



OpenMP Application Programming Interface

Examples

Version 4.0.2 – March 2015

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 <code>parallel</code> Construct	14
6 Controlling the Number of Threads on Multiple Nesting Levels	17
7 Interaction Between the <code>num_threads</code> Clause and <code>omp_set_dynamic</code>	20
8 The <code>proc_bind</code> 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 <code>do</code> Construct	29
10 Fortran Private Loop Iteration Variables	31
11 The <code>nowait</code> Clause	33
12 The <code>collapse</code> Clause	37
13 The <code>parallel sections</code> Construct	41
14 The <code>firstprivate</code> Clause and the <code>sections</code> Construct	43
15 The <code>single</code> Construct	45

16 The task and taskwait Constructs	47
17 Task Dependencies	66
17.1 Flow Dependence	66
17.2 Anti-dependence	67
17.3 Output Dependence	68
17.4 Concurrent Execution with Dependences	70
17.5 Matrix multiplication	71
18 The taskgroup Construct	73
19 The taskyield Construct	76
20 The workshare Construct	78
21 The master Construct	82
22 The critical Construct	84
23 Worksharing Constructs Inside a critical Construct	86
24 Binding of barrier Regions	88
25 The atomic Construct	91
26 Restrictions on the atomic Construct	98
27 The flush Construct without a List	102
28 Placement of flush, barrier, taskwait and taskyield Directives	106
29 The ordered Clause and the ordered Construct	110
30 Cancellation Constructs	114
31 The threadprivate Directive	119
32 Parallel Random Access Iterator Loop	125
33 Fortran Restrictions on shared and private Clauses with Common Blocks	126

34 The <code>default (none)</code> Clause	129
35 Race Conditions Caused by Implied Copies of Shared Variables in Fortran	131
36 The <code>private</code> Clause	133
37 Fortran Restrictions on Storage Association with the <code>private</code> Clause	137
38 C/C++ Arrays in a <code>firstprivate</code> Clause	140
39 The <code>lastprivate</code> Clause	142
40 The <code>reduction</code> Clause	144
41 The <code>copyin</code> Clause	150
42 The <code>copyprivate</code> Clause	152
43 Nested Loop Constructs	157
44 Restrictions on Nesting of Regions	160
45 The <code>omp_set_dynamic</code> and <code>omp_set_num_threads</code> Routines	167
46 The <code>omp_get_num_threads</code> Routine	169
47 The <code>omp_init_lock</code> Routine	172
48 Ownership of Locks	174
49 Simple Lock Routines	176
50 Nestable Lock Routines	179
51 SIMD Constructs	182
52 <code>target</code> Construct	193
52.1 <code>target</code> Construct on <code>parallel</code> Construct	193
52.2 <code>target</code> Construct with <code>map</code> Clause	194

52.3 <code>map</code> Clause with <code>to/from</code> map-types	195
52.4 <code>map</code> Clause with Array Sections	197
52.5 <code>target</code> Construct with <code>if</code> Clause	198
53 <code>target data</code> Construct	200
53.1 Simple <code>target data</code> Construct	200
53.2 <code>target data</code> Region Enclosing Multiple <code>target</code> Regions	201
53.3 <code>target data</code> Construct with Orphaned Call	204
53.4 <code>target data</code> Construct with <code>if</code> Clause	208
54 <code>target update</code> Construct	212
54.1 Simple <code>target data</code> and <code>target update</code> Constructs	212
54.2 <code>target update</code> Construct with <code>if</code> Clause	214
55 <code>declare target</code> Construct	216
55.1 <code>declare target</code> and <code>end declare target</code> for a Function	216
55.2 <code>declare target</code> Construct for Class Type	218
55.3 <code>declare target</code> and <code>end declare target</code> for Variables	219
55.4 <code>declare target</code> and <code>end declare target</code> with <code>declare simd</code>	222
56 <code>teams</code> Constructs	224
56.1 <code>target</code> and <code>teams</code> Constructs with <code>omp_get_num_teams</code> and <code>omp_get_team_num</code> Routines	224
56.2 <code>target</code> , <code>teams</code> , and <code>distribute</code> Constructs	226
56.3 <code>target teams</code> , and Distribute Parallel Loop Constructs	227
56.4 <code>target teams</code> and Distribute Parallel Loop Constructs with Scheduling Clauses	229
56.5 <code>target teams</code> and <code>distribute simd</code> Constructs	230
56.6 <code>target teams</code> and Distribute Parallel Loop SIMD Constructs	232
57 Asynchronous Execution of a <code>target</code> Region Using Tasks	233
58 Array Sections in Device Constructs	238
59 Device Routines	243
59.1 <code>omp_is_initial_device</code> Routine	243
59.2 <code>omp_get_num_devices</code> Routine	245

