```

Example fort_sa_private.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

CHAPTER 38

C/C++ Arrays in a `firstprivate` Clause

C / C++

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 carrays_fpriv.1c

```
#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++

CHAPTER 39

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 lastprivate.1c

```
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 */
}
```

C / C++

Fortran

1 *Example lastprivate.1f*

```
      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
```

Fortran

CHAPTER 40

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 / C++

Example reduction.1c

```
#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]);
    }
}
```

C / C++

Fortran

Example reduction.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
```

Fortran

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

C / C++

Example reduction.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);
      }
    }
  }
```

C / C++

Fortran

1 *Example reduction.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)
    !$OMP PARALLEL SHARED(X, Y, A, B, C, D, N) &
    !$OMP&         PRIVATE(A_P, B_P, C_P, D_P)
      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 reduction.3f

```
 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 reduction.4f

```
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 reduction.5f

```
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
```

Fortran

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.

C / C++

Example reduction.3c

```
#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);
  }
  return 0;
}
```

C / C++

Fortran

Example reduction.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

CHAPTER 41

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 / C++

Example copyin.1c

```
#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 / C++

Fortran

Example copyin.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
```

Fortran

CHAPTER 42

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.

C / C++

Example copyprivate.1c

```
#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);
    }
}
```

C / C++

Fortran

Example copyprivate.1f

```
      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
```

Fortran

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.

C / C++

Example copyprivate.2c

```
#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;
}
```

C / C++

Fortran

Example copyprivate.2f

```
        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
```

Fortran

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.

C / C++

Example copyprivate.3c

```
#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;
}
```

C / C++

Fortran

Example copyprivate.3f

```
      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 copyprivate.4f

```
      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)
        ! Variable A is private and is
        ! assigned the same value in each thread
        ! Variable B is shared

!$OMP   BARRIER
!$OMP   SINGLE
          DEALLOCATE (B)
!$OMP   END SINGLE NOWAIT
      END SUBROUTINE S
```

Fortran

CHAPTER 43

Nested Loop Constructs

The following example of loop construct nesting is conforming because the inner and outer loop regions bind to different **parallel** regions:

C / C++

Example nested_loop.1c

```
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);
      }
    }
  }
}
```

C / C++

Fortran

1 *Example nested_loop.1f*

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

      SUBROUTINE GOOD_NESTING(N)
      INTEGER N

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

Fortran

2 The following variation of the preceding example is also conforming:

C / C++

3 *Example nested_loop.2c*

```
void work(int i, int j) {}


void work1(int i, int n)
{
  int j;
  #pragma omp parallel default(shared)
  {
    #pragma omp for
    for (j=0; j<n; j++)
      work(i, j);
  }
}


void good_nesting2(int n)
{
  int i;
  #pragma omp parallel default(shared)
  {
    #pragma omp for
    for (i=0; i<n; i++)
      work1(i, n);
  }
}
```

C / C++

Fortran

1 *Example nested_loop.2f*

```
      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
```

Fortran

CHAPTER 44

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 / C++

Example nesting_restrict.1c

```
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);
    }
  }
}
```

C / C++

Fortran

Example nesting_restrict.1f

```
      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
```

Fortran

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

C / C++

Example nesting_restrict.2c

```
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);
  }
}
```

C / C++

Fortran

Example nesting_restrict.2f

```
      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
```

Fortran

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

C / C++

Example nesting_restrict.3c

```
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);
      }
  }
}
```

C / C++

Fortran

Example nesting_restrict.3f

```
      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
```

Fortran

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

C / C++

Example nesting_restrict.4c

```
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);
      }
  }
}
```

C / C++

Fortran

Example nesting_restrict.4f

```
      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
```

Fortran

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:

C / C++

Example nesting_restrict.5c

```
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);
    }
  }
}
```

C / C++

Fortran

Example nesting_restrict.5f

```
      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
```

Fortran

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:

C / C++

Example nesting_restrict.6c

```
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);
    }
  }
}
```

C / C++

Fortran

Example nesting_restrict.6f

```fortran
      SUBROUTINE WRONG6(N)
      INTEGER N

!$OMP   PARALLEL DEFAULT(SHARED)
!$OMP     SINGLE
            CALL WORK(N,1)
! incorrect nesting of barrier region in a single region
!$OMP       BARRIER
            CALL WORK(N,2)
!$OMP     END SINGLE
!$OMP   END PARALLEL
      END SUBROUTINE WRONG6
```

Fortran

CHAPTER 45

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.

C / C++

Example set_dynamic_nthrs.1c

```
#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);
    }
}
```

C / C++

Fortran

1 *Example set_dynamic_nthrs.1f*

```
      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
```

Fortran

CHAPTER 46

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.

C / C++

Example get_nthrs.1c

```
#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);
}
```

C / C++

Fortran

Example get_nthrs.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
```

Fortran

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

C / C++

Example get_nthrs.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);
  }
}
```

C / C++

Fortran

1 *Example get_nthrs.2f*

```
      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
```

Fortran

CHAPTER 47

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`**.

C / C++

Example init_lock.1c

```
#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;
}
```

C / C++

Fortran

Example init_lock.1f

```
      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
```

Fortran

CHAPTER 48

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. Beginning 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 beginning with OpenMP 3.0, because the task region that releases the lock **lck** is different from the task region that acquires the lock.

C / C++

Example lock_owner.1c

```
#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;
}
```

C / C++

Fortran

1 *Example lock_owner.1f*

```
        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
```

Fortran

CHAPTER 49

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.

Note that the argument to the lock routines should have type **omp_lock_t**, and that there is no need to flush it.

C / C++

Example simple_lock.1c

```
#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;
  }
```

C / C++

Note that there is no need to flush the lock variable.

Fortran

Example simple_lock.1f

```
      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
```

Fortran

CHAPTER 50

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.

C / C++

Example nestable_lock.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());
  }
}
```

C / C++

Fortran

1 *Example nestable_lock.1f*

```
      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

!$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
```

Fortran

CHAPTER 51

SIMD Constructs

The following examples illustrate the use of SIMD constructs for vectorization.

Compilers may not vectorize loops when they are complex or possibly have dependencies, even though the programmer is certain the loop will execute correctly as a vectorized loop. The **simd** construct assures the compiler that the loop can be vectorized.

C / C++

Example SIMD.1c

```
void star( double *a, double *b, double *c, int n, int *ioff )
{
   int i;
   #pragma omp simd
   for ( i = 0; i < n; i++ )
      a[i] *= b[i] * c[i+ *ioff];
}
```

C / C++

Fortran

Example SIMD.1f

```
subroutine star(a,b,c,n,ioff_ptr)
   implicit none
   double precision :: a(*),b(*),c(*)
   integer          :: n, i
   integer, pointer :: ioff_ptr

   !$omp simd
   do i = 1,n
      a(i) = a(i) * b(i) * c(i+ioff_ptr)
   end do

end subroutine
```

Fortran

When a function can be inlined within a loop the compiler has an opportunity to vectorize the loop. By guaranteeing SIMD behavior of a function's operations, characterizing the arguments of the function and privatizing temporary variables of the loop, the compiler can often create faster, vector code for the loop. In the examples below the **declare simd** construct is used on the *add1* and *add2* functions to enable creation of their corresponding SIMD function versions for execution within the associated SIMD loop. The functions characterize two different approaches of accessing data within the function: by a single variable and as an element in a data array, respectively. The *add3* C function uses dereferencing.

The **declare simd** constructs also illustrate the use of **uniform** and **linear** clauses. The **uniform(fact)** clause indicates that the variable *fact* is invariant across the SIMD lanes. In the *add2* function *a* and *b* are included in the **unform** list because the C pointer and the Fortran array references are constant. The *i* index used in the *add2* function is included in a **linear** clause with a constant-linear-step of 1, to guarantee a unity increment of the associated loop. In the **declare simd** construct for the *add3* C function the **linear(a,b:1)** clause instructs the compiler to generate unit-stride loads across the SIMD lanes; otherwise, costly *gather* instructions would be generated for the unknown sequence of access of the pointer dereferences.

In the **simd** constructs for the loops the **private(tmp)** clause is necessary to assure that the each vector operation has its own *tmp* variable.

C / C++

Example SIMD.2c

```
#include <stdio.h>

#pragma omp declare simd uniform(fact)
double add1(double a, double b, double fact)
{
   double c;
   c = a + b + fact;
   return c;
}

#pragma omp declare simd uniform(a,b,fact) linear(i:1)
double add2(double *a, double *b, int i, double fact)
{
   double c;
   c = a[i] + b[i] + fact;
   return c;
}

#pragma omp declare simd uniform(fact) linear(a,b:1)
double add3(double *a, double *b, double fact)
{
   double c;
   c = *a + *b + fact;
   return c;
}

void work( double *a, double *b, int n )
{
   int i;
   double tmp;
   #pragma omp simd private(tmp)
   for ( i = 0; i < n; i++ ) {
      tmp  = add1( a[i],  b[i], 1.0);
      a[i] = add2( a,     b, i, 1.0) + tmp;
      a[i] = add3(&a[i], &b[i], 1.0);
   }
}

int main(){
   int i;
   const int N=32;
   double a[N], b[N];

   for ( i=0; i<N; i++ ) {
      a[i] = i; b[i] = N-i;
   }

   work(a, b, N );

   for ( i=0; i<N; i++ ) {
      printf("%d %f\n", i, a[i]);
   }

   return 0;
}
```

C / C++

Fortran

Example SIMD.2f

```
program main
   implicit none
   integer, parameter :: N=32
   integer :: i
   double precision   :: a(N), b(N)
   do i = 1,N
      a(i) = i-1
      b(i) = N-(i-1)
      end do
      call work(a, b, N )
      do i = 1,N
         print*, i,a(i)
      end do
   end program

   function add1(a,b,fact) result(c)
   !$omp declare simd(add1) uniform(fact)
      implicit none
      double precision :: a,b,fact, c
      c = a + b + fact
   end function

   function add2(a,b,i, fact) result(c)
   !$omp declare simd(add2) uniform(a,b,fact) linear(i:1)
      implicit none
      integer          :: i
      double precision :: a(*),b(*),fact, c
      c = a(i) + b(i) + fact
   end function

   subroutine work(a, b, n )
      implicit none
      double precision           :: a(n),b(n), tmp
      integer                    :: n, i
      double precision, external :: add1, add2

      !$omp simd private(tmp)
      do i = 1,n
         tmp  = add1(a(i), b(i), 1.0d0)
         a(i) = add2(a,    b, i, 1.0d0) + tmp
         a(i) = a(i) + b(i) + 1.0d0
      end do
   end subroutine
```

Fortran

A thread that encounters a SIMD construct executes a vectorized code of the iterations. Similar to the concerns of a worksharing loop a loop vectorized with a SIMD construct must assure that temporary and reduction variables are privatized and declared as reductions with clauses. The example below illustrates the use of **private** and **reduction** clauses in a SIMD construct.

