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

Copyright © 1997-2015 OpenMP Architecture Review Board.
Permission to copy without fee all or part of this material is granted, provided the OpenMP Architecture Review Board copyright notice and the title of this document appear. Notice is given that copying is by permission of OpenMP Architecture Review Board.
This page intentionally left blank
# Contents

1 A Simple Parallel Loop .................................................. 3

2 The OpenMP Memory Model ............................................. 4

3 Conditional Compilation ................................................ 10

4 Internal Control Variables (ICVs) .................................... 11

5 The parallel Construct ................................................... 14

6 Controlling the Number of Threads on Multiple Nesting Levels .... 17

7 Interaction Between the num_threads Clause and omp_set_dynamic ........ 20

8 The proc_bind Clause ..................................................... 22
   8.1 Spread Affinity Policy ........................................... 22
   8.2 Close Affinity Policy ........................................... 25
   8.3 Master Affinity Policy ........................................... 27

9 Fortran Restrictions on the do Construct .......................... 29

10 Fortran Private Loop Iteration Variables ........................... 31

11 The nowait Clause ....................................................... 33

12 The collapse Clause ..................................................... 37

13 The parallel sections Construct ...................................... 41

14 The firstprivate Clause and the sections Construct ................ 43

15 The single Construct ................................................... 45
34 The default (none) Clause
35 Race Conditions Caused by Implied Copies of Shared Variables in Fortran
36 The private Clause
37 Fortran Restrictions on Storage Association with the private Clause
38 C/C++ Arrays in a firstprivate Clause
39 The lastprivate Clause
40 The reduction Clause
41 The copyin Clause
42 The copyprivate Clause
43 Nested Loop Constructs
44 Restrictions on Nesting of Regions
45 The omp_set_dynamic and omp_set_num_threads Routines
46 The omp_get_num_threads Routine
47 The omp_init_lock Routine
48 Ownership of Locks
49 Simple Lock Routines
50 Nestable Lock Routines
51 SIMD Constructs
52 target Construct
   52.1 target Construct on parallel Construct
   52.2 target Construct with map Clause

Contents
52.3 *map* Clause with *to/from* map-types
52.4 *map* Clause with Array Sections
52.5 *target* Construct with *if* Clause

53 *target data* Construct
53.1 Simple *target data* Construct
53.2 *target data* Region Enclosing Multiple *target* Regions
53.3 *target data* Construct with Orphaned Call
53.4 *target data* Construct with *if* Clause

54 *target update* Construct
54.1 Simple *target data* and *target update* Constructs
54.2 *target update* Construct with *if* Clause

55 *declare target* Construct
55.1 *declare target* and *end declare target* for a Function
55.2 *declare target* Construct for Class Type
55.3 *declare target* and *end declare target* for Variables
55.4 *declare target* and *end declare target* with *declare simd*

56 *teams* Constructs
56.1 *target* and *teams* Constructs with *omp_get_num_teams* and *omp_get_team_num* Routines
56.2 *target, teams*, and *distribute* Constructs
56.3 *target teams*, and Distribute Parallel Loop Constructs
56.4 *target teams* and Distribute Parallel Loop Constructs with Scheduling Clauses
56.5 *target teams* and *distribute simd* Constructs
56.6 *target teams* and Distribute Parallel Loop SIMD Constructs

57 Asynchronous Execution of a *target* Region Using Tasks

58 Array Sections in Device Constructs

59 Device Routines
59.1 *omp_is_initial_device* Routine
59.2 *omp_get_num_devices* Routine
59.3 `omp_set_default_device` and
`omp_get_default_device` Routines ...................................... 246

60 Fortran ASSOCIATE Construct .............................................. 248

A Document Revision History .................................................. 250

A.1 Changes from 4.0.1 to 4.0.2 ................................................ 250
A.2 Changes from 4.0 to 4.0.1 ................................................ 250
A.3 Changes from 3.1 to 4.0 ................................................ 250
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.

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

```fortran
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
```
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.

```
#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
    }
    #pragma omp parallel num_threads(2) shared(x)
    {
        if (omp_get_thread_num() == 0) {
```
The following example demonstrates why synchronization is difficult to perform correctly through variables. The value of flag is undefined in both prints on thread 1 and the value of data is only well-defined in the second print.
Example mem_model.2c

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

Example mem_model.2f

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

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

The next example demonstrates why synchronization is difficult to perform correctly through variables. Because the `write(1)-flush(1)-flush(2)-read(2)` sequence cannot be guaranteed in the example, the statements on thread 0 and thread 1 may execute in either order.
Example 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");
        }
        else if(omp_get_thread_num()==2)
        {
            /* Loop until we see that flag reaches 2 */
            #pragma omp flush(flag)
            while(flag < 2)
            {
                #pragma omp flush(flag)
            }
            printf("Thread 2 awoken\n");
        }
        return 0;
    }
}
Example 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'
  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
```
Conditional Compilation

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

`Example cond_comp.1c`

```c
#include <stdio.h>

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

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`

```fortran
PROGRAM EXAMPLE
C234567890
!$ PRINT *, "Compiled by an OpenMP-compliant implementation."
END PROGRAM EXAMPLE
```
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.

Example icv.1c

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

int main (void)
{
    omp_set_nested(1);
    omp_set_max_active_levels(8);
    omp_set_dynamic(0);
    omp_set_num_threads(2);
    #pragma omp parallel
    {
        omp_set_num_threads(3);
        #pragma omp parallel
        {
            omp_set_num_threads(4);
            #pragma omp single
            {
                /* The following should print:
                   * Inner: max_act_level=8, num_thds=3, max_thds=4
                   * Inner: max_act_level=8, num_thds=3, max_thds=4
                */
                printf ("Inner: max_act_level=%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_level=8, num_thds=2, max_thds=3
            */
            printf ("Outer: max_act_level=%d, num_thds=%d, max_thds=%d\n",
                    omp_get_max_active_levels(), omp_get_num_threads(),
                    omp_get_max_threads());
        }
    }
```

Example icv.f

```fortran
program icv
  use omp_lib
  call omp_set_nested(.true.)
  call omp_set_max_active_levels(8)
  call omp_set_dynamic(.false.)
  call omp_set_num_threads(2)
  call omp_set_num_threads(3)
  call omp_set_num_threads(4)
  !$omp single
    print *, "Inner: max_act_lev=", omp_get_max_active_levels(),
    & " num_thds=", omp_get_num_threads(),
    & " max_thds=", omp_get_max_threads()
  !$omp end single
  !$omp parallel
    print *, "Outer: max_act_lev=", omp_get_max_active_levels(),
    & " num_thds=", omp_get_num_threads(),
    & " max_thds=", omp_get_max_threads()
  !$omp end parallel
end
```

CHAPTER 4. INTERNAL CONTROL VARIABLES (ICVS)
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:

\[\begin{align*}
\text{C / C++} \\
\text{Example parallel.1c}
\end{align*}\]

\begin{verbatim}
#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);
    }
}
\end{verbatim}
```c
int main()
{
    float array[10000];
    sub(array, 10000);
    return 0;
}
```

---

**Example parallel.1f**

```fortran
SUBROUTINE SUBDOMAIN(X, ISTART, IPOINTS)
    INTEGER ISTART, IPOINTS
    REAL X(*)
    INTEGER I
    DO 100 I=1,IPOINTS
        X(ISTART+I) = 123.456
    100 CONTINUE
END SUBROUTINE SUBDOMAIN

SUBROUTINE SUB(X, NPOINTS)
    INCLUDE "omp_lib.h" ! or USE OMP_LIB
    REAL X(*)
    INTEGER NPOINTS
    INTEGER IAM, NT, IPOINTS, ISTART
    !$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,NPOINTS)
    IAM = OMP_GET_THREAD_NUM()
    NT = OMP_GET_NUM_THREADS()
    IPOINTS = NPOINTS/NT
    ISTART = IAM * IPOINTS
    IF (IAM .EQ. NT-1) THEN
        IPOINTS = NPOINTS - ISTART
    ENDIF
    CALL SUBDOMAIN(X,ISTART,IPOINTS)
    !$OMP END PARALLEL
END SUBROUTINE SUB
```
PROGRAM PAREXAMPLE
REAL ARRAY(10000)
CALL SUB(ARRAY, 10000)
END PROGRAM PAREXAMPLE
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:

---

**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
*/
      printf ("Inner: num_thds=%d\n", omp_get_num_threads());
      }
    }
/*
If nesting is not supported, the following should print:
* Inner: num_thds=1
* Inner: num_thds=1
*/
  }
}
```