59.3 <code>omp_set_default_device</code> and <code>omp_get_default_device</code> 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

1

Introduction

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

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

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

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

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

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

23 <http://www.openmp.org>

1

Examples

2

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

C / C++

3

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

C / C++

1 CHAPTER 1

2 A Simple Parallel Loop

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

C / C++

6 Example ploop.1c

```
S-1 void simple(int n, float *a, float *b)
S-2 {
S-3     int i;
S-4
S-5 #pragma omp parallel for
S-6     for (i=1; i<n; i++) /* i is private by default */
S-7         b[i] = (a[i] + a[i-1]) / 2.0;
S-8 }
```

C / C++

Fortran

7 Example ploop.1f

```
S-1      SUBROUTINE SIMPLE(N, A, B)
S-2
S-3      INTEGER I, N
S-4      REAL B(N), A(N)
S-5
S-6      !$OMP PARALLEL DO !I is private by default
S-7          DO I=2,N
S-8              B(I) = (A(I) + A(I-1)) / 2.0
S-9          ENDDO
S-10     !$OMP END PARALLEL DO
S-11
S-12      END SUBROUTINE SIMPLE
```

Fortran

1 CHAPTER 2

2

The OpenMP Memory Model

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

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

C / C++

10 Example *mem_model.1c*

```
S-1 #include <stdio.h>
S-2 #include <omp.h>
S-3
S-4 int main() {
S-5     int x;
S-6
S-7     x = 2;
S-8     #pragma omp parallel num_threads(2) shared(x)
S-9     {
S-10
S-11         if (omp_get_thread_num() == 0) {
S-12             x = 5;
S-13         } else {
S-14             /* Print 1: the following read of x has a race */
S-15             printf("1: Thread# %d: x = %d\n", omp_get_thread_num(),x );
S-16         }
S-17
S-18         #pragma omp barrier
S-19
S-20         if (omp_get_thread_num() == 0) {
```

```

S-21     /* Print 2 */
S-22     printf("2: Thread# %d: x = %d\n", omp_get_thread_num(),x );
S-23 } else {
S-24     /* Print 3 */
S-25     printf("3: Thread# %d: x = %d\n", omp_get_thread_num(),x );
S-26 }
S-27 }
S-28 return 0;
S-29 }
```

C / C++
Fortran

1 *Example mem_model.If*

```

S-1 PROGRAM MEMMODEL
S-2 INCLUDE "omp_lib.h"          ! or USE OMP_LIB
S-3 INTEGER X
S-4
S-5 X = 2
S-6 !$OMP PARALLEL NUM_THREADS(2) SHARED(X)
S-7
S-8     IF (OMP_GET_THREAD_NUM() .EQ. 0) THEN
S-9         X = 5
S-10    ELSE
S-11        ! PRINT 1: The following read of x has a race
S-12        PRINT *, "1: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
S-13    ENDIF
S-14
S-15 !$OMP BARRIER
S-16
S-17     IF (OMP_GET_THREAD_NUM() .EQ. 0) THEN
S-18         ! PRINT 2
S-19         PRINT *, "2: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
S-20     ELSE
S-21         ! PRINT 3
S-22         PRINT *, "3: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
S-23     ENDIF
S-24
S-25 !$OMP END PARALLEL
S-26
S-27 END PROGRAM MEMMODEL
```

Fortran

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

1

Example mem_model.2c

```

S-1 #include <omp.h>
S-2 #include <stdio.h>
S-3 int main()
S-4 {
S-5     int data;
S-6     int flag=0;
S-7     #pragma omp parallel num_threads(2)
S-8     {
S-9         if (omp_get_thread_num()==0)
S-10         {
S-11             /* Write to the data buffer that will be
S-12             read by thread */
S-13             data = 42;
S-14             /* Flush data to thread 1 and strictly order
S-15             the write to data
S-16             relative to the write to the flag */
S-17             #pragma omp flush(flag, data)
S-18             /* Set flag to release thread 1 */
S-19             flag = 1;
S-20             /* Flush flag to ensure that thread 1 sees
S-21             the change */
S-22             #pragma omp flush(flag)
S-23         }
S-24         else if(omp_get_thread_num()==1)
S-25         {
S-26             /* Loop until we see the update to the flag */
S-27             #pragma omp flush(flag, data)
S-28             while (flag < 1)
S-29             {
S-30                 #pragma omp flush(flag, data)
S-31             }
S-32             /* Values of flag and data are undefined */
S-33             printf("flag=%d data=%d\n", flag, data);
S-34             #pragma omp flush(flag, data)
S-35             /* Values data will be 42, value of flag
S-36             still undefined */
S-37             printf("flag=%d data=%d\n", flag, data);
S-38         }
S-39     }
S-40     return 0;
S-41 }
```