C / C++

Example SIMD.3c

```
double work( double *a, double *b, int n )
{
   int i;
   double tmp, sum;
   sum = 0.0;
   #pragma omp simd private(tmp) reduction(+:sum)
   for (i = 0; i < n; i++) {
      tmp = a[i] + b[i];
      sum += tmp;
   }
   return sum;
}
```

C / C++

Fortran

Example SIMD.3f

```
subroutine work( a, b, n, sum )
   implicit none
   integer :: i, n
   double precision :: a(n), b(n), sum, tmp

   sum = 0.0d0
   !$omp simd private(tmp) reduction(+:sum)
   do i = 1,n
      tmp = a(i) + b(i)
      sum = sum + tmp
   end do

end subroutine work
```

Fortran

A **`safelen(N)`** clause in a **`simd`** construct assures the compiler that there are no loop-carried dependencies for vectors of size *N* or below. If the **`safelen`** clause is not specified, then the default safelen value is the number of loop iterations.

The **`safelen(16)`** clause in the example below guarantees that the vector code is safe for vectors up to and including size 16. In the loop, *m* can be 16 or greater, for correct code execution. If the value of *m* is less than 16, the behavior is undefined.

C / C++

1 *Example SIMD.4c*

```
void work( float *b, int n, int m )
{
   int i;
   #pragma omp simd safelen(16)
   for (i = m; i < n; i++)
      b[i] = b[i-m] - 1.0f;
}
```

C / C++

Fortran

2 *Example SIMD.4f*

```
subroutine work( b, n, m )
   implicit none
   real       :: b(n)
   integer    :: i,n,m

   !$omp simd safelen(16)
   do i = m+1, n
      b(i) = b(i-m) - 1.0
   end do
end subroutine work
```

Fortran

3 The following SIMD construct instructs the compiler to collapse the *i* and *j* loops into a single
4 SIMD loop in which SIMD chunks are executed by threads of the team. Within the workshared
5 loop chunks of a thread, the SIMD chunks are executed in the lanes of the vector units.

C / C++

6 *Example SIMD.5c*

```
void work( double **a, double **b, double **c, int n )
{
   int i, j;
   double tmp;
   #pragma omp for simd collapse(2) private(tmp)
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++) {
         tmp = a[i][j] + b[i][j];
         c[i][j] = tmp;
      }
   }
}
```

C / C++

Fortran

Example SIMD.5f

```fortran
subroutine work( a, b, c,  n )
   implicit none
   integer :: i,j,n
   double precision :: a(n,n), b(n,n), c(n,n), tmp

   !$omp for simd collapse(2) private(tmp)
   do j = 1,n
      do i = 1,n
         tmp = a(i,j) + b(i,j)
         c(i,j) = tmp
      end do
   end do

end subroutine work
```

Fortran

The following examples illustrate the use of the **declare simd** construct with the **inbranch** and **notinbranch** clauses. The **notinbranch** clause informs the compiler that the function *foo* is never called conditionally in the SIMD loop of the function *myaddint*. On the other hand, the **inbranch** clause for the function goo indicates that the function is always called conditionally in the SIMD loop inside the function *myaddfloat*.

C / C++

Example SIMD.6c

```c
#pragma omp declare simd linear(p:1) notinbranch
int foo(int *p){
  *p = *p + 10;
  return *p;
}

int myaddint(int *a, int *b, int n)
{
#pragma omp simd
  for (int i=0; i<n; i++){
      a[i]  = foo(&b[i]);  /* foo is not called under a condition */
  }
  return a[n-1];
}

#pragma omp declare simd linear(p:1) inbranch
float goo(float *p){
  *p = *p + 18.5f;
  return *p;
}

int myaddfloat(float *x, float *y, int n)
{
#pragma omp simd
  for (int i=0; i<n; i++){
     x[i] = (x[i] > y[i]) ? goo(&y[i]) : y[i];
       /* goo is called under the condition (or within a branch) */
  }
  return x[n-1];
}
```

C / C++

Fortran

1 *Example SIMD.6f*

```
function foo(p) result(r)
!$omp declare simd(foo) notinbranch
  implicit none
  integer :: p, r
  p = p + 10
  r = p
end function foo

function myaddint(int *a, int *b, int n) result(r)
  implicit none
  integer :: a(*), b(*), n, r
  integer :: i
  integer, external :: foo

  !$omp simd
  do i=1, n
      a(i) = foo(b[i])  ! foo is not called under a condition
  end do
  r = a(n)

end function myaddint

function goo(p) result(r)
!$omp declare simd(goo) inbranch
  implicit none
  real :: p, r
  p = p + 18.5
  r = p
end function goo
```

```fortran

function myaddfloat(x, y, n) result(r)
  implicit none
  real :: x(*), y(*), r
  integer :: n
  integer :: i
  real, external :: goo

  !$omp simd
  do i=1, n
     if (x(i) > y(i)) then
        x(i) = goo(y(i))
        ! goo is called under the condition (or within a branch)
     else
        x(i) = y(i)
     endif
  end do

  r = x(n)
end function myaddfloat
```

Fortran

In the code below, the function *fib()* is called in the main program and also recursively called in the function *fib()* within an **if** condition. The compiler creates a masked vector version and a non-masked vector version for the function *fib()* while retaining the original scalar version of the *fib()* function.

C / C++

Example SIMD.7c

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

#define N 45
int a[N], b[N], c[N];

#pragma omp declare simd inbranch
int fib( int n )
{
   if (n <= 2)
      return n;
   else {
      return fib(n-1) + fib(n-2);
   }
}

int main(void)
```

```
{
   int i;

   #pragma omp simd
   for (i=0; i < N; i++) b[i] = i;

   #pragma omp simd
   for (i=0; i < N; i++) {
      a[i] = fib(b[i]);
   }
   printf("Done a[%d] = %d\n", N-1, a[N-1]);
   return 0;
}
```

C / C++

Fortran

Example SIMD.7f

```
program fibonacci
   implicit none
   integer,parameter :: N=45
   integer           :: a(0:N-1), b(0:N-1)
   integer           :: i
   integer, external :: fib

   !$omp simd
   do i = 0,N-1
      b(i) = i
   end do

   !$omp simd
   do i=0,N-1
      a(i) = fib(b(i))
   end do

   write(*,*) "Done a(", N-1, ") = ", a(N-1)
                           ! 44  1134903168
end program

recursive function fib(n) result(r)
!$omp declare simd(fib) inbranch
   implicit none
   integer  :: n, r

   if (n <= 2) then
      r = n
   else
```

```
      r = fib(n-1) + fib(n-2)
    endif

end function fib
```

Fortran

CHAPTER 52

`target` Construct

52.1 `target` Construct on `parallel` Construct

This following example shows how the **`target`** construct offloads a code region to a target device. The variables *p*, *v1*, *v2*, and *N* are implicitly mapped to the target device.

C / C++

Example target.1c

```
extern void init(float*, float*, int);
extern void output(float*, int);
void vec_mult(int N)
{
   int i;
   float p[N], v1[N], v2[N];
   init(v1, v2, N);
   #pragma omp target
   #pragma omp parallel for private(i)
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
   output(p, N);
}
```

C / C++

Fortran

Example target.1f

```
subroutine vec_mult(N)
   integer ::  i,N
   real    ::  p(N), v1(N), v2(N)
   call init(v1, v2, N)
   !$omp target
   !$omp parallel do
   do i=1,N
      p(i) = v1(i) * v2(i)
   end do
   !$omp end target
   call output(p, N)
end subroutine
```

Fortran

52.2 `target` Construct with `map` Clause

This following example shows how the **`target`** construct offloads a code region to a target device. The variables *p*, *v1* and *v2* are explicitly mapped to the target device using the **`map`** clause. The variable *N* is implicitly mapped to the target device.

C / C++

Example target.2c

```
extern void init(float*, float*, int);
extern void output(float*, int);
void vec_mult(int N)
{
   int i;
   float p[N], v1[N], v2[N];
   init(v1, v2, N);
   #pragma omp target map(v1, v2, p)
   #pragma omp parallel for
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
   output(p, N);
}
```

C / C++

Fortran

Example target.2f

```
subroutine vec_mult(N)
   integer ::  i,N
   real    ::  p(N), v1(N), v2(N)
   call init(v1, v2, N)
   !$omp target map(v1,v2,p)
   !$omp parallel do
   do i=1,N
      p(i) = v1(i) * v2(i)
   end do
   !$omp end target
   call output(p, N)
end subroutine
```

Fortran

52.3 `map` Clause with `to`/`from` map-types

The following example shows how the **`target`** construct offloads a code region to a target device. In the **`map`** clause, the **`to`** and **`from`** map-types define the mapping between the original (host) data and the target (device) data. The **`to`** map-type specifies that the data will only be read on the device, and the **`from`** map-type specifies that the data will only be written to on the device. By specifying a guaranteed access on the device, data transfers can be reduced for the **`target`** region.

The **`to`** map-type indicates that at the start of the **`target`** region the variables *v1* and *v2* are initialized with the values of the corresponding variables on the host device, and at the end of the **`target`** region the variables *v1* and *v2* are not assigned to their corresponding variables on the host device.

The **`from`** map-type indicates that at the start of the **`target`** region the variable *p* is not initialized with the value of the corresponding variable on the host device, and at the end of the **`target`** region the variable *p* is assigned to the corresponding variable on the host device.

C / C++

Example target.3c