---
S-25 #pragma omp barrier
S-26 omp_set_nested(0);
S-27 #pragma omp parallel
S-28 {
S-29     #pragma omp single
S-30     {
S-31         /*
S-32         * Even if OMP_NUM_THREADS=2,3 was set, the following should
S-33         * print, because nesting is disabled:
S-34         * Inner: num_thds=1
S-35         * Inner: num_thds=1
S-36         */
S-37         printf("Inner: num_thds=%d\n", omp_get_num_threads());
S-38     }
S-39 }
S-40 #pragma omp barrier
S-41 #pragma omp single
S-42 {
S-43     /*
S-44     * If OMP_NUM_THREADS=2,3 was set, the following should print:
S-45     * Outer: num_thds=2
S-46     */
S-47     printf("Outer: num_thds=%d\n", omp_get_num_threads());
S-48 }
S-49 }
S-50 return 0;
S-51 }

Example nthrs_nesting.lf

S-1 program icv
S-2 use omp_lib
S-3 call omp_set_nested(.true.)
S-4 call omp_set_dynamic(.false.)
S-5 !$omp parallel
S-6 !$omp parallel
S-7 !$omp single
S-8     ! If OMP_NUM_THREADS=2,3 was set, the following should print:
S-9     ! Inner: num_thds= 3
S-10     ! Inner: num_thds= 3
S-11     ! If nesting is not supported, the following should print:
S-12     ! Inner: num_thds= 1
S-13     ! Inner: num_thds= 1
S-14     printf *, "Inner: num_thds=", omp_get_num_threads()
S-15 !$omp end 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()
```

If OMP_NUM_THREADS=2,3 was set, the following should print:

```
! Outer: num_thds= 2
print *, "Outer: num_thds=", omp_get_num_threads()
```

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.

```
#include <omp.h>

int main() {
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(10)
    {
        /* do work here */
    }
    return 0;
}
```
Example nthrs_dynamic.1f

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

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

Example nthrs_dynamic.2c

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

Example nthrs_dynamic.2f

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

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

CHAPTER 7. INTERACTION BETWEEN THE NUM_THREADS CLAUSE AND OMP_SET_DYNAMIC 21
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
S-1 void work();
S-2 int main()
S-3 {
S-4 #pragma omp parallel proc_bind(spread) num_threads(4)
S-5 {
S-6     work();
S-7 }
S-8     return 0;
S-9 }
```

```
C / C++

Fortran

Example affinity.1f
S-1 PROGRAM EXAMPLE
S-2 !$OMP PARALLEL PROC_BIND(SPREAD) NUM_THREADS(4)
S-3 CALL WORK()
S-4 !$OMP END PARALLEL
S-5 END PROGRAM EXAMPLE
```

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 \texttt{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.

\begin{verbatim}
Example affinity.2c

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

\begin{verbatim}
Example affinity.2f

subroutine foo
!$omp parallel num_threads(16) proc_bind(spread)
call work()
!$omp end parallel
end subroutine
\end{verbatim}

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

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

---

Example affinity.4c

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

---

Example affinity.4f

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

Fortran

Example affinity.5f

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

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
Fortran Restrictions on the do Construct

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

Example fort_do.f

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

SUBROUTINE DO_GOOD()
INTEGER I, J
REAL A(1000)
DO 100 I = 1, 10
!$OMP DO
DO 100 J = 1, 10
CALL WORK(I, J)
100 CONTINUE ! !$OMP ENDDO implied here

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

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

29
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 Private Loop Iteration Variables

In general loop iteration variables will be private, when used in the *do-loop* of a *do* and *parallel do* construct or in sequential loops in a *parallel* construct (see 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*

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

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

*Example fort_loopvar.2f*

```fortran
SUBROUTINE PLOOP_2(A,B,N,I1,I2)
  REAL A(*), B(*)
  INTEGER I1, I2, N
  !$OMP PARALLEL SHARED(A,B,I1,I2)
  !$OMP SECTIONS
```

Note however that the use of shared loop iteration variables can easily lead to race conditions.
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
#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]);
    }
}
```
In the following example, static scheduling distributes the same logical iteration numbers to the threads that execute the three loop regions. This allows the `nowait` clause to be used, even though there is a data dependence between the loops. The dependence is satisfied as long the same thread executes the same logical iteration numbers in each loop.

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

```c
#include <math.h>

void nowait_example2(int n, float *a, float *b, float *c, float *y, float *z)
{
    int i;

    #pragma omp parallel
    {
        #pragma omp for schedule(static) nowait
        for (i=0; i<n; i++)
            c[i] = (a[i] + b[i]) / 2.0f;

        #pragma omp for schedule(static) nowait
        for (i=0; i<n; i++)
            z[i] = sqrtf(c[i]);

        #pragma omp for schedule(static) nowait
        for (i=1; i<=n; i++)
            y[i] = z[i-1] + a[i];
    }
}
```

Example nowait.2f

```fortran
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
    !$OMP END DO NOWAIT
```
S-21  !$OMP END PARALLEL
S-22  END SUBROUTINE NOWAIT_EXAMPLE2

Fortran
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.

---

**Example collapse.1c**

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

---
Fortran

Example collapse.1f

```fortran
subroutine sub(a)
  real a(*)
  integer kl, ku, ks, jl, ju, js, il, iu, is
  common /csub/ kl, ku, ks, jl, ju, js, il, iu, is
  integer i, j, k
  !$omp do collapse(2) private(i,j,k)
  do k = kl, ku, ks
    do j = jl, ju, js
      do i = il, iu, is
        call bar(a,i,j,k)
      enddo
    enddo
  enddo
  !$omp end do
end subroutine
```

In the next example, the \texttt{k} and \texttt{j} loops are associated with the loop construct. So the iterations of the \texttt{k} and \texttt{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 \texttt{k} and \texttt{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, \texttt{k} will have the value 2 and \texttt{j} will have the value 3. Since \texttt{klast} and \texttt{jlast} are \texttt{lastprivate}, their values are assigned by the sequentially last iteration of the collapsed \texttt{k} and \texttt{j} loop. This example prints: 2 3.

C / C++

Example collapse.2c

```c
#include <stdio.h>
void test()
{
  int j, k, jlast, klast;
  #pragma omp parallel
  {
    #pragma omp for collapse(2) lastprivate(jlast, klast)
    for (k=1; k<=2; k++)
      for (j=1; j<=3; j++)
        { jlast=j; klast=k; }
    #pragma omp single
  }
```

OpenMP Examples Version 4.0.2 - March 2015
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
```
Example collapse.3c

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

Example collapse.3f

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

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

Example psections.1c

```c
void XAXIS();
void YAXIS();
void ZAXIS();

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

S-1       SUBROUTINE SECT_EXAMPLE()
S-2       !$OMP PARALLEL SECTIONS
S-3       !$OMP SECTION
S-4       CALL XAXIS()
S-5       !$OMP SECTION
S-6       CALL YAXIS()
S-7
S-8       !$OMP SECTION
S-9       CALL ZAXIS()
S-10
S-11      !$OMP END PARALLEL SECTIONS
S-12      END SUBROUTINE SECT_EXAMPLE
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

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

```c
```
/* 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
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.

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");
        work2();
        #pragma omp single nowait
        printf("Finished work1 and beginning work2.\n");
        work2();
    }
}
```
Example single.1f

S-1    SUBROUTINE WORK1()
S-2    END SUBROUTINE WORK1
S-3
S-4    SUBROUTINE WORK2()
S-5    END SUBROUTINE WORK2
S-6
S-7    PROGRAM SINGLE_EXAMPLE
S-8    !$OMP PARALLEL
S-9
S-10   !$OMP SINGLE
S-11    print *, "Beginning work1."
S-12   !$OMP END SINGLE
S-13
S-14    CALL WORK1()
S-15
S-16   !$OMP SINGLE
S-17    print *, "Finishing work1."
S-18   !$OMP END SINGLE
S-19
S-20   !$OMP SINGLE
S-21    print *, "Finished work1 and beginning work2."
S-22   !$OMP END SINGLE NOWAIT
S-23
S-24    CALL WORK2()
S-25
S-26   !$OMP END PARALLEL
S-27
S-28    END PROGRAM SINGLE_EXAMPLE
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
S-1 struct node {
S-2     struct node *left;
S-3     struct node *right;
S-4 };
S-5 extern void process(struct node *);
S-6 void traverse( struct node *p ) {
S-7     if (p->left)
S-8         #pragma omp task // p is firstprivate by default
S-9         traverse(p->left);
S-10    if (p->right)
S-11        #pragma omp task // p is firstprivate by default
S-12        traverse(p->right);
S-13    process(p);
S-14 }
```
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
  !$OMP TASK ! P is firstprivate by default
  call traverse(P%left)
ENDIF
IF (associated(P%right)) THEN
  !$OMP TASK ! P is firstprivate by default
  call traverse(P%right)
ENDIF
CALL process ( P )
END SUBROUTINE

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++
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.
node * p = head;
while (p) {
  #pragma omp task
  // p is firstprivate by default
  process(p);
  p = p->next;
}
}

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
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
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;
    }
}
```