```
extern void init(float*, float*, int);
extern void output(float*, int);
void vec_mult(int N)
{
   int i;
   float p[N], v1[N], v2[N];
   init(v1, v2, N);
   #pragma omp target map(to: v1, v2) map(from: p)
   #pragma omp parallel for
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
   output(p, N);
}
```

C / C++

The **to** and **from** map-types allow programmers to optimize data motion. Since data for the *v* arrays are not returned, and data for the *p* array are not transferred to the device, only one-half of the data is moved, compared to the default behavior of an implicit mapping.

Fortran

Example target.3f

```
subroutine vec_mult(N)
   integer ::  i,N
   real    ::  p(N), v1(N), v2(N)
   call init(v1, v2, N)
   !$omp target map(to: v1,v2) map(from: p)
   !$omp parallel do
   do i=1,N
      p(i) = v1(i) * v2(i)
   end do
   !$omp end target
   call output(p, N)
end subroutine
```

Fortran

52.4 **map** Clause with Array Sections

The following example shows how the **target** construct offloads a code region to a target device. In the **map** clause, map-types are used to optimize the mapping of variables to the target device. Because variables *p*, *v1* and *v2* are pointers, array section notation must be used to map the arrays. The notation **:N** is equivalent to **0:N**.

C / C++

Example target.4c

```
extern void init(float*, float*, int);
extern void output(float*, int);
void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);
   #pragma omp target map(to: v1[0:N], v2[:N]) map(from: p[0:N])
   #pragma omp parallel for
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
   output(p, N);
}
```

C / C++

In C, the length of the pointed-to array must be specified. In Fortran the extent of the array is known and the length need not be specified. A section of the array can be specified with the usual Fortran syntax, as shown in the following example. The value 1 is assumed for the lower bound for array section *v2(:N)*.

Fortran

Example target.4f

```
module mults
contains
subroutine vec_mult(p,v1,v2,N)
   real,pointer,dimension(:) :: p, v1, v2
   integer                   :: N,i
   call init(v1, v2, N)
   !$omp target map(to: v1(1:N), v2(:N)) map(from: p(1:N))
   !$omp parallel do
   do i=1,N
      p(i) = v1(i) * v2(i)
   end do
   !$omp end target
   call output(p, N)
end subroutine
end module
```

Fortran

A more realistic situation in which an assumed-size array is passed to **vec_mult** requires that the length of the arrays be specified, because the compiler does not know the size of the storage. A section of the array must be specified with the usual Fortran syntax, as shown in the following example. The value 1 is assumed for the lower bound for array section *v2(:N)*.

Fortran

Example target.4bf

```
module mults
contains
subroutine vec_mult(p,v1,v2,N)
   real,dimension(*) :: p, v1, v2
   integer           :: N,i
   call init(v1, v2, N)
   !$omp target map(to: v1(1:N), v2(:N)) map(from: p(1:N))
   !$omp parallel do
   do i=1,N
      p(i) = v1(i) * v2(i)
   end do
   call output(p, N)
   !$omp end target
end subroutine
end module
```

Fortran

52.5 `target` Construct with `if` Clause

The following example shows how the **target** construct offloads a code region to a target device.

The **if** clause on the **target** construct indicates that if the variable *N* is smaller than a given threshold, then the **target** region will be executed by the host device.

The **if** clause on the **parallel** construct indicates that if the variable *N* is smaller than a second threshold then the **parallel** region is inactive.

C / C++

1 *Example target.5c*

```
#define THRESHOLD1 1000000
#define THRESHOLD2 1000
extern void init(float*, float*, int);
extern void output(float*, int);
void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);
   #pragma omp target if(N>THRESHOLD1) map(to: v1[0:N], v2[:N])\
        map(from: p[0:N])
   #pragma omp parallel for if(N>THRESHOLD2)
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
   output(p, N);
}
```

C / C++

Fortran

2 *Example target.5f*

```
module params
integer,parameter :: THRESHOLD1=1000000, THRESHHOLD2=1000
end module
subroutine vec_mult(p, v1, v2, N)
   use params
   real    ::  p(N), v1(N), v2(N)
   integer ::  i
   call init(v1, v2, N)
   !$omp target if(N>THRESHHOLD1) map(to: v1, v2 ) map(from: p)
      !$omp parallel do if(N>THRESHOLD2)
      do i=1,N
         p(i) = v1(i) * v2(i)
      end do
   !$omp end target
   call output(p, N)
end subroutine
```

Fortran

CHAPTER 53

target data Construct

53.1 Simple **target data** Construct

This example shows how the **target data** construct maps variables to a device data environment. The **target data** construct creates a new device data environment and maps the variables *v1*, *v2*, and *p* to the new device data environment. The **target** construct enclosed in the **target data** region creates a new device data environment, which inherits the variables *v1*, *v2*, and *p* from the enclosing device data environment. The variable *N* is mapped into the new device data environment from the encountering task's data environment.

C / C++

Example target_data.1c

```
extern void init(float*, float*, int);
extern void output(float*, int);
void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);
   #pragma omp target data map(to: v1[0:N], v2[:N]) map(from: p[0:N])
   {
      #pragma omp target
      #pragma omp parallel for
      for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];
   }
   output(p, N);
}
```

C / C++

The Fortran code passes a reference and specifies the extent of the arrays in the declaration. No length information is necessary in the map clause, as is required with C/C++ pointers.

Fortran

Example target_data.1f

```fortran
subroutine vec_mult(p, v1, v2, N)
   real    ::  p(N), v1(N), v2(N)
   integer ::  i
   call init(v1, v2, N)
   !$omp target data map(to: v1, v2) map(from: p)
   !$omp target
   !$omp parallel do
      do i=1,N
         p(i) = v1(i) * v2(i)
      end do
   !$omp end target
   !$omp end target data
   call output(p, N)
end subroutine
```

Fortran

53.2 `target data` Region Enclosing Multiple `target` Regions

The following examples show how the **`target data`** construct maps variables to a device data environment of a **`target`** region. The **`target data`** construct creates a device data environment and encloses **`target`** regions, which have their own device data environments. The device data environment of the **`target data`** region is inherited by the device data environment of an enclosed **`target`** region. The **`target data`** construct is used to create variables that will persist throughout the **`target data`** region.

In the following example the variables *v1* and *v2* are mapped at each **`target`** construct. Instead of mapping the variable *p* twice, once at each **`target`** construct, *p* is mapped once by the **`target data`** construct.

C / C++

Example target_data.2c

```
extern void init(float*, float*, int);
extern void init_again(float*, float*, int);
extern void output(float*, int);
void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);
   #pragma omp target data map(from: p[0:N])
   {
      #pragma omp target map(to: v1[:N], v2[:N])
      #pragma omp parallel for
      for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];
      init_again(v1, v2, N);
      #pragma omp target map(to: v1[:N], v2[:N])
      #pragma omp parallel for
      for (i=0; i<N; i++)
        p[i] = p[i] + (v1[i] * v2[i]);
   }
   output(p, N);
}
```

C / C++

The Fortran code uses reference and specifies the extent of the *p*, *v1* and *v2* arrays. No length information is necessary in the **map** clause, as is required with C/C++ pointers. The arrays *v1* and *v2* are mapped at each **target** construct. Instead of mapping the array *p* twice, once at each target construct, *p* is mapped once by the **target data** construct.

Fortran

Example target_data.2f

```
subroutine vec_mult(p, v1, v2, N)
   real    ::  p(N), v1(N), v2(N)
   integer ::  i
   call init(v1, v2, N)
   !$omp target data map(from: p)
      !$omp target map(to: v1, v2 )
         !$omp parallel do
         do i=1,N
            p(i) = v1(i) * v2(i)
         end do
      !$omp end target
      call init_again(v1, v2, N)
      !$omp target map(to: v1, v2 )
         !$omp parallel do
         do i=1,N
            p(i) = p(i) + v1(i) * v2(i)
         end do
      !$omp end target
   !$omp end target data
   call output(p, N)
end subroutine
```

Fortran

In the following example, the variable tmp defaults to **tofrom** map-type and is mapped at each **target** construct. The array *Q* is mapped once at the enclosing **target data** region instead of at each **target** construct.

C / C++

Example target_data.3c

```
#include <math.h>
#define COLS 100
void gramSchmidt(float Q[][COLS], const int rows)
{
    int cols = COLS;
    #pragma omp target data map(Q[0:rows][0:cols])
    for(int k=0; k < cols; k++)
    {
        double tmp = 0.0;
        #pragma omp target
        #pragma omp parallel for reduction(+:tmp)
        for(int i=0; i < rows; i++)
            tmp += (Q[i][k] * Q[i][k]);
        tmp = 1/sqrt(tmp);
        #pragma omp target
        #pragma omp parallel for
        for(int i=0; i < rows; i++)
            Q[i][k] *= tmp;
    }
}
```

C / C++

In the following example the arrays *v1* and *v2* are mapped at each **target** construct. Instead of mapping the array *Q* twice at each **target** construct, *Q* is mapped once by the **target data** construct. Note, the *tmp* variable is implicitly remapped for each **target** region, mapping the value from the device to the host at the end of the first **target** region, and from the host to the device for the second **target** region.

Fortran

Example target_data.3f

```
subroutine gramSchmidt(Q,rows,cols)
integer              ::   rows,cols,  i,k
double precision     :: Q(rows,cols), tmp
      !$omp target data map(Q)
      do k=1,cols
         tmp = 0.0d0
        !$omp target
           !$omp parallel do reduction(+:tmp)
           do i=1,rows
              tmp = tmp + (Q(i,k) * Q(i,k))
           end do
        !$omp end target
          tmp = 1.0d0/sqrt(tmp)
        !$omp target
           !$omp parallel do
           do i=1,rows
               Q(i,k) = Q(i,k)*tmp
           enddo
        !$omp end target
      end do
      !$omp end target data
end subroutine
```

Fortran

53.3 `target data` Construct with Orphaned Call

The following two examples show how the **`target data`** construct maps variables to a device data environment. The **`target data`** construct's device data environment encloses the **`target`** construct's device data environment in the function **`vec_mult()`**.

When the type of the variable appearing in an array section is pointer, the pointer variable and the storage location of the corresponding array section are mapped to the device data environment. The pointer variable is treated as if it had appeared in a **`map`** clause with a map-type of **`alloc`**. The array section's storage location is mapped according to the map-type in the **`map`** clause (the default map-type is **`tofrom`**).

The **target** construct's device data environment inherits the storage locations of the array sections *v1[0:N]*, *v2[:n]*, and *p0[0:N]* from the enclosing target data construct's device data environment. Neither initialization nor assignment is performed for the array sections in the new device data environment.

The pointer variables *p1*, *v3*, and *v4* are mapped into the target construct's device data environment with an implicit map-type of alloc and they are assigned the address of the storage location associated with their corresponding array sections. Note that the following pairs of array section storage locations are equivalent (*p0[:N]*, *p1[:N]*), (*v1[:N]*,*v3[:N]*), and (*v2[:N]*,*v4[:N]*).

C / C++

Example target_data.4c

```
void vec_mult(float*, float*, float*, int);
extern void init(float*, float*, int);
extern void output(float*, int);
void foo(float *p0, float *v1, float *v2, int N)
{
   init(v1, v2, N);
   #pragma omp target data map(to: v1[0:N], v2[:N]) map(from: p0[0:N])
   {
      vec_mult(p0, v1, v2, N);
   }
   output(p0, N);
}
void vec_mult(float *p1, float *v3, float *v4, int N)
{
   int i;
   #pragma omp target map(to: v3[0:N], v4[:N]) map(from: p1[0:N])
   #pragma omp parallel for
   for (i=0; i<N; i++)
     p1[i] = v3[i] * v4[i];
}
```

C / C++

The Fortran code maps the pointers and storage in an identical manner (same extent, but uses indices from 1 to *N*).

The **target** construct's device data environment inherits the storage locations of the arrays *v1*, *v2* and *p0* from the enclosing **target data** constructs's device data environment. However, in Fortran the associated data of the pointer is known, and the shape is not required.

The pointer variables *p1*, *v3*, and *v4* are mapped into the **target** construct's device data environment with an implicit map-type of **alloc** and they are assigned the address of the storage location associated with their corresponding array sections. Note that the following pair of array storage locations are equivalent (*p0*,*p1*), (*v1*,*v3*), and (*v2*,*v4*).

Fortran

Example target_data.4f

```
module mults
contains
subroutine foo(p0,v1,v2,N)
real,pointer,dimension(:) :: p0, v1, v2
integer                   :: N,i
   call init(v1, v2, N)
   !$omp target data map(to: v1, v2) map(from: p0)
    call vec_mult(p0,v1,v2,N)
   !omp end target data
   call output(p0, N)
end subroutine
subroutine vec_mult(p1,v3,v4,N)
real,pointer,dimension(:) :: p1, v3, v4
integer                   :: N,i
   !$omp target map(to: v3, v4) map(from: p1)
   !$omp parallel do
   do i=1,N
      p1(i) = v3(i) * v4(i)
   end do
   !$omp end target
end subroutine
end module
```

Fortran

In the following example, the variables *p1*, *v3*, and *v4* are references to the pointer variables *p0*, *v1* and *v2* respectively. The **target** construct's device data environment inherits the pointer variables *p0*, *v1*, and *v2* from the enclosing **target data** construct's device data environment. Thus, *p1*, *v3*, and *v4* are already present in the device data environment.

C / C++

Example target_data.5c

```
void vec_mult(float* &, float* &, float* &, int &);
extern void init(float*, float*, int);
extern void output(float*, int);
void foo(float *p0, float *v1, float *v2, int N)
{
   init(v1, v2, N);
   #pragma omp target data map(to: v1[0:N], v2[:N]) map(from: p0[0:N])
   {
      vec_mult(p0, v1, v2, N);
   }
   output(p0, N);
}
```

```
void vec_mult(float* &p1, float* &v3, float* &v4, int &N)
{
   int i;
   #pragma omp target map(to: v3[0:N], v4[:N]) map(from: p1[0:N])
   #pragma omp parallel for
   for (i=0; i<N; i++)
     p1[i] = v3[i] * v4[i];
}
```

C / C++

In the following example, the usual Fortran approach is used for dynamic memory. The *p0*, *v1*, and *v2* arrays are allocated in the main program and passed as references from one routine to another. In **`vec_mult`**, *p1*, *v3* and *v4* are references to the *p0*, *v1*, and *v2* arrays, respectively. The **`target`** construct's device data environment inherits the arrays *p0*, *v1*, and *v2* from the enclosing target data construct's device data environment. Thus, *p1*, *v3*, and *v4* are already present in the device data environment.

Fortran

Example target_data.5f

```
module my_mult
contains
subroutine foo(p0,v1,v2,N)
real,dimension(:) :: p0, v1, v2
integer           :: N,i
   call init(v1, v2, N)
   !$omp target data map(to: v1, v2) map(from: p0)
    call vec_mult(p0,v1,v2,N)
   !omp end target data
   call output(p0, N)
end subroutine
subroutine vec_mult(p1,v3,v4,N)
real,dimension(:) :: p1, v3, v4
integer           :: N,i
   !$omp target map(to: v3, v4) map(from: p1)
   !$omp parallel do
   do i=1,N
      p1(i) = v3(i) * v4(i)
   end do
   !$omp end target
end subroutine
end module
program main
use my_mult
integer, parameter :: N=1024
real,allocatable, dimension(:) :: p, v1, v2
   allocate( p(N), v1(N), v2(N) )
   call foo(p,v1,v2,N)
end program
```

Fortran

53.4 target data Construct with if Clause

The following two examples show how the **target data** construct maps variables to a device data environment.

In the following example, the if clause on the **target data** construct indicates that if the variable *N* is smaller than a given threshold, then the **target data** construct will not create a device data environment.

The **target** constructs enclosed in the **target data** region must also use an **if** clause on the same condition, otherwise the pointer variable *p* is implicitly mapped with a map-type of **tofrom**, but the storage location for the array section *p[0:N]* will not be mapped in the device data environments of the **target** constructs.

C / C++

Example target_data.6c

```
#define THRESHOLD 1000000
extern void init(float*, float*, int);
extern void init_again(float*, float*, int);
extern void output(float*, int);
void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);
   #pragma omp target data if(N>THRESHOLD) map(from: p[0:N])
   {
      #pragma omp target if (N>THRESHOLD) map(to: v1[:N], v2[:N])
      #pragma omp parallel for
      for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];
      init_again(v1, v2, N);
      #pragma omp target if (N>THRESHOLD) map(to: v1[:N], v2[:N])
      #pragma omp parallel for
      for (i=0; i<N; i++)
        p[i] = p[i] + (v1[i] * v2[i]);
   }
   output(p, N);
}
```

C / C++

The **if** clauses work the same way for the following Fortran code. The **target** constructs enclosed in the **target data** region should also use an **if** clause with the same condition, so that the **target data** region and the **target** region are either both created for the device, or are both ignored.

Fortran

Example target_data.6f

```
module params
integer,parameter :: THRESHOLD=1000000
end module
subroutine vec_mult(p, v1, v2, N)
   use params
   real    ::  p(N), v1(N), v2(N)
   integer ::  i
   call init(v1, v2, N)
   !$omp target data if(N>THRESHOLD) map(from: p)
      !$omp target if(N>THRESHOLD) map(to: v1, v2)
         !$omp parallel do
         do i=1,N
            p(i) = v1(i) * v2(i)
         end do
      !$omp end target
      call init_again(v1, v2, N)
      !$omp target if(N>THRESHOLD) map(to: v1, v2)
         !$omp parallel do
         do i=1,N
            p(i) = p(i) + v1(i) * v2(i)
         end do
      !$omp end target
   !$omp end target data
   call output(p, N)
end subroutine
```

Fortran

In the following example, when the **if** clause conditional expression on the **target** construct evaluates to *false*, the target region will execute on the host device. However, the **target data** construct created an enclosing device data environment that mapped *p[0:N]* to a device data environment on the default device. At the end of the **target data** region the array section *p[0:N]* will be assigned from the device data environment to the corresponding variable in the data environment of the task that encountered the **target data** construct, resulting in undefined values in *p[0:N]*.

C / C++

Example target_data.7c

```
#define THRESHOLD 1000000
extern void init(float*, float*, int);
extern void output(float*, int);
void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);
   #pragma omp target data map(from: p[0:N])
   {
      #pragma omp target if (N>THRESHOLD) map(to: v1[:N], v2[:N])
      #pragma omp parallel for
      for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];
   } /* UNDEFINED behavior if N<=THRESHOLD */
   output(p, N);
}
```

C / C++

The **if** clauses work the same way for the following Fortran code. When the **if** clause conditional expression on the **target** construct evaluates to *false*, the **target** region will execute on the host device. However, the **target data** construct created an enclosing device data environment that mapped the *p* array (and *v1* and *v2*) to a device data environment on the default target device. At the end of the **target data** region the *p* array will be assigned from the device data environment to the corresponding variable in the data environment of the task that encountered the **target data** construct, resulting in undefined values in *p*.

Fortran

Example target_data.7f