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

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.
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.
Example tasking.7c

S-1
int tp;
S-2
#pragma omp threadprivate(tp)
S-3
int var;
S-4
void work()
S-5
{
S-6    #pragma omp task
S-7    {
S-8        /* do work here */
S-9    #pragma omp task
S-10   {
S-11        tp = 1;
S-12        /* do work here */
S-13    #pragma omp task
S-14   {
S-15        /* no modification of tp */
S-16        var = tp; // value of tp can be 1 or 2
S-17        }  
S-18        tp = 2;
S-19   }
S-20    }  
S-21    }
S-22}

Example tasking.7f

S-1
module example
S-2    integer tp
S-3  !$omp threadprivate(tp)
S-4    integer var
S-5
S-6    contains
S-7  !$omp task
S-8  ! do work here
S-9  !$omp task
S-10     tp = 1
S-11  ! do work here
S-12  !$omp task
S-13     ! no modification of tp
S-14  !$omp end task
S-15     var = tp    ! value of var can be 1 or 2
S-16  !$omp end task
S-17     tp = 2
In this example, scheduling constraints prohibit a thread in the team from executing a new task that modifies `tp` while another such task region tied to the same thread is suspended. Therefore, the value written will persist across the task scheduling point.

Example tasking.8c

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

C / C++
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.
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.
**Example tasking.10c**

```
#include <omp.h>
void work() {
  omp_lock_t lock;
  omp_init_lock(&lock);
  // Task Scheduling Point 1
  omp_set_lock(&lock);
  // Capture data for the following task
  omp_unset_lock(&lock);
  // Lock is shared by default in the task
  omp_destroy_lock(&lock);
}
```

**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 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
```

CHAPTER 16. THE TASK AND TASKWAIT CONSTRUCTS
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++

```c
#include <stdio.h>

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

---

OpenMP Examples Version 4.0.2 - March 2015
Example tasking.11f

```fortran
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
```

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

Example tasking.12c

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

use omp_lib
integer :: pos, n
cor character, pointer :: state(:)
cor character, target, dimension(n) :: new_state1, new_state2
cor integer, parameter :: LIMIT = 3
cor if (pos .eq. n) then
cor call check_solution(state)
cor return
cor endif
cor !$omp task final(pos > LIMIT) mergeable
cor if (.not. omp_in_final()) then
cor new_state1(1:pos) = state(1:pos)
cor state => new_state1
cor endif
cor state(pos+1) = 'z'
cor call bin_search(pos+1, n, state)
cor !$omp end task
cor !$omp task final(pos > LIMIT) mergeable
cor if (.not. omp_in_final()) then
cor new_state2(1:pos) = state(1:pos)
cor state => new_state2
cor endif
cor state(pos+1) = 'y'
cor call bin_search(pos+1, n, state)
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
void foo ( )
{
    int i;
    #pragma omp task if(0) // This task is undeferred
    {
        #pragma omp task // This task is a regular task
        for (i = 0; i < 3; i++) {
            #pragma omp task // This task is a regular task
            bar();
        }
    }
    #pragma omp task final(1) // This task is a regular task
    {
        #pragma omp task // This task is included
        for (i = 0; i < 3; i++) {
            #pragma omp task // This task is also included
            bar();
        }
    }
}```
Example tasking.14f

```fortran
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
```

CHAPTER 16. THE **TASK** AND **TASKWAIT** CONSTRUCTS
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.

Example task_dep.1c

```c
#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++
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.
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.
Example task_dep.3c

```c
#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;
}
```

Example task_dep.3f

```fortran
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
```

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.

---

**Example task_dep.4c**

```c
#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;
}
```

---

**Example task_dep.4f**

```fortran
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
```
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.

6 17.5 Matrix multiplication

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

```
Example task_dep.5c

S-1 // Assume BS divides N perfectly
S-2 void matmul_depend(int N, int BS, float A[N][N], float B[N][N], float C[N][N])
S-3 {
S-4   int i, j, k, ii, jj, kk;
S-5     for (i = 0; i < N; i+=BS) {
S-6       for (j = 0; j < N; j+=BS) {
S-7         for (k = 0; k < N; k+=BS) {
S-8           // Note 1: i, j, k, A, B, C are firstprivate by default
S-9           // Note 2: A, B and C are just pointers
S-10          #pragma omp task private(ii, jj, kk) \ 
S-11            depend ( in: A[i:BS][k:BS], B[k:BS][j:BS] ) \ 
S-12                depend ( inout: C[i:BS][j:BS] )
S-13       for (ii = i; ii < i+BS; ii++)
S-14         for (jj = j; jj < j+BS; jj++)
S-15           for (kk = k; kk < k+BS; kk++)
S-17         }
S-18       }
S-19   }
S-20 }
S-21 }
```

CHAPTER 17. TASK DEPENDENCES 71
Example task_dep.5f

1

! 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) ) &
        !$omp depend ( inout: C(i:i+BM, j:j+BM) )
        do ii = i, i+BM
          do jj = j, j+BM
            do kk = k, k+BM
              C(jj,ii) = C(jj,ii) + A(kk,ii) * B(jj,kk)
            end do
          end do
        end do
      end do
    end do
  end do
  !$omp end task
end subroutine
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
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)
    {
```
```c
#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;
}
```

```fortran```
module tree_type_mod
integer, parameter :: max_steps=100
type tree_type
  type(tree_type), pointer :: left, right
dtype contains
  subroutine compute_something(tree)
    type(tree_type), pointer :: tree
  end subroutine compute_something
end type
```
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
    !$omp end single
    !$omp end parallel ! only now is background work required to be complete
    call print_results()
end program
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
#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);
        }
```

76
Example taskyield.f

```fortran
subroutine foo ( lock, n )
  use omp_lib
  integer (kind=omp_lock_kind) :: lock
  integer n
  integer i
  do i = 1, n
    !$omp task
    call something_useful()
    do while ( .not.omp_test_lock(lock) )
      !$omp taskyield
    end do
    call something_critical()
    call omp_unset_lock(lock)
    !$omp end task
  end do
end subroutine
```

CHAPTER 19. THE TASKYIELD CONSTRUCT 77
The workshare Construct

The following are examples of the workshare construct.

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

Example 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 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
   !$OMP NOWAIT
   CC = DD
   EE = FF
   !$OMP END NOWAIT
END SUBROUTINE WSHARE1
```
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 \texttt{SUM(AA)} is workshared, but the update to \texttt{R} is atomic.

\textit{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 \textit{compound statements}, made up of a control part and a statement part. When \texttt{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 \texttt{workshare} construct.

Each task gets worked on in order by the threads:

\begin{verbatim}
AA = BB then
CC = DD then
EE .ne. 0 then
\end{verbatim}
```
FF = 1 / EE then
GG = HH

Example workshare.4f

S-1 SUBROUTINE WSHARE4(AA, BB, CC, DD, EE, FF, GG, HH, N)
S-2 INTEGER N
S-3 REAL AA(N,N), BB(N,N), CC(N,N)
S-4 REAL DD(N,N), EE(N,N), FF(N,N)
S-5 REAL GG(N,N), HH(N,N)
S-6
S-7 !$OMP PARALLEL
S-8 !$OMP WORKSHARE
S-9    AA = BB
S-10   CC = DD
S-11   WHERE (EE .ne. 0) FF = 1 / EE
S-12   GG = HH
S-13 !$OMP END WORKSHARE
S-14 !$OMP END PARALLEL
S-15
S-16 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

S-1 SUBROUTINE WSHARE5(AA, BB, CC, DD, N)
S-2 INTEGER N
S-3 REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
S-4
S-5 INTEGER SHR
S-6
S-7 !$OMP PARALLEL SHARED(SHR)
S-8 !$OMP WORKSHARE
S-9    AA = BB
S-10   SHR = 1
S-11   CC = DD * SHR
S-12 !$OMP END WORKSHARE
S-13 !$OMP END PARALLEL
S-14
S-15 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.
```
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.6f

```fortran
SUBROUTINE WSHARE6_WRONG(AA, BB, CC, DD, N)
    INTEGER N
    REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)

    INTEGER PRI

    !$OMP PARALLEL PRIVATE(PRI)
    !$OMP WORKSHARE
    AA = BB
    PRI = 1
    CC = DD * PRI
    !$OMP END WORKSHARE
    !$OMP END PARALLEL

END SUBROUTINE WSHARE6_WRONG
```