```
module params
integer, parameter :: THRESHOLD=1000000
end module
subroutine vec_mult(p, v1, v2, N)
   use params
   real    ::  p(N), v1(N), v2(N)
   integer ::  i
   call init(v1, v2, N)
   !$omp target data map(from: p)
      !$omp target if(N>THRESHOLD) map(to: v1, v2)
         !$omp parallel do
         do i=1,N
            p(i) = v1(i) * v2(i)
         end do
      !$omp end target
    !$omp end target data
    call output(p, N)  !*** UNDEFINED behavior if N<=THRESHOLD
end subroutine
```

Fortran

CHAPTER 54

`target update` Construct

54.1 Simple `target data` and `target update` Constructs

The following example shows how the **`target update`** construct updates variables in a device data environment.

The **`target data`** construct maps array sections *v1[:N]* and *v2[:N]* (arrays *v1* and *v2* in the Fortran code) into a device data environment.

The task executing on the host device encounters the first **`target`** region and waits for the completion of the region.

After the execution of the first **`target`** region, the task executing on the host device then assigns new values to *v1[:N]* and *v2[:N]* (*v1* and *v2* arrays in Fortran code) in the task's data environment by calling the function **`init_again()`**.

The **`target update`** construct assigns the new values of *v1* and *v2* from the task's data environment to the corresponding mapped array sections in the device data environment of the **`target data`** construct.

The task executing on the host device then encounters the second **`target`** region and waits for the completion of the region.

The second **`target`** region uses the updated values of *v1[:N]* and *v2[:N]*.

C / C++

1 *Example target_update.1c*

```
extern void init(float *, float *, int);
extern void init_again(float *, float *, int);
extern void output(float *, int);
void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);
   #pragma omp target data map(to: v1[:N], v2[:N]) map(from: p[0:N])
   {
      #pragma omp target
      #pragma omp parallel for
      for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];
      init_again(v1, v2, N);
      #pragma omp target update to(v1[:N], v2[:N])
      #pragma omp target
      #pragma omp parallel for
      for (i=0; i<N; i++)
        p[i] = p[i] + (v1[i] * v2[i]);
   }
   output(p, N);
}
```

C / C++

Fortran

2 *Example target_update.1f*

```
subroutine vec_mult(p, v1, v2, N)
   real    ::  p(N), v1(N), v2(N)
   integer ::  i
   call init(v1, v2, N)
   !$omp target data map(to: v1, v2) map(from: p)
      !$omp target
      !$omp parallel do
         do i=1,N
            p(i) = v1(i) * v2(i)
         end do
      !$omp end target
      call init_again(v1, v2, N)
      !$omp target update to(v1, v2)
      !$omp target
      !$omp parallel do
         do i=1,N
            p(i) = p(i) + v1(i) * v2(i)
         end do
      !$omp end target
   !$omp end target data
   call output(p, N)
end subroutine
```

Fortran

54.2 `target update` Construct with `if` Clause

The following example shows how the **`target update`** construct updates variables in a device data environment.

The **`target data`** construct maps array sections *v1[:N]* and *v2[:N]* (arrays *v1* and *v2* in the Fortran code) into a device data environment. In between the two **`target`** regions, the task executing on the host device conditionally assigns new values to *v1* and *v2* in the task's data environment. The function **`maybe_init_again()`** returns *true* if new data is written.

When the conditional expression (the return value of **`maybe_init_again()`**) in the **`if`** clause is *true*, the **`target update`** construct assigns the new values of *v1* and *v2* from the task's data environment to the corresponding mapped array sections in the **`target data`** construct's device data environment.

C / C++

Example target_update.2c

```
extern void init(float *, float *, int);
extern int maybe_init_again(float *, int);
extern void output(float *, int);
void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);
   #pragma omp target data map(to: v1[:N], v2[:N]) map(from: p[0:N])
   {
      int changed;
      #pragma omp target
      #pragma omp parallel for
      for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];
      changed = maybe_init_again(v1,  N);
      #pragma omp target update if (changed) to(v1[:N])
      changed = maybe_init_again(v2,  N);
      #pragma omp target update if (changed) to(v2[:N])
      #pragma omp target
      #pragma omp parallel for
      for (i=0; i<N; i++)
        p[i] = p[i] + (v1[i] * v2[i]);
   }
   output(p, N);
}
```

C / C++

Fortran

Example target_update.2f

```
subroutine vec_mult(p, v1, v2, N)
   interface
      logical function maybe_init_again (v1, N)
      real :: v1(N)
      integer :: N
      end function
   end interface
   real    ::  p(N), v1(N), v2(N)
   integer ::  i
   logical :: changed
   call init(v1, v2, N)
   !$omp target data map(to: v1, v2) map(from: p)
      !$omp target
         !$omp parallel do
         do i=1, N
            p(i) = v1(i) * v2(i)
         end do
      !$omp end target
      changed = maybe_init_again(v1, N)
      !$omp target if(changed) update to(v1(:N))
      changed = maybe_init_again(v2, N)
      !$omp target if(changed) update to(v2(:N))
      !$omp target
         !$omp parallel do
         do i=1, N
            p(i) = p(i) + v1(i) * v2(i)
         end do
      !$omp end target
   !$omp end target data
   call output(p, N)
end subroutine
```

Fortran

CHAPTER 55

declare target Construct

55.1 **declare target and end declare target for a Function**

The following example shows how the **declare target** directive is used to indicate that the corresponding call inside a **target** region is to a **fib** function that can execute on the default target device.

A version of the function is also available on the host device. When the **if** clause conditional expression on the **target** construct evaluates to *false*, the **target** region (thus **fib**) will execute on the host device.

For C/C++ codes the declaration of the function **fib** appears between the **declare target** and **end declare target** directives.

C / C++

Example declare_target.1c

```
#pragma omp declare target
extern void fib(int N);
#pragma omp end declare target
#define THRESHOLD 1000000
void fib_wrapper(int n)
{
   #pragma omp target if(n > THRESHOLD)
   {
      fib(n);
   }
}
```

C / C++

The Fortran **fib** subroutine contains a **declare target** declaration to indicate to the compiler to create an device executable version of the procedure. The subroutine name has not been included on the **declare target** directive and is, therefore, implicitly assumed.

The program uses the **module_fib** module, which presents an explicit interface to the compiler with the **declare target** declarations for processing the **fib** call.

Fortran

Example declare_target.1f

```
module module_fib
contains
   subroutine fib(N)
      integer :: N
      !$omp declare target
      !...
   end subroutine
end module
module params
integer :: THRESHOLD=1000000
end module
program my_fib
use params
use module_fib
   !$omp target if( N > THRESHOLD )
      call fib(N)
   !$omp end target
end program
```

Fortran

The next Fortran example shows the use of an external subroutine. Without an explicit interface (through module use or an interface block) the **declare target** declarations within a external subroutine are unknown to the main program unit; therefore, a **declare target** must be provided within the program scope for the compiler to determine that a target binary should be available.

Fortran

Example declare_target.2f

```fortran
program my_fib
integer :: N = 8
!$omp declare target(fib)
   !$omp target
      call fib(N)
   !$omp end target
end program
subroutine fib(N)
integer :: N
!$omp declare target
     print*,"hello from fib"
     !...
end subroutine
```

Fortran

55.2 `declare target` Construct for Class Type

C++

The following example shows how the **`declare target`** and **`end declare target`** directives are used to enclose the declaration of a variable *varY* with a class type **`typeY`**. The member function **`typeY::foo()`** cannot be accessed on a target device because its declaration did not appear between **`declare target`** and **`end declare target`** directives.

Example declare_target.2c

```cpp
struct typeX
{
   int a;
};
class typeY
{
   int a;
  public:
   int foo() { return a^0x01;}
};
#pragma omp declare target
struct typeX varX;  // ok
class typeY varY; // ok if varY.foo() not called on target device
```

```
#pragma omp end declare target
void foo()
{
   #pragma omp target
   {
      varX.a = 100; // ok
      varY.foo(); // error foo() is not available on a target device
   }
}
```

C++

55.3 declare target and end declare target for Variables

The following examples show how the **declare target** and **end declare target** directives are used to indicate that global variables are mapped to the implicit device data environment of each target device.

In the following example, the declarations of the variables *p*, *v1*, and *v2* appear between **declare target** and **end declare target** directives indicating that the variables are mapped to the implicit device data environment of each target device. The **target update** directive is then used to manage the consistency of the variables *p*, *v1*, and *v2* between the data environment of the encountering host device task and the implicit device data environment of the default target device.

C / C++

Example declare_target.3c

```
#define N 1000
#pragma omp declare target
float p[N], v1[N], v2[N];
#pragma omp end declare target
extern void init(float *, float *, int);
extern void output(float *, int);
void vec_mult()
{
   int i;
   init(v1, v2, N);
   #pragma omp target update to(v1, v2)
   #pragma omp target
   #pragma omp parallel for
```

```
    for (i=0; i<N; i++)
      p[i] = v1[i] * v2[i];
    #pragma omp target update from(p)
    output(p, N);
}
```

C / C++

The Fortran version of the above C code uses a different syntax. Fortran modules use a list syntax on the **declare target** directive to declare mapped variables.

Fortran

Example declare_target.3f

```
module my_arrays
!$omp declare target (N, p, v1, v2)
integer, parameter :: N=1000
real               :: p(N), v1(N), v2(N)
end module
subroutine vec_mult()
use my_arrays
   integer :: i
   call init(v1, v2, N);
   !$omp target update to(v1, v2)
   !$omp target
   !$omp parallel do
   do i = 1,N
     p(i) = v1(i) * v2(i)
   end do
   !$omp end target
   !$omp target update from (p)
   call output(p, N)
end subroutine
```

Fortran

The following example also indicates that the function **Pfun()** is available on the target device, as well as the variable *Q*, which is mapped to the implicit device data environment of each target device. The **target update** directive is then used to manage the consistency of the variable *Q* between the data environment of the encountering host device task and the implicit device data environment of the default target device.

In the following example, the function and variable declarations appear between the **declare target** and **end declare target** directives.

C / C++

Example declare_target.4c

```
#define N 10000
#pragma omp declare target
float Q[N][N];
float Pfun(const int i, const int k)
{ return Q[i][k] * Q[k][i]; }
#pragma omp end declare target
float accum(int k)
{
    float tmp = 0.0;
    #pragma omp target update to(Q)
    #pragma omp target
    #pragma omp parallel for reduction(+:tmp)
    for(int i=0; i < N; i++)
        tmp += Pfun(i,k);
    return tmp;
}
```

C / C++

The Fortran version of the above C code uses a different syntax. In Fortran modules a list syntax on the **declare target** directive is used to declare mapped variables and procedures. The *N* and *Q* variables are declared as a comma separated list. When the **declare target** directive is used to declare just the procedure, the procedure name need not be listed – it is implicitly assumed, as illustrated in the **Pfun()** function.

Fortran

Example declare_target.4f

```
module my_global_array
!$omp declare target (N,Q)
integer, parameter :: N=10
real               :: Q(N,N)
contains
function Pfun(i,k)
!$omp declare target
real               :: Pfun
integer,intent(in) :: i,k
   Pfun=(Q(i,k) * Q(k,i))
end function
end module
function accum(k) result(tmp)
use my_global_array
real    :: tmp
integer :: i, k
   tmp = 0.0e0
   !$omp target
   !$omp parallel do reduction(+:tmp)
   do i=1,N
      tmp = tmp + Pfun(k,i)
   end do
   !$omp end target
end function
```

Fortran

55.4 declare target and end declare target with declare simd

The following example shows how the **declare target** and **end declare target** directives are used to indicate that a function is available on a target device. The **declare simd** directive indicates that there is a SIMD version of the function **P()** that is available on the target device as well as one that is available on the host device.

C / C++

Example declare_target.5c

```
#define N 10000
#define M 1024
#pragma omp declare target
float Q[N][N];
#pragma omp declare simd uniform(i) linear(k) notinbranch
float P(const int i, const int k)
{
  return Q[i][k] * Q[k][i];
}
#pragma omp end declare target
float accum(void)
{
  float tmp = 0.0;
  int i, k;
#pragma omp target
#pragma omp parallel for reduction(+:tmp)
  for (i=0; i < N; i++) {
    float tmp1 = 0.0;
#pragma omp simd reduction(+:tmp1)
    for (k=0; k < M; k++) {
      tmp1 += P(i,k);
    }
    tmp += tmp1;
  }
  return tmp;
}
```

C / C++

The Fortran version of the above C code uses a different syntax. Fortran modules use a list syntax of the **declare target** declaration for the mapping. Here the *N* and *Q* variables are declared in the list form as a comma separated list. The function declaration does not use a list and implicitly assumes the function name. In this Fortran example row and column indices are reversed relative to the C/C++ example, as is usual for codes optimized for memory access.

Fortran

Example declare_target.5f

```
module my_global_array
!$omp declare target (N,Q)
integer, parameter :: N=10000, M=1024
real               :: Q(N,N)
contains
function P(k,i)
!$omp declare simd uniform(i) linear(k) notinbranch
!$omp declare target
real               :: P
integer,intent(in) :: k,i
   P=(Q(k,i) * Q(i,k))
end function
end module
function accum() result(tmp)
use my_global_array
real    :: tmp, tmp1
integer :: i
   tmp = 0.0e0
   !$omp target
   !$omp parallel do private(tmp1) reduction(+:tmp)
   do i=1,N
      tmp1 = 0.0e0
      !$omp simd reduction(+:tmp1)
      do k = 1,M
         tmp1 = tmp1 + P(k,i)
      end do
      tmp = tmp + tmp1
   end do
   !$omp end target
end function
```

Fortran

CHAPTER 56

teams Constructs

56.1 target and teams Constructs with omp_get_num_teams and omp_get_team_num Routines

The following example shows how the **target** and **teams** constructs are used to create a league of thread teams that execute a region. The **teams** construct creates a league of at most two teams where the master thread of each team executes the **teams** region.

The **omp_get_num_teams** routine returns the number of teams executing in a **teams** region. The **omp_get_team_num** routine returns the team number, which is an integer between 0 and one less than the value returned by **omp_get_num_teams**. The following example manually distributes a loop across two teams.

C / C++

Example teams.1c

```
#include <stdlib.h>
#include <omp.h>
float dotprod(float B[], float C[], int N)
{
   float sum0 = 0.0;
   float sum1 = 0.0;
   #pragma omp target map(to: B[:N], C[:N])
   #pragma omp teams num_teams(2)
   {
      int i;
      if (omp_get_num_teams() != 2)
         abort();
      if (omp_get_team_num() == 0)
      {
         #pragma omp parallel for reduction(+:sum0)
```

```
            for (i=0; i<N/2; i++)
               sum0 += B[i] * C[i];
         }
         else if (omp_get_team_num() == 1)
         {
            #pragma omp parallel for reduction(+:sum1)
            for (i=N/2; i<N; i++)
               sum1 += B[i] * C[i];
         }
      }
      return sum0 + sum1;
   }
```

C / C++

Fortran

Example teams.1f

```
function dotprod(B,C,N) result(sum)
use omp_lib, ONLY : omp_get_num_teams, omp_get_team_num
    real    :: B(N), C(N), sum,sum0, sum1
    integer :: N, i
    sum0 = 0.0e0
    sum1 = 0.0e0
    !$omp target map(to: B, C)
    !$omp teams num_teams(2)
      if (omp_get_num_teams() /= 2) stop "2 teams required"
      if (omp_get_team_num() == 0) then
         !$omp parallel do reduction(+:sum0)
         do i=1,N/2
            sum0 = sum0 + B(i) * C(i)
         end do
      else if (omp_get_team_num() == 1) then
         !$omp parallel do reduction(+:sum1)
         do i=N/2+1,N
            sum1 = sum1 + B(i) * C(i)
         end do
      end if
    !$omp end teams
    !$omp end target
    sum = sum0 + sum1
end function
```

Fortran

56.2 `target`, `teams`, and `distribute` Constructs

The following example shows how the **`target`**, **`teams`**, and **`distribute`** constructs are used to execute a loop nest in a **`target`** region. The **`teams`** construct creates a league and the master thread of each team executes the **`teams`** region. The **`distribute`** construct schedules the subsequent loop iterations across the master threads of each team.

The number of teams in the league is less than or equal to the variable *num_blocks*. Each team in the league has a number of threads less than or equal to the variable *block_threads*. The iterations in the outer loop are distributed among the master threads of each team.

When a team's master thread encounters the parallel loop construct before the inner loop, the other threads in its team are activated. The team executes the **`parallel`** region and then workshares the execution of the loop.

Each master thread executing the **`teams`** region has a private copy of the variable *sum* that is created by the **`reduction`** clause on the **`teams`** construct. The master thread and all threads in its team have a private copy of the variable *sum* that is created by the **`reduction`** clause on the parallel loop construct. The second private *sum* is reduced into the master thread's private copy of *sum* created by the **`teams`** construct. At the end of the **`teams`** region, each master thread's private copy of *sum* is reduced into the final *sum* that is implicitly mapped into the **`target`** region.

C / C++

Example teams.2c

```
float dotprod(float B[], float C[], int N, int block_size,
  int num_teams, int block_threads)
{
    float sum = 0;
    int i, i0;
    #pragma omp target map(to: B[0:N], C[0:N])
    #pragma omp teams num_teams(num_teams) thread_limit(block_threads) \
      reduction(+:sum)
    #pragma omp distribute
    for (i0=0; i0<N; i0 += block_size)
       #pragma omp parallel for reduction(+:sum)
       for (i=i0; i< min(i0+block_size,N); i++)
           sum += B[i] * C[i];
    return sum;
}
```

C / C++

Fortran

Example teams.2f

```fortran
function dotprod(B,C,N, block_size, num_teams, block_threads) result(sum)
implicit none
    real    :: B(N), C(N), sum
    integer :: N, block_size, num_teams, block_threads, i, i0
    sum = 0.0e0
    !$omp target map(to: B, C)
    !$omp teams num_teams(num_teams) thread_limit(block_threads) &
    !$omp&  reduction(+:sum)
    !$omp distribute
       do i0=1,N, block_size
          !$omp parallel do reduction(+:sum)
          do i = i0, min(i0+block_size,N)
             sum = sum + B(i) * C(i)
          end do
       end do
    !$omp end teams
    !$omp end target
end function
```

Fortran

56.3 `target teams`, and Distribute Parallel Loop Constructs

The following example shows how the **`target teams`** and distribute parallel loop constructs are used to execute a **`target`** region. The **`target teams`** construct creates a league of teams where the master thread of each team executes the **`teams`** region.

The distribute parallel loop construct schedules the loop iterations across the master threads of each team and then across the threads of each team.

C / C++

Example teams.3c

```
float dotprod(float B[], float C[], int N)
{
   float sum = 0;
   int i;
   #pragma omp target teams map(to: B[0:N], C[0:N])
   #pragma omp distribute parallel for reduction(+:sum)
   for (i=0; i<N; i++)
      sum += B[i] * C[i];
   return sum;
}
```

C / C++

Fortran

Example teams.3f

```
function dotprod(B,C,N) result(sum)
   real    :: B(N), C(N), sum
   integer :: N, i
   sum = 0.0e0
   !$omp target teams map(to: B, C)
   !$omp distribute parallel do reduction(+:sum)
      do i = 1,N
         sum = sum + B(i) * C(i)
      end do
   !$omp end teams
   !$omp end target
end function
```

Fortran

56.4 `target` `teams` and Distribute Parallel Loop Constructs with Scheduling Clauses

The following example shows how the **`target`** **`teams`** and distribute parallel loop constructs are used to execute a **`target`** region. The **`teams`** construct creates a league of at most eight teams where the master thread of each team executes the **`teams`** region. The number of threads in each team is less than or equal to 16.

The **`distribute`** parallel loop construct schedules the subsequent loop iterations across the master threads of each team and then across the threads of each team.

The **`dist_schedule`** clause on the distribute parallel loop construct indicates that loop iterations are distributed to the master thread of each team in chunks of 1024 iterations.

The **`schedule`** clause indicates that the 1024 iterations distributed to a master thread are then assigned to the threads in its associated team in chunks of 64 iterations.

C / C++

Example teams.4c