Example workshare.7f

```fortran
SUBROUTINE WSHARE7(AA, BB, CC, N)
    INTEGER N
    REAL AA(N), BB(N), CC(N)

    !$OMP PARALLEL
    !$OMP WORKSHARE
    AA(1:50) = BB(11:60)
    CC(11:20) = AA(1:10)
    !$OMP END WORKSHARE
    !$OMP END PARALLEL

END SUBROUTINE WSHARE7
```
The master Construct

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

---

**C / C++**

---

```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];
        }
    }
    #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];
    }
    for( i = 1; i < n-1; ++i ){
        error = y - x[i];
    }
```

if( error > tol || error < -tol ) ++toobig;

#pragma omp master
{
    ++c;
    printf( "iteration %d, toobig=%d\n", c, toobig );
}
}while( toobig > 0 );
}

---

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

```c
void critical_example(float *x, float *y)
{
    int ix_next, iy_next;
    #pragma omp parallel shared(x, y) private(ix_next, iy_next)
    {
        #pragma omp critical (xaxis)
        ix_next = dequeue(x);
        work(ix_next, x);
        #pragma omp critical (yaxis)
        iy_next = dequeue(y);
        work(iy_next, y);
    }
}
```

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
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++

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

```fortran
SUBROUTINE CRITICAL_WORK()

    INTEGER I
    I = 1

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

END SUBROUTINE CRITICAL_WORK
```

CHAPTER 23. WORKSHARING CONSTRUCTS INSIDE A CRITICAL CONSTRUCT 87
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

```c
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)  
{  
```

88
int i;
#pragma omp parallel private(i) shared(n)
{
    #pragma omp for
    for (i=0; i<n; i++)
        sub2(i);
}
int main()
{
    sub1(2);
    sub2(2);
    sub3(2);
    return 0;
}

---

Example barrier_regions.f

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)
    !$OMP END DO
END SUBROUTINE SUB1

CHAPTER 24. BINDING OF BARRIER REGIONS 89
S-27         END DO
S-28         !$OMP END PARALLEL
S-29         END SUBROUTINE SUB1
S-30         
S-31         PROGRAM EXAMPLE
S-32         CALL SUB1(2)
S-33         CALL SUB2(2)
S-34         CALL SUB3(2)
S-35         END PROGRAM EXAMPLE
The atomic Construct

The following example avoids race conditions (simultaneous updates of an element of \( x \) by multiple threads) by using the \texttt{atomic} construct.

The advantage of using the \texttt{atomic} construct in this example is that it allows updates of two different elements of \( x \) to occur in parallel. If a \texttt{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 \texttt{atomic} directive applies only to the statement immediately following it. As a result, elements of \( y \) are not updated atomically in this example.

\begin{verbatim}
S-1 float work1(int i)
S-2 {
S-3     return 1.0 * i;
S-4 }
S-5 float work2(int i)
S-6 {
S-7     return 2.0 * i;
S-8 }
S-9 void atomic_example(float *x, float *y, int *index, int n)
S-10 {
S-11     int i;
S-12     #pragma omp parallel for shared(x, y, index, n)
S-13         for (i=0; i<n; i++) {
S-14             #pragma omp atomic update
S-15             x[index[i]] += work1(i);
S-16             y[i] += work2(i);
\end{verbatim}
int main()
{
    float x[1000];
    float y[10000];
    int index[10000];
    int i;

    for (i = 0; i < 10000; i++) {
        index[i] = i % 1000;
        y[i]=0.0;
    }

    for (i = 0; i < 1000; i++)
        x[i] = 0.0;

    atomic_example(x, y, index, 10000);

    return 0;
}

Example 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)
    $$OMP END PARALLEL DO
END SUBROUTINE SUB

OpenMP Examples Version 4.0.2 - March 2015
The following example illustrates the read and write clauses for the atomic directive. These clauses ensure that the given variable is read or written, respectively, as a whole. Otherwise, some other thread might read or write part of the variable while the current thread was reading or writing another part of the variable. Note that most hardware provides atomic reads and writes for some set of properly aligned variables of specific sizes, but not necessarily for all the variable types supported by the OpenMP API.

```
Example 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;
}

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

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.
Example atomic.3c

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

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

void do_locked_work(struct locktype *lock)
{
    int atomic_read(const int *p);
    void work();

    // Obtain the lock
    int myturn = fetch_and_add(&lock->ticketnumber);
    while (atomic_read(&lock->turn) != myturn)
    {
    // Do some work. The flush is needed to ensure visibility of
    // variables not involved in atomic directives
    #pragma omp flush
    work();
    #pragma omp flush
    // Release the lock
    fetch_and_add(&lock->turn);
    }
```
Example atomic.3f

```fortran
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 do_locked_work
end module m
```
end subroutine
end module
Restrictions on the \texttt{atomic} Construct

The following non-conforming examples illustrate the restrictions on the \texttt{atomic} construct.

\begin{verbatim}
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 */
    }
}
\end{verbatim}
Example atomic_restrict.1f

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

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

Example atomic_restrict.2c

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

    #pragma omp parallel
    {
        #pragma omp atomic update
        *i += 1;
        #pragma omp atomic update
        *r += 1.0;
        /* Incorrect because the atomic constructs reference the same location
         through incompatible types */
    }
}
```

CHAPTER 26. RESTRICTIONS ON THE ATOMIC CONSTRUCT
The following example is non-conforming because \texttt{I} and \texttt{R} reference the same location but have different types.

\textit{Example atomic\_restrict.2f}

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

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

SUBROUTINE ATOMIC_WRONG2()
    COMMON /BLK/ I
    INTEGER I

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

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

\textit{Example atomic\_restrict.3f}

```fortran
SUBROUTINE ATOMIC_WRONG3
    INTEGER:: I
    REAL:: R

    EQUIVALENCE(I,R)

    !$OMP PARALLEL
    !$OMP ATOMIC UPDATE
    I = I + 1
    !$OMP END PARALLEL

    !$OMP PARALLEL
    !$OMP ATOMIC UPDATE
    R = R + 1.0
    !$OMP END PARALLEL
```

100 OpenMP Examples Version 4.0.2 - March 2015
END SUBROUTINE ATOMIC_WRONG3
The **flush** Construct without a List

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

```c/c++
Example flush_nolist.1c
S-1  int x, *p = &x;
S-2
S-3  void f1(int *q)
S-4  {
S-5      *q = 1;
S-6      #pragma omp flush
S-7          /* x, p, and *q are flushed */
S-8          /* because they are shared and accessible */
S-9          /* q is not flushed because it is not shared. */
S-10     }
S-11
S-12  void f2(int *q)
S-13  {
S-14      #pragma omp barrier
S-15      *q = 2;
S-16      #pragma omp barrier
S-17
S-18      /* a barrier implies a flush */
S-19      /* x, p, and *q are flushed */
S-20      /* because they are shared and accessible */
S-21      /* q is not flushed because it is not shared. */
S-22     }
S-23
S-24  int g(int n)
```
```c
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;
}
```

**Example flush_nolist.1f**

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

   Q = 1

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

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

INTEGER Q

!$OMP BARRIER
Q = 2
!$OMP BARRIER

! a barrier implies a flush
! X, P and Q are flushed
! because they are shared and accessible
END SUBROUTINE F2

INTEGER FUNCTION G(N)

COMMON /DATA/ X, P

INTEGER, TARGET :: X
INTEGER, POINTER :: P

INTEGER N

INTEGER I, J, SUM

I = 1
SUM = 0
P = 1

!$OMP PARALLEL REDUCTION(+: SUM) NUM_THREADS(10)
CALL F1(J)

! I, N and SUM were not flushed
! because they were not accessible in F1
! J was flushed because it was accessible
SUM = SUM + J

CALL F2(J)

! I, N, and SUM were not flushed
! because they were not accessible in f2
! J was flushed because it was accessible
SUM = SUM + I + J + P + N

!$OMP END PARALLEL

G = SUM

END FUNCTION G

PROGRAM FLUSH_NOLIST

COMMON /DATA/ X, P

INTEGER, TARGET :: X
INTEGER, POINTER :: P

INTEGER RESULT, G

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

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

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

Example standalone.1f

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

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

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

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

    GOTO 100

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

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

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

END SUBROUTINE

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

Example standalone.2c