```
#define N 1024*1024
float dotprod(float B[], float C[])
{
    float sum = 0;
    int i;
    #pragma omp target map(to: B[0:N], C[0:N])
    #pragma omp teams num_teams(8) thread_limit(16)
    #pragma omp distribute parallel for reduction(+:sum) \
                dist_schedule(static, 1024) schedule(static, 64)
    for (i=0; i<N; i++)
        sum += B[i] * C[i];
    return sum;
}
```

C / C++

Fortran

Example teams.4f

```
module arrays
integer,parameter :: N=1024*1024
real :: B(N), C(N)
end module
function dotprod() result(sum)
use arrays
   real    :: sum
   integer :: i
   sum = 0.0e0
   !$omp target map(to: B, C)
   !$omp teams num_teams(8) thread_limit(16)
   !$omp distribute parallel do reduction(+:sum) &
   !$omp&  dist_schedule(static, 1024) schedule(static, 64)
      do i = 1,N
         sum = sum + B(i) * C(i)
      end do
   !$omp end teams
   !$omp end target
end function
```

Fortran

56.5 `target teams` and `distribute simd` Constructs

The following example shows how the **`target teams`** and **`distribute simd`** constructs are used to execute a loop in a **`target`** region. The **`target teams`** construct creates a league of teams where the master thread of each team executes the **`teams`** region.

The **`distribute simd`** construct schedules the loop iterations across the master thread of each team and then uses SIMD parallelism to execute the iterations.

C / C++

1 *Example teams.5c*

```
extern void init(float *, float *, int);
extern void output(float *, int);
void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);
   #pragma omp target teams map(to: v1[0:N], v2[:N]) map(from: p[0:N])
   #pragma omp distribute simd
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
   output(p, N);
}
```

C / C++

Fortran

2 *Example teams.5f*

```
subroutine vec_mult(p, v1, v2, N)
   real    ::  p(N), v1(N), v2(N)
   integer ::  i
   call init(v1, v2, N)
   !$omp target teams map(to: v1, v2) map(from: p)
      !$omp distribute simd
         do i=1,N
            p(i) = v1(i) * v2(i)
         end do
   !$omp end target teams
   call output(p, N)
end subroutine
```

Fortran

56.6 `target teams` and Distribute Parallel Loop SIMD Constructs

The following example shows how the **`target teams`** and the distribute parallel loop SIMD constructs are used to execute a loop in a **`target teams`** region. The **`target teams`** construct creates a league of teams where the master thread of each team executes the **`teams`** region.

The distribute parallel loop SIMD construct schedules the loop iterations across the master thread of each team and then across the threads of each team where each thread uses SIMD parallelism.

C / C++

Example teams.6c

```
extern void init(float *, float *, int);
extern void output(float *, int);
void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);
   #pragma omp target teams map(to: v1[0:N], v2[:N]) map(from: p[0:N])
   #pragma omp distribute parallel for simd
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
   output(p, N);
}
```

C / C++

Fortran

Example teams.6f

```
subroutine vec_mult(p, v1, v2, N)
   real    ::  p(N), v1(N), v2(N)
   integer ::  i
   call init(v1, v2, N)
   !$omp target teams map(to: v1, v2) map(from: p)
      !$omp distribute parallel do simd
         do i=1,N
            p(i) = v1(i) * v2(i)
         end do
   !$omp end target teams
   call output(p, N)
end subroutine
```

Fortran

CHAPTER 57

Asynchronous Execution of a `target` Region Using Tasks

The following example shows how the **`task`** and **`target`** constructs are used to execute multiple **`target`** regions asynchronously. The task that encounters the **`task`** construct generates an explicit task that contains a **`target`** region. The thread executing the explicit task encounters a task scheduling point while waiting for the execution of the **`target`** region to complete, allowing the thread to switch back to the execution of the encountering task or one of the previously generated explicit tasks.

C / C++

Example async_target.1c

```
#pragma omp declare target
float F(float);
#pragma omp end declare target
#define N 1000000000
#define CHUNKSZ 1000000
void init(float *, int);
float Z[N];
void pipedF()
{
   int C, i;
   init(Z, N);
   for (C=0; C<N; C+=CHUNKSZ)
   {
      #pragma omp task
      #pragma omp target map(Z[C:CHUNKSZ])
      #pragma omp parallel for
      for (i=0; i<CHUNKSZ; i++)
         Z[i] = F(Z[i]);
   }
   #pragma omp taskwait
}
```

C / C++

The Fortran version has an interface block that contains the **declare target**. An identical statement exists in the function declaration (not shown here).

Fortran

Example async_target.1f

```fortran
module parameters
integer, parameter :: N=1000000000, CHUNKSZ=1000000
end module
subroutine pipedF()
use parameters, ONLY: N, CHUNKSZ
integer            :: C, i
real               :: z(N)

interface
   function F(z)
   !$omp declare target
     real, intent(IN) ::z
     real             ::F
   end function F
end interface

   call init(z,N)

   do C=1,N,CHUNKSZ

      !$omp task
      !$omp target map(z(C:C+CHUNKSZ-1))
      !$omp parallel do
         do i=C,C+CHUNKSZ-1
            z(i) = F(z(i))
         end do
      !$omp end target
      !$omp end task

   end do
   !$omp taskwait
   print*, z

end subroutine pipedF
```

Fortran

The following example shows how the **task** and **target** constructs are used to execute multiple **target** regions asynchronously. The task dependence ensures that the storage is allocated and initialized on the device before it is accessed.

C / C++

Example async_target.2c

```
#include <stdlib.h>
#include <omp.h>
#pragma omp declare target
extern void init(float *, float *, int);
#pragma omp end declare target
extern void foo();
extern void output(float *, int);
void vec_mult(float *p, int N, int dev)
{
   float *v1, *v2;
   int i;
   #pragma omp task shared(v1, v2) depend(out: v1, v2)
   #pragma omp target device(dev) map(v1, v2)
   {
       // check whether on device dev
       if (omp_is_initial_device())
          abort();
       v1 = malloc(N*sizeof(float));
       v2 = malloc(N*sizeof(float));
       init(v1, v2, N);
   }
   foo(); // execute other work asychronously
   #pragma omp task shared(v1, v2, p) depend(in: v1, v2)
   #pragma omp target device(dev) map(to: v1, v2) map(from: p[0:N])
   {
       // check whether on device dev
       if (omp_is_initial_device())
          abort();
       #pragma omp parallel for
       for (i=0; i<N; i++)
         p[i] = v1[i] * v2[i];
       free(v1);
       free(v2);
   }
   #pragma omp taskwait
   output(p, N);
}
```

C / C++

The Fortran example below is similar to the C version above. Instead of pointers, though, it uses the convenience of Fortran allocatable arrays on the device. An allocatable array has the same behavior in a **map** clause as a C pointer, in this case.

If there is no shape specified for an allocatable array in a **map** clause, only the array descriptor (also called a dope vector) is mapped. That is, device space is created for the descriptor, and it is initially populated with host values. In this case, the *v1* and *v2* arrays will be in a non-associated state on the device. When space for *v1* and *v2* is allocated on the device the addresses to the space will be included in their descriptors.

At the end of the first **target** region, the descriptor (of an unshaped specification of an allocatable array in a **map** clause) is returned with the raw device address of the allocated space. The content of the array is not returned. In the example the data in arrays *v1* and *v2* are not returned. In the second **target** directive, the *v1* and *v2* descriptors are re-created on the device with the descriptive information; and references to the vectors point to the correct local storage, of the space that was not freed in the first **target** directive. At the end of the second **target** region, the data in array *p* is copied back to the host since *p* is not an allocatable array.

A **depend** clause is used in the **task** directive to provide a wait at the beginning of the second **target** region, to insure that there is no race condition with *v1* and *v2* in the two tasks. It would be noncompliant to use *v1* and/or *v2* in lieu of *N* in the **depend** clauses, because the use of non-allocated allocatable arrays as list items in the first **depend** clause would lead to unspecified behavior.

Fortran

Example async_target.2f

```
subroutine mult(p,  N, idev)
  use omp_lib, ONLY: omp_is_initial_device
  real              :: p(N)
  real,allocatable :: v1(:), v2(:)
  integer ::  i, idev
  !$omp declare target (init)

  !$omp task shared(v1,v2) depend(out: N)
     !$omp target device(idev) map(v1,v2)
        if( omp_is_initial_device() ) &
           stop "not executing on target device"
        allocate(v1(N), v2(N))
        call init(v1,v2,N)
     !$omp end target
  !$omp end task

  call foo()  ! execute other work asychronously

  !$omp task shared(v1,v2,p) depend(in: N)
     !$omp target device(idev) map(to: v1,v2) map(from: p)
        if( omp_is_initial_device() ) &
           stop "not executing on target device"
        !$omp parallel do
           do i = 1,N
```

```
               p(i) = v1(i) * v2(i)
            end do
         deallocate(v1,v2)

      !$omp end target
   !$omp end task

   !$omp taskwait
   call output(p, N)

end subroutine
```

Fortran

CHAPTER 58

Array Sections in Device Constructs

The following examples show the usage of array sections in **map** clauses on **target** and **target data** constructs.

This example shows the invalid usage of two seperate sections of the same array inside of a **target** construct.

C / C++

Example array_sections.1c

```
void foo ()
{
   int A[30];
#pragma omp target data map( A[0:4] )
{
   /* Cannot map distinct parts of the same array */
   #pragma omp target map( A[7:20] )
   {
      A[2] = 0;
   }
}
}
```

C / C++

Fortran

Example array_sections.1f

```
subroutine foo()
integer :: A(30)
   A = 1
   !$omp target data map( A(1:4) )
     ! Cannot map distinct parts of the same array
     !$omp target map( A(8:27) )
        A(3) = 0
     !$omp end target map
   !$omp end target data
end subroutine
```

Fortran

This example shows the invalid usage of two separate sections of the same array inside of a **target** construct.

C / C++

Example array_sections.2c