```c
void standalone_ok()
{
    int a = 1;
    
    #pragma omp parallel
    {
        if (a != 0) {
            #pragma omp flush(a)
            
            if (a != 0) {
                #pragma omp barrier
                
                if (a != 0) {
                    #pragma omp taskwait
                    
                    if (a != 0) {
                        #pragma omp taskyield
                        
                        if (a != 0) {
                            #pragma omp taskyield
                            
                            if (a != 0) {
                                #pragma omp taskyield
                                
                            }
                        }
                    }
                }
            }
        }
    }
}
```

The following example is conforming because the `flush`, `barrier`, `taskwait`, and `taskyield` directives are enclosed in an `if` construct or follow the labeled branch target.
Example 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
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
#include <stdio.h>

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

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

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

int main()
{
    ordered_example(0, 100, 5);
    return 0;
}
```
Example ordered.1

```fortran
SUBROUTINE WORK(K)
    INTEGER k

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

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

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

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

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

Example ordered.2

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

void ordered_wrong(int n) 
{
    int i;
    #pragma omp for ordered
    for (i=0; i<n; i++) {
        /* incorrect because an iteration may not execute more than one
           ordered region */
        #pragma omp ordered
        work(i);
    }
```
The following is a conforming example with more than one `ordered` construct. Each iteration will execute only one `ordered` region:
Example ordered.3c

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

Example ordered.3f

```fortran
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
```
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++
#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) {
```
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.

Example cancellation.1f

```fortran
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
    endif
    !$omp cancel do
    endif
    ! ...
```
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 undetected tasks after the first ten levels of the tree.

```c
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) {
          
```
The following is the equivalent parallel search example in Fortran.

```fortran
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 = search_tree(tree, value, 0);
      end if
    end if
  return found;
end subroutine search_tree
end module parallel_search
```

Example cancellation.2f

```fortran
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 = search_tree(tree, value, 0);
      end if
    end if
  return found;
end subroutine search_tree
end module parallel_search
```
S-17    found => tree
S-18    else
S-19    !$omp task shared(found) if(level<10)
S-20    call search_tree(tree%left, value, level+1, found_left)
S-21    if (associated(found_left)) then
S-22    !$omp critical
S-23    found => found_left
S-24    !$omp end critical
S-25
S-26    !$omp cancel taskgroup
S-27    endif
S-28    !$omp end task
S-29
S-30    !$omp task shared(found) if(level<10)
S-31    call search_tree(tree%right, value, level+1, found_right)
S-32    if (associated(found_right)) then
S-33    !$omp critical
S-34    found => found_right
S-35    !$omp end critical
S-36
S-37    !$omp cancel taskgroup
S-38    endif
S-39    !$omp end task
S-40
S-41    !$omp taskwait
S-42    endif
S-43    endif
S-44    end subroutine
S-45
S-46    subroutine search_tree_parallel(tree, value, found)
S-47    type(binary_tree), intent(in), pointer :: tree
S-48    integer, intent(in) :: value
S-49    type(binary_tree), pointer :: found
S-50
S-51    found => NULL()
S-52    !$omp parallel shared(found, tree, value)
S-53    !$omp master
S-54    !$omp taskgroup
S-55    call search_tree(tree, value, 0, found)
S-56    !$omp end taskgroup
S-57    !$omp end master
S-58    !$omp end parallel
S-59    end subroutine
S-60
S-61    end module parallel_search

Fortran
The threadprivate Directive

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

C / C++

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

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

Fortran

```
INTEGER FUNCTION INCREMENT_COUNTER()
COMMON/INC_COMMON/COUNTER
!$OMP THREADPRIVATE(/INC_COMMON/)
    COUNTER = COUNTER +1
    INCREMENT_COUNTER = COUNTER
    RETURN
END FUNCTION INCREMENT_COUNTER
```
The following example uses `threadprivate` on a static variable:

```
Example 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. */
}

The following examples show non-conforming uses and correct uses of the `threadprivate` directive.

```fortran
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.2f

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

Example \texttt{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_SUB
END SUBROUTINE INC_GOOD
```

The following is an example of the use of \texttt{threadprivate} for local variables:

Example \texttt{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
```

OpenMP Examples Version 4.0.2 - March 2015
The above program, if executed by two threads, will print one of the following two sets of output:

```
1  a = 11 12 13
2  ptr = 4
3  i = 15
4  A is not allocated
5  ptr = 4
6  i = 5
7  or
8
9  A is not allocated
10  ptr = 4
11  i = 15
12
13  a = 1 2 3
14  ptr = 4
15  i = 5
16```

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

```
Example threadprivate.6f
```

```
S-1    MODULE INC_MODULE_GOOD3
S-2       REAL, POINTER :: WORK(:)
S-3       SAVE WORK
S-4   !$OMP THREADPRIVATE(WORK)
S-5     END MODULE INC_MODULE_GOOD3
S-6
S-7     SUBROUTINE SUB1(N)
S-8       USE INC_MODULE_GOOD3
S-9   !$OMP PARALLEL PRIVATE(THE_SUM)
S-10     ALLOCATE(WORK(N))
S-11    CALL SUB2(THE_SUM)
```
WRITE(*,*) THE_SUM

!$OMP END PARALLEL

END SUBROUTINE SUB1

SUBROUTINE SUB2(THE_SUM)

USE INC_MODULE_GOOG3

WORK(:) = 10
THE_SUM = SUM(WORK)

END SUBROUTINE SUB2

PROGRAM INC_GOOG3

N = 10
CALL SUB1(N)

END PROGRAM INC_GOOG3

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)
};
Parallel Random Access Iterator Loop

The following example shows a parallel random access iterator loop.

Example pra_iterator.1c

```cpp
#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 //
    }
}```
CHAPTER 33

Fortran Restrictions on shared and private Clauses with Common Blocks

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

The following example is conforming:

Example 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_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
```
SUBROUTINE COMMON_GOOD2()
    COMMON /C/ X,Y
    REAL X, Y
    INTEGER I
    !$OMP PARALLEL
    !$OMP DO PRIVATE(/C/)
    DO I=1,1000
        ! do work here
    ENDDO
    !$OMP END DO
    !$OMP DO PRIVATE(X)
    DO I=1,1000
        ! do work here
    ENDDO
    !$OMP END PARALLEL
END SUBROUTINE COMMON_GOOD2

The following example is conforming:

Example 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
    !$OMP PARALLEL PRIVATE(/C/), SHARED(X)
    ! do work here
    !$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

S-1 SUBROUTINE COMMON_WRONG2()
S-2 COMMON /C/ X, Y
S-3 ! Incorrect: common block C cannot be declared both
S-4 ! shared and private
S-5 !$OMP PARALLEL PRIVATE (/C/), SHARED(/C/)
S-6 ! do work here
S-7 !$OMP END PARALLEL
S-8
S-9 END SUBROUTINE COMMON_WRONG2
The following example distinguishes the variables that are affected by the `default(none)` clause from those that are not.

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

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

    #pragma omp parallel default(none) private(a) shared(z, 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++) {
```
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
CHAPTER 35

Race Conditions Caused by Implied Copies of Shared Variables in Fortran

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

Example 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
The **private** Clause

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

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

int main()
{
    int i, j;
    int *ptr_i, *ptr_j;
    i = 1;
    j = 2;
    ptr_i = &i;
    ptr_j = &j;
    #pragma omp parallel private(i) firstprivate(j)
    {
        i = 3;
        j = j + 2;
        assert (*ptr_i == 1 && *ptr_j == 2);
    }
    assert(i == 1 && j == 2);
    return 0;
}
```
Example 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
```

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

Example 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" */
    }
}
```
Fortran Example private.2f

1

S-1 MODULE PRIV_EXAMPLE2
S-2 REAL A
S-3
S-4 CONTAINS
S-5 S-6 SUBROUTINE G(K)
S-7 REAL K
S-8 A = K ! Accessed in the region but outside of the
S-9 ! construct; therefore unspecified whether
S-10 ! original or private list item is modified.
S-11 END SUBROUTINE G
S-12 S-13 SUBROUTINE F(N)
S-14 INTEGER N
S-15 REAL A
S-16 S-17 INTEGER I
S-18 !$OMP PARALLEL DO PRIVATE(A)
S-19 DO I = 1,N
S-20 A = I
S-21 CALL G(A*2)
S-22 ENDDO
S-23 !$OMP END PARALLEL DO
S-24 END SUBROUTINE F
S-25 S-26 END MODULE PRIV_EXAMPLE2

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

C / C++ Example private.3c

2

S-1 #include <assert.h>
S-2 void priv_example3()
S-3 {
S-4 int i, a;
S-5 S-6 #pragma omp parallel private(a)
S-7 {
S-8 a = 1;
S-9 #pragma omp parallel for private(a)
for (i=0; i<10; i++)
{
    a = 2;
}
assert(a == 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 Restrictions on Storage Association with the \texttt{private} Clause

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

\textit{Example fort\_sa\_private.1f}

\begin{verbatim}
S-1 SUBROUTINE SUB()
S-2 COMMON /BLOCK/ X
S-3 PRINT *,X ! X is undefined
S-4 END SUBROUTINE SUB
S-5 PROGRAM PRIV_RESTRICT
S-6 COMMON /BLOCK/ X
S-7 X = 1.0
S-8 !$OMP PARALLEL PRIVATE (X)
S-9 X = 2.0
S-10 CALL SUB()
S-11 !$OMP END PARALLEL
S-12 END PROGRAM PRIV_RESTRICT
\end{verbatim}

\textit{Example fort\_sa\_private.2f}

\begin{verbatim}
S-1 PROGRAM PRIV_RESTRICT2
S-2 COMMON /BLOCK2/ X
S-3 X = 1.0
S-4
S-5 !$OMP PARALLEL PRIVATE (X)
S-6 X = 2.0
S-7 CALL SUB()
S-8 !$OMP END PARALLEL
\end{verbatim}
CONTAINS

SUBROUTINE SUB()
COMMON /BLOCK2/ Y

PRINT *,X ! X is undefined
PRINT *,Y ! Y is undefined
END SUBROUTINE SUB

END PROGRAM PRIV_RESTRICT2

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

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
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(J) = A(J) + 1 ! A becomes undefined at this point
ENDDO
$S-20$ ENDDO
$S-21$ !$OMP END PARALLEL DO ! The LASTPRIVATE write for A has
$S-22$ ! undefined results
$S-23$
$S-24$ PRINT *, B ! B is undefined since the LASTPRIVATE
$S-25$ ! write of A was not defined
$S-26$ END PROGRAM PRIV_RESTRICT4

Example fort_sa_private.5f

S-1 SUBROUTINE SUB1(X)
S-2 DIMENSION X(10)
S-3
S-4 ! This use of X does not conform to the
S-5 ! specification. It would be legal Fortran 90,
S-6 ! but the OpenMP private directive allows the
S-7 ! compiler to break the sequence association that
S-8 ! A had with the rest of the common block.
S-9
S-10 FORALL (I = 1:10) X(I) = I
S-11 END SUBROUTINE SUB1
S-12
S-13 PROGRAM PRIV_RESTRICT5
S-14 COMMON /BLOCK5/ A
S-15
S-16 DIMENSION B(10)
S-17 EQUIVALENCE (A,B(1))
S-18
S-19 ! the common block has to be at least 10 words
S-20 A = 0
S-21
S-22 !$OMP PARALLEL PRIVATE(/BLOCK5/)
S-23
S-24 ! Without the private clause,
S-25 ! we would be passing a member of a sequence
S-26 ! that is at least ten elements long.
S-27 ! With the private clause, A may no longer be
S-28 ! sequence-associated.
S-29
S-30 CALL SUB1(A)
S-31 !$OMP MASTER
S-32 PRINT *, A
S-33 !$OMP END MASTER
S-34
S-35 !$OMP END PARALLEL
S-36 END PROGRAM PRIV_RESTRICT5
C/C++ Arrays in a firstprivate Clause

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

In this example:

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

Note that B and E involve variable length array types.

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

Example carrays_fpriv.1c

```c
#include <assert.h>
int A[2][2] = {1, 2, 3, 4};
void f(int n, int B[n][n], int C[])
{
    int D[2][2] = {1, 2, 3, 4};
    int E[n][n];
```
assert(n >= 2);
E[1][1] = 4;

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

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

int main() {
    f(2, A, A[0]);
    return 0;
}
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 \texttt{lastprivate} clause so that the values of the variables are the same as when the loop is executed sequentially.

\begin{Verbatim}
\begin{Verbatim}
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 */
}
\end{Verbatim}
\end{Verbatim}
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
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++
#include <math.h>

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

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);
}
}

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)
!$OMP PARALLEL SHARED(X, Y, A, B, C, D, N) &
A_P = 0.0
B_P = 0
C_P = HUGE(C_P)
D_P = -HUGE(D_P)
!$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)
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
```

```
S-1  PROGRAM REDUCTION_WRONG
S-2   MAX = HUGE(0)
S-3      M = 0
S-4
S-5      !$OMP PARALLEL DO REDUCTION(MAX: M)
S-6 ! MAX is no longer the intrinsic so this is non-conforming
S-7     DO I = 1, 100
S-8       CALL SUB(M,I)
S-9     END DO
S-10
S-11  END PROGRAM REDUCTION_WRONG
S-12
S-13  SUBROUTINE SUB(M,I)
S-14      M = MAX(M,I)
S-15  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
```

```
S-1  MODULE M
S-2     INTRINSIC MAX
S-3  END MODULE M
S-4
S-5  PROGRAM REDUCTION3
S-6     USE M, REN => MAX
S-7     N = 0
S-8      !$OMP PARALLEL DO REDUCTION(REN: N) ! still does MAX
S-9      DO I = 1, 100
S-10     N = MAX(N,I)
S-11    END DO
S-12  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
```
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.
S-19      #pragma omp single
S-20      printf ("Sum is %d\n", a);
S-21    }
S-22      return 0;
S-23    }

Example reduction.6f

S-1      INTEGER A, I
S-2
S-3      !$OMP PARALLEL SHARED(A) PRIVATE(I)
S-4
S-5      !$OMP MASTER
S-6      A = 0
S-7      !$OMP END MASTER
S-8
S-9      ! To avoid race conditions, add a barrier here.
S-10
S-11      !$OMP DO REDUCTION(+:A)
S-12      DO I= 0, 9
S-13      A = A + I
S-14      END DO
S-15
S-16      !$OMP SINGLE
S-17      PRINT *, "Sum is ", A
S-18      !$OMP END SINGLE
S-19
S-20      !$OMP END PARALLEL
S-21      END
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.

```
#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();
    }
}
```
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
```
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.

```
#include <stdio.h>

float x, y;
#pragma omp threadprivate(x, y)

void init(float a, float b) {
    #pragma omp single copyprivate(a, b, x, y)
    {
        scanf("%f %f %f %f", &a, &b, &x, &y);
    }
}
```

Example copyprivate.1f

```fortran
SUBROUTINE INIT(A,B)
    REAL A, B
    COMMON /XY/ X,Y
    !$OMP THREADPRIVATE (/XY/)
    !$OMP SINGLE
    READ (11) A,B,X,Y
    !$OMP END SINGLE COPYPRIVATE (A,B,/XY/)
END SUBROUTINE INIT
```

In this example, assume that the input must be performed by the master thread. Since the `master` construct does not support the `copyprivate` clause, it cannot broadcast the input value that is read. However, `copyprivate` is used to broadcast an address where the input value is stored.

Example copyprivate.2c

```c
#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);
    }
```
Example copyprivate.2f

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

!$OMP SINGLE
ALLOCATE (TMP)

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

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

!$OMP BARRIER
READ_NEXT = TMP
!$OMP BARRIER

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

END FUNCTION READ_NEXT
```

Suppose that the number of lock variables required within a parallel region cannot easily be determined prior to entering it. The copyprivate clause can be used to provide access to shared lock variables that are allocated within that parallel region.
Example copyprivate.3c

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>
omp_lock_t *new_lock()
{
  omp_lock_t *lock_ptr;
  #pragma omp single copyprivate(lock_ptr)
  {
    lock_ptr = (omp_lock_t *) malloc(sizeof(omp_lock_t));
    omp_init_lock(lock_ptr);
  }
  return lock_ptr;
}
```

Example copyprivate.3f

```fortran
FUNCTION NEW_LOCK()
  USE OMP_LIB ! or INCLUDE "omp_lib.h"
  INTEGER(OMP_LOCK_KIND), POINTER :: NEW_LOCK
  !$OMP SINGLE
  ALLOCATE(NEW_LOCK)
  CALL OMP_INIT_LOCK(NEW_LOCK)
  !$OMP END SINGLE COPYPRIVATE(NEW_LOCK)
END FUNCTION NEW_LOCK
```

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

Example copyprivate.4f
SUBROUTINE S(N)
 INTEGER N
 REAL, DIMENSION(:), ALLOCATABLE :: A
 REAL, DIMENSION(:), POINTER :: B