```
void foo ()
{
   int A[30], *p;
#pragma omp target data map( A[0:4] )
{
   p = &A[0];
   /* invalid because p[3] and A[3] are the same
    * location on the host but the array section
    * specified via p[...] is not a subset of A[0:4] */
   #pragma omp target map( p[3:20] )
   {
      A[2] = 0;
      p[8] = 0;
   }
}
}
```

C / C++

Fortran

Example array_sections.2f

```
subroutine foo()
integer,target  :: A(30)
integer,pointer :: p(:)
   A=1
   !$omp target data map( A(1:4) )
     p=>A
     ! invalid because p(4) and A(4) are the same
     ! location on the host but the array section
     ! specified via p(...) is not a subset of A(1:4)
     !$omp target map( p(4:23) )
        A(3) = 0
        p(9) = 0
     !$omp end target
   !$omp end target data
end subroutine
```

Fortran

This example shows the valid usage of two separate sections of the same array inside of a **target** construct.

C / C++

Example array_sections.3c

```
void foo ()
{
   int A[30], *p;
#pragma omp target data map( A[0:4] )
{
   p = &A[0];
   #pragma omp target map( p[7:20] )
   {
      A[2] = 0;
      p[8] = 0;
   }
}
}
```

C / C++

Fortran

1 *Example array_sections.3f*

```
subroutine foo()
integer,target  :: A(30)
integer,pointer :: p(:)
   !$omp target data map( A(1:4) )
     p=>A
     !$omp target map( p(8:27) )
        A(3)  = 0
        p(9)  = 0
     !$omp end target map
   !$omp end target data
end subroutine
```

Fortran

2 This example shows the valid usage of a wholly contained array section of an already mapped array
3 section inside of a **target** construct.

C / C++

4 *Example array_sections.4c*

```
void foo ()
{
   int A[30], *p;
#pragma omp target data map( A[0:10] )
{
   p = &A[0];
   #pragma omp target map( p[3:7] )
   {
      A[2] = 0;
      p[8] = 0;
      A[8] = 1;
   }
}
}
```

C / C++

Fortran

Example array_sections.4f

```
subroutine foo()
integer,target  :: A(30)
integer,pointer :: p(:)
   !$omp target data map( A(1:10) )
     p=>A
     !$omp target map( p(4:10) )
        A(3) = 0
        p(9) = 0
        A(9) = 1
     !$omp end target
   !$omp end target data
end subroutine
```

Fortran

CHAPTER 59

Device Routines

59.1 `omp_is_initial_device` Routine

The following example shows how the **`omp_is_initial_device`** runtime library routine can be used to query if a code is executing on the initial host device or on a target device. The example then sets the number of threads in the **`parallel`** region based on where the code is executing.

C / C++

Example device.1c

```
#include <stdio.h>
#include <omp.h>
#pragma omp declare target
void vec_mult(float *p, float *v1, float *v2, int N);
extern float *p, *v1, *v2;
extern int N;
#pragma omp end declare target
extern void init_vars(float *, float *, int);
extern void output(float *, int);
void foo()
{
   init_vars(v1, v2, N);
   #pragma omp target device(42) map(p[:N], v1[:N], v2[:N])
   {
      vec_mult(p, v1, v2, N);
   }
   output(p, N);
}
void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   int nthreads;
   if (!omp_is_initial_device())
   {
      printf("1024 threads on target device\n");
      nthreads = 1024;
   }
   else
   {
      printf("8 threads on initial device\n");
      nthreads = 8;
   }
   #pragma omp parallel for private(i) num_threads(nthreads);
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
}
```

C / C++

Fortran

Example device.1f

```
module params
   integer,parameter :: N=1024
end module params
module vmult
contains
   subroutine vec_mult(p, v1, v2, N)
   use omp_lib, ONLY : omp_is_initial_device
   !$omp declare target
   real    :: p(N), v1(N), v2(N)
   integer :: i, nthreads, N
      if (.not. omp_is_initial_device()) then
         print*, "1024 threads on target device"
         nthreads = 1024
      else
         print*, "8 threads on initial device"
         nthreads = 8
      endif
      !$omp parallel do private(i) num_threads(nthreads)
      do i = 1,N
        p(i) = v1(i) * v2(i)
      end do
   end subroutine vec_mult
end module vmult
program prog_vec_mult
use params
use vmult
real :: p(N), v1(N), v2(N)
   call init(v1,v2,N)
   !$omp target device(42) map(p, v1, v2)
      call vec_mult(p, v1, v2, N)
   !$omp end target
   call output(p, N)
end program
```

Fortran

59.2 omp_get_num_devices Routine

The following example shows how the **omp_get_num_devices** runtime library routine can be used to determine the number of devices.

C / C++

Example device.2c

```
#include <omp.h>
extern void init(float *, float *, int);
extern void output(float *, int);
void vec_mult(float *p, float *v1, float *v2, int N)
{
   int i;
   init(v1, v2, N);
   int ndev = omp_get_num_devices();
   int do_offload = (ndev>0 && N>1000000);
   #pragma omp target if(do_offload) map(to: v1[0:N], v2[:N]) map(from: p[0:N])
   #pragma omp parallel for if(N>1000) private(i)
   for (i=0; i<N; i++)
     p[i] = v1[i] * v2[i];
   output(p, N);
}
```

C / C++

Fortran

Example device.2f

```fortran
subroutine vec_mult(p, v1, v2, N)
use omp_lib, ONLY : omp_get_num_devices
real    :: p(N), v1(N), v2(N)
integer :: N, i, ndev
logical :: do_offload
   call init(v1, v2, N)
   ndev = omp_get_num_devices()
   do_offload = (ndev>0) .and. (N>1000000)
   !$omp target if(do_offload) map(to: v1, v2) map(from: p)
   !$omp parallel do if(N>1000)
      do i=1,N
         p(i) = v1(i) * v2(i)
      end do
   !$omp end target
   call output(p, N)
end subroutine
```

Fortran

59.3 omp_set_default_device and omp_get_default_device Routines

The following example shows how the **omp_set_default_device** and **omp_get_default_device** runtime library routines can be used to set the default device and determine the default device respectively.

C / C++

Example device.3c

```c
#include <omp.h>
#include <stdio.h>
void foo(void)
{
   int default_device = omp_get_default_device();
   printf("Default device = %d\n", default_device);
   omp_set_default_device(default_device+1);
   if (omp_get_default_device() != default_device+1)
      printf("Default device is still = %d\n", default_device);
}
```

C / C++

Fortran

1 *Example device.3f*

```
program foo
use omp_lib, ONLY : omp_get_default_device, omp_set_default_device
integer :: old_default_device, new_default_device
   old_default_device = omp_get_default_device()
   print*, "Default device = ", old_default_device
   new_default_device = old_default_device + 1
   call omp_set_default_device(new_default_device)
   if (omp_get_default_device() == old_default_device) &
      print*,"Default device is STILL = ", old_default_device
end program
```

Fortran

CHAPTER 60

Fortran ASSOCIATE Construct

Fortran

The following is an invalid example of specifying an associate name on a data-sharing attribute clause. The constraint in the Data Sharing Attribute Rules section in the OpenMP 4.0 API Specifications states that an associate name preserves the association with the selector established at the **ASSOCIATE** statement. The associate name *b* is associated with the shared variable *a*. With the predetermined data-sharing attribute rule, the associate name *b* is not allowed to be specified on the **private** clause.

Example associate.1f

```
      program example
      real :: a, c
      associate (b => a)
!$omp parallel private(b, c)        ! invalid to privatize b
      c = 2.0*b
!$omp end parallel
      end associate
      end program
```

In next example, within the **parallel** construct, the association name *thread_id* is associated with the private copy of *i*. The print statement should output the unique thread number.

Example associate.2f

```
      program example
      use omp_lib
      integer  i
!$omp parallel private(i)
      i = omp_get_thread_num()
      associate(thread_id => i)
        print *, thread_id       ! print private i value
      end associate
!$omp end parallel
      end program
```

The following example illustrates the effect of specifying a selector name on a data-sharing attribute clause. The associate name *u* is associated with *v* and the variable *v* is specified on the **`private`** clause of the **`parallel`** construct. The construct association is established prior to the **`parallel`** region. The association between *u* and the original *v* is retained (see the Data Sharing Attribute Rules section in the OpenMP 4.0 API Specifications). Inside the **`parallel`** region, *v* has the value of -1 and *u* has the value of the original *v*.

Example associate.3f

```
program example
  integer :: v
  v = 15
associate(u => v)
!$omp parallel private(v)
  v = -1
  print *, v                ! private v=-1
  print *, u                ! original v=15
!$omp end parallel
end associate
end program
```

Fortran

APPENDIX A

Document Revision History

A.1 Changes from 4.0.1 to 4.0.2

- Names of examples were changed from numbers to mnemonics
- Added SIMD examples (Section 51 on page 182)
- Applied miscellaneous fixes in several source codes
- Added the revision history

A.2 Changes from 4.0 to 4.0.1

Added the following new examples:

- the **`proc_bind`** clause (Section 8 on page 22)
- the **`taskgroup`** construct (Section 18 on page 73)

A.3 Changes from 3.1 to 4.0

Beginning with OpenMP 4.0, examples were placed in a separate document from the specification document.

Version 4.0 added the following new examples:

- task dependences (Section 17 on page 66)
- cancellation constructs (Section 30 on page 114)
- **`target`** construct (Section 52 on page 193)
- **`target data`** construct (Section 53 on page 200)
- **`target update`** construct (Section 54 on page 212)
- **`declare target`** construct (Section 55 on page 216)
- **`teams`** constructs (Section 56 on page 224)
- asynchronous execution of a **`target`** region using tasks (Section 57 on page 233)
- array sections in device constructs (Section 58 on page 238)
- device runtime routines (Section 59 on page 243)
- Fortran ASSOCIATE construct (Section 60 on page 248)