ALLOCATE (A(N))
!
 ALLOCATE (B(N))
 READ (11) A,B
!
 END SINGLE COPYPRIVATE(A,B)
!
 Variable A is private and is
 ! assigned the same value in each thread
 ! Variable B is shared
!
 BARRIER

 DEALLOCATE (B)
!
 END SINGLE NOWAIT

 END SUBROUTINE S
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
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);
            }
        }
    }
}
```

Fortran

Example nested_loop.1f

1

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

The following variation of the preceding example is also conforming:

C / C++

Example nested_loop.2c

2

void work(int i, int j) {}

3

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)
{

```c
int i;

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

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

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

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

Example nested_loop.2f

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

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

SUBROUTINE GOOD_NESTING2(N)
INTEGER N
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
    DO I = 1, N
        CALL WORK1(I, N)
    END DO
!$OMP END PARALLEL
END SUBROUTINE GOOD_NESTING2
```
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:

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

Example nesting_restrict.1c
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
```

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

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

```
void wrong3(int n)
{
  #pragma omp parallel default(shared)
  {
    #pragma omp for
    for (i=0; i<n; i++) {
      /* incorrect nesting of regions */
      #pragma omp single
      work(i, 0);
    }
  }
}
```
The following example is non-conforming because a `barrier` region cannot be closely nested inside a loop region:
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:
The following example is non-conforming because the barrier region cannot be closely nested inside the single region. If this were permitted, it would result in deadlock due to the fact that only one thread executes the single region:
Example nesting_restrict.6f

S-1 SUBROUTINE WRONG6(N)
S-2 INTEGER N
S-3
S-4 !$OMP PARALLEL DEFAULT(SHARED)
S-5 !$OMP SINGLE
S-6 CALL WORK(N,1)
S-7 ! incorrect nesting of barrier region in a single region
S-8 !$OMP BARRIER
S-9 CALL WORK(N,2)
S-10 !$OMP END SINGLE
S-11 !$OMP END PARALLEL
S-12 END SUBROUTINE WRONG6
The `omp_set_dynamic` and `omp_set_num_threads` Routines

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

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

```c
#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)
```

---

Example set_dynamic_nthrs.1c
if (omp_get_num_threads() != 16)
    abort();

iam = omp_get_thread_num();
ipoints = npoints/16;
do_by_16(x, iam, ipoints);

Example set_dynamic_nthrs.f

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

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

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

```c
#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++
#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);
}
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
```

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++
Example get_nthrs.2f

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

    SUBROUTINE CORRECT()
        INCLUDE "omp_lib.h" ! or USE OMP_LIB
        INTEGER I
        !$OMP PARALLEL PRIVATE(I)
        I = OMP_GET_THREAD_NUM()
        CALL WORK(I)
        !$OMP END PARALLEL
    END SUBROUTINE CORRECT
```

CHAPTER 46. THE OMP\_GET\_NUM\_THREADS ROUTINE
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
#include <omp.h>

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

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

S-1 FUNCTION NEW_LOCKS()
S-2 USE OMP_LIB ! or INCLUDE "omp_lib.h"
S-3 INTEGER(OMP_LOCK_KIND), DIMENSION(1000) :: NEW_LOCKS
S-4
S-5 INTEGER I
S-6
S-7 !$OMP PARALLEL DO PRIVATE(I)
S-8 DO I=1,1000
S-9 CALL OMP_INIT_LOCK(NEW_LOCKS(I))
S-10 END DO
S-11 !$OMP END PARALLEL DO
S-12
S-13 END FUNCTION NEW_LOCKS
Ownership of Locks

Ownership of locks has changed since OpenMP 2.5. In OpenMP 2.5, locks are owned by threads; so a lock released by the `omp_unset_lock` routine must be owned by the same thread executing the routine. 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
#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
        ...
    }
    ...
```

The changes in OpenMP 3.0 require that locks be released in the same task region as they were acquired. The example program shows how this can be done using the `omp_unset_lock` routine.
Example lock_owner.f

1

S-1 program lock
S-2 use omp_lib
S-3 integer :: x
S-4 integer (kind=omp_lock_kind) :: lck
S-5
S-6 call omp_init_lock (lck)
S-7 call omp_set_lock(lck)
S-8 x = 0
S-9
S-10 !$omp parallel shared (x)
S-11 !$omp master
S-12 x = x + 1
S-13 call omp_unset_lock(lck)
S-14 !$omp end master
S-15
S-16 ! Some more stuff.
S-17 !$omp end parallel
S-18
S-19 call omp_destroy_lock(lck)
S-20 end
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.

Example simple_lock.1c

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

void skip(int i) {} 
void work(int i) {} 

int main()
{
    omp_lock_t lck; 
    int id; 
    omp_init_lock(&lck);

    #pragma omp parallel shared(lck) private(id)
    {
        id = omp_get_thread_num();
        omp_set_lock(&lck);
        /* only one thread at a time can execute this printf */
        printf("My thread id is %d.\n", id);
        omp_unset_lock(&lck);

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

work(id); /* we now have the lock and can do the work */

omp_unset_lock(&lck);
}
omp_destroy_lock(&lck);
return 0;

C / C++

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

Fortran

Example simple_lock.f

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
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
#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)
{
    /* Called both from incr_pair and elsewhere, */
    /* so need a nestable lock. */
    omp_set_nest_lock(&p->lck);
    incr_a(p, a);
    incr_b(p, b);
    omp_unset_nest_lock(&p->lck);
}
```
Example nestable_lock.f

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

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

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

SUBROUTINE NESTLOCK(P)
USE OMP_LIB ! or INCLUDE "omp_lib.h"
USE DATA
TYPE(LOCKED_PAIR) :: P
INTEGER WORK1, WORK2, WORK3
EXTERNAL WORK1, WORK2, WORK3
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
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.

**Example SIMD.1c**

```c
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];
}
```

**Example SIMD.1f**

```fortran
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
```
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 `uniform` 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 each vector operation has its own `tmp` variable.

```c
#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-1;
        b[i] = N-(i-1);
    }
    work(a, b, N );
    for ( i=0; i<N; i++ ) {
        printf("%d %f\n", i, a[i]);
    }
    return 0;
}

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)

Example SIMD.2f
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.
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;
}

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

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.
Example SIMD.4c

```c
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;
}
```

Example SIMD.4f

```fortran
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
```

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

Example SIMD.5c

```c
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;
        }
    }
}
```
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
```

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`.

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){
```
```c
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];
}
```

Example SIMD.6f

```c
function foo(p) result(r)
!
integer :: p, r
p = p + 10
r = p
end function foo

function myaddint(int *a, int *b, int n) result(r)
!
integer :: a(*), b(*), n, r
integer :: i
integer, external :: foo
!
#pragma 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)
!
real :: p, r
p = p + 18.5
r = p
end function goo
```
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

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.

Example SIMD.7c

#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)
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
    ...
\begin{verbatim}
S-30    r = fib(n-1) + fib(n-2)
S-31      endif
S-32
S-33    end function fib
\end{verbatim}
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
S-1 extern void init(float*, float*, int);
S-2 extern void output(float*, int);
S-3 void vec_mult(int N)
S-4 {
S-5     int i;
S-6     float p[N], v1[N], v2[N];
S-7     init(v1, v2, N);
S-8     #pragma omp target
S-9     #pragma omp parallel for private(i)
S-10    for (i=0; i<N; i++)
S-11    p[i] = v1[i] * v2[i];
S-12    output(p, N);
S-13 }
```
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++
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);
}
```
The following example shows how the \texttt{target} construct offloads a code region to a target device. In the \texttt{map} clause, the \texttt{to} and \texttt{from} map-types define the mapping between the original (host) data and the target (device) data. The \texttt{to} map-type specifies that the data will only be read on the device, and the \texttt{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 \texttt{target} region.

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

The \texttt{from} map-type indicates that at the start of the \texttt{target} region the variable \texttt{p} is not initialized with the value of the corresponding variable on the host device, and at the end of the \texttt{target} region the variable \texttt{p} is assigned to the corresponding variable on the host device.
Example target.3c

```c
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);
}
```

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.

Example target.3f

```fortran
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
```

OpenMP Examples Version 4.0.2 - March 2015
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

```c
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);
}
```

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

```fortran
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
```

CHAPTER 52. TARGET CONSTRUCT 197
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).

```
Example target.4bf
```

```fortran
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
```

### 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.
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])
    #pragma omp parallel for if(N>THRESHOLD2)
    for (i=0; i<N; i++)
    p[i] = v1[i] * v2[i];
    output(p, N);
}
```

Example target.5f

```
module params
integer,parameter :: THRESHOLD1=1000000, THRESHOLD2=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)
    !$omp target if(N>THRESHOLD1) map(to: v1, v2 ) map(from: p)
    !$omp parallel do if(N>THRESHOLD2)
    do i=1,N
        p(i) = v1(i) * v2(i)
    end do
    !$omp end target
end subroutine
```
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.

```
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);
}
```
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.

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
```

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

```c
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);
}
```

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

```fortran
subroutine vec_mult(p, v1, v2, N)
    real :: p(N), v1(N), v2(N)
    integer :: i
    call init(v1, v2, N)
    !$omp target data map(from: p)
    !$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)
```

202 OpenMP Examples Version 4.0.2 - March 2015
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.

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.
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 \((v2[:N], v4[:N])\).

---

**Example target_data.4c**

```c

S-1 void vec_mult(float*, float*, float*, int);
S-2 extern void init(float*, float*, int);
S-3 extern void output(float*, int);
S-4 void foo(float *p0, float *v1, float *v2, int N)
S-5 {
S-6   init(v1, v2, N);
S-7   #pragma omp target data map(to: v1[0:N], v2[:N]) map(from: p0[0:N])
S-8   {
S-9     vec_mult(p0, v1, v2, N);
S-10 }
S-11 output(p0, N);
S-12 }
S-13 void vec_mult(float *p1, float *v3, float *v4, int N)
S-14 {
S-15   int i;
S-16   #pragma omp target map(to: v3[0:N], v4[:N]) map(from: p1[0:N])
S-17   #pragma omp parallel for
S-18   for (i=0; i<N; i++)
S-19     p1[i] = v3[i] * v4[i];
S-20 }
```

---

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 \((v2,v4)\).
In the following example, the variables \( p_1 \), \( v_3 \), and \( v_4 \) are references to the pointer variables \( p_0 \), \( v_1 \) and \( v_2 \) respectively. The \texttt{target} construct’s device data environment inherits the pointer variables \( p_0 \), \( v_1 \), and \( v_2 \) from the enclosing \texttt{target data} construct’s device data environment. Thus, \( p_1 \), \( v_3 \), and \( v_4 \) are already present in the device data environment.
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.

### Example `target_data.5f`

```fortran
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) )
```

CHAPTER 53. TARGET DATA CONSTRUCT
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.

Example target_data.6c

```c
#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 data 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);
}
```
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.

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].
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`. 
!$omp end target
!$omp end target data
call output(p, N) !*** UNDEFINED behavior if N<=THRESHOLD
end subroutine
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 \texttt{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 \( v2[:N] \).
Example target_update.1c

```c
#include <stdio.h>

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

Example target_update.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 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
    !$omp parallel do
    do i=1,N
        p(i) = p(i) + (v1(i) * v2(i))
    end do
end subroutine vec_mult
```
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.
Example target_update.2f

```fortran
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
```

CHAPTER 54. **TARGET UPDATE CONSTRUCT** 215
**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
Example declare_target.1c
```

```c
#pragma omp declare target
eextern 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);
    }
}
```
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
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
```

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.
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
```

55.2 declare target Construct for Class Type

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

```c++
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
```
55.3 declare target and end declare target for Variables

The following examples show how the \texttt{declare target} and \texttt{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 \texttt{declare target} and \texttt{end declare target} directives indicating that the variables are mapped to the implicit device data environment of each target device. The \texttt{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.
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.

**Example declare_target.3f**

```fortran
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
```

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

```c
#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

```fortran
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
```

CHAPTER 55. DECLARE TARGET CONSTRUCT 221
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
#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;
}
```
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.

```
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
end do
end function
```
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

```c
#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;

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
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++ ---
```
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.
Example teams.3c

```c
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;
}
```

Example teams.3f

```fortran
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
```
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.

---

Example teams.4c

```c
#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;
}
```
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

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.
Example teams.5c

S-1 extern void init(float *, float *, int);
S-2 extern void output(float *, int);
S-3 void vec_mult(float *p, float *v1, float *v2, int N)
S-4 {
S-5     int i;
S-6     init(v1, v2, N);
S-7     #pragma omp target teams map(to: v1[0:N], v2[:N]) map(from: p[0:N])
S-8     #pragma omp distribute simd
S-9     for (i=0; i<N; i++)
S-10    p[i] = v1[i] * v2[i];
S-11    output(p, N);
S-12 }

Example teams.5f

S-1 subroutine vec_mult(p, v1, v2, N)
S-2     real :: p(N), v1(N), v2(N)
S-3     integer :: i
S-4     call init(v1, v2, N)
S-5     !$omp target teams map(to: v1, v2) map(from: p)
S-6     !$omp distribute simd
S-7     do i=1,N
S-8     p(i) = v1(i) * v2(i)
S-9     end do
S-10    !$omp end target teams
S-11    call output(p, N)
S-12 end subroutine
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.

Example teams.6c

```c
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);
}
```

Example teams.6f

```fortran
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```

OpenMP Examples Version 4.0.2 - March 2015
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

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

---

Example async_target.f

```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
```

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.
Example async_target.2c

```c
#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 asynchronously
    #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);
}
```

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

```fortran
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
```

---

236 OpenMP Examples Version 4.0.2 - March 2015
p(i) = v1(i) \times v2(i)
end do
dealloc(v1,v2)

!$omp end target
!$omp end task

!$omp taskwait
call output(p, N)

end subroutine
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 separate sections of the same array inside of a `target` construct.

```c
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;
        }
    }
}
```

Example array_sections.1f

```fortran
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
```

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

Example array_sections.2c

```c
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;
}
}
```

CHAPTER 58. ARRAY SECTIONS IN DEVICE CONSTRUCTS 239
Fortran

Example array_sections.2f

```fortran
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
```

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

```c
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;
    }
  }
}
```
This example shows the valid usage of a wholly contained array section of an already mapped array section inside of a `target` construct.
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
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
#include <stdio.h>
#include <omp.h>
#pragma omp declare target
void vec_mult(float *, float *, float *, int);
extern float *, float *, float *, int;
#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 *, float *, float *, int)
{
    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];
The following example shows how the `omp_get_num_devices` runtime library routine can be used to determine the number of devices.

```c
#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);
}
```
Example device.2f

S-1 subroutine vec_mult(p, v1, v2, N)
S-2 use omp_lib, ONLY : omp_get_num_devices
S-3 real :: p(N), v1(N), v2(N)
S-4 integer :: N, i, ndev
S-5 logical :: do_offload
S-6 call init(v1, v2, N)
S-7 ndev = omp_get_num_devices()
S-8 do_offload = (ndev>0) .and. (N>1000000)
S-9 !$omp target if(do_offload) map(to: v1, v2) map(from: p)
S-10 !$omp parallel do if(N>1000)
S-11 do i=1,N
S-12 p(i) = v1(i) * v2(i)
S-13 end do
S-14 !$omp end target
S-15 call output(p, N)
S-16 end subroutine

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.

Example device.3c

S-1 #include <omp.h>
S-2 #include <stdio.h>
S-3 void foo(void)
S-4 {
S-5 int default_device = omp_get_default_device();
S-6 printf("Default device = %d\n", default_device);
S-7 omp_set_default_device(default_device+1);
S-8 if (omp_get_default_device() != default_device+1)
S-9 printf("Default device is still = %d\n", default_device);
S-10 }
Example device.3f

S-1  program foo
S-2  use omp_lib, ONLY : omp_get_default_device, omp_set_default_device
S-3  integer :: old_default_device, new_default_device
S-4    old_default_device = omp_get_default_device()
S-5    print*, "Default device = ", old_default_device
S-6    new_default_device = old_default_device + 1
S-7    call omp_set_default_device(new_default_device)
S-8    if (omp_get_default_device() == old_default_device) &
S-9      print*, "Default device is STILL = ", old_default_device
S-10  end program
Fortran ASSOCIATE Construct

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

```
S-1      program example
S-2      real :: a, c
S-3      associate (b => a)
S-4      !$omp parallel private(b, c)        ! invalid to privatize b
S-5      c = 2.0*b
S-6      !$omp end parallel
S-7      end associate
S-8      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

```
S-1      program example
S-2      use omp_lib
S-3      integer  i
S-4      !$omp parallel private(i)
S-5      i = omp_get_thread_num()
S-6      associate(thread_id => i)
S-7      print *, thread_id            ! print private i value
S-8      end associate
S-9      !$omp end parallel
S-10     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

```fortran
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
```

---

CHAPTER 60. FORTRAN ASSOCIATE CONSTRUCT
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)