1 Example mem_model.2f

```

S-1      PROGRAM EXAMPLE
S-2      INCLUDE "omp_lib.h" ! or USE OMP_LIB
S-3      INTEGER DATA
S-4      INTEGER FLAG
S-5
S-6      FLAG = 0
S-7      !$OMP PARALLEL NUM_THREADS(2)
S-8      IF(OMP_GET_THREAD_NUM() .EQ. 0) THEN
S-9          ! Write to the data buffer that will be read by thread 1
S-10         DATA = 42
S-11         ! Flush DATA to thread 1 and strictly order the write to DATA
S-12         ! relative to the write to the FLAG
S-13         !$OMP FLUSH(FLAG, DATA)
S-14         ! Set FLAG to release thread 1
S-15         FLAG = 1;
S-16         ! Flush FLAG to ensure that thread 1 sees the change */
S-17         !$OMP FLUSH(FLAG)
S-18     ELSE IF(OMP_GET_THREAD_NUM() .EQ. 1) THEN
S-19         ! Loop until we see the update to the FLAG
S-20         !$OMP FLUSH(FLAG, DATA)
S-21         DO WHILE(FLAG .LT. 1)
S-22             !$OMP FLUSH(FLAG, DATA)
S-23         ENDDO
S-24
S-25         ! Values of FLAG and DATA are undefined
S-26         PRINT *, 'FLAG=', FLAG, ' DATA=', DATA
S-27         !$OMP FLUSH(FLAG, DATA)
S-28
S-29         !Values DATA will be 42, value of FLAG still undefined */
S-30         PRINT *, 'FLAG=', FLAG, ' DATA=', DATA
S-31     ENDIF
S-32     !$OMP END PARALLEL
S-33 END

```

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

1

Example mem_model.3c

```

S-1 #include <omp.h>
S-2 #include <stdio.h>
S-3 int main()
S-4 {
S-5     int flag=0;
S-6
S-7     #pragma omp parallel num_threads(3)
S-8     {
S-9         if(omp_get_thread_num()==0)
S-10        {
S-11            /* Set flag to release thread 1 */
S-12            #pragma omp atomic update
S-13            flag++;
S-14            /* Flush of flag is implied by the atomic directive */
S-15        }
S-16        else if(omp_get_thread_num()==1)
S-17        {
S-18            /* Loop until we see that flag reaches 1*/
S-19            #pragma omp flush(flag)
S-20            while(flag < 1)
S-21            {
S-22                #pragma omp flush(flag)
S-23            }
S-24            printf("Thread 1 awoken\n");
S-25
S-26            /* Set flag to release thread 2 */
S-27            #pragma omp atomic update
S-28            flag++;
S-29            /* Flush of flag is implied by the atomic directive */
S-30        }
S-31        else if(omp_get_thread_num()==2)
S-32        {
S-33            /* Loop until we see that flag reaches 2 */
S-34            #pragma omp flush(flag)
S-35            while(flag < 2)
S-36            {
S-37                #pragma omp flush(flag)
S-38            }
S-39            printf("Thread 2 awoken\n");
S-40        }
S-41    }
S-42    return 0;
S-43 }
```

1

Example mem_model.3f

```

S-1      PROGRAM EXAMPLE
S-2      INCLUDE "omp_lib.h" ! or USE OMP_LIB
S-3      INTEGER FLAG
S-4
S-5      FLAG = 0
S-6      !$OMP PARALLEL NUM_THREADS(3)
S-7      IF(OMP_GET_THREAD_NUM() .EQ. 0) THEN
S-8          ! Set flag to release thread 1
S-9          !$OMP ATOMIC UPDATE
S-10         FLAG = FLAG + 1
S-11         !Flush of FLAG is implied by the atomic directive
S-12     ELSE IF(OMP_GET_THREAD_NUM() .EQ. 1) THEN
S-13         ! Loop until we see that FLAG reaches 1
S-14         !$OMP FLUSH(FLAG, DATA)
S-15         DO WHILE(FLAG .LT. 1)
S-16             !$OMP FLUSH(FLAG, DATA)
S-17         ENDDO
S-18
S-19         PRINT *, 'Thread 1 awoken'
S-20
S-21         ! Set FLAG to release thread 2
S-22         !$OMP ATOMIC UPDATE
S-23         FLAG = FLAG + 1
S-24         !Flush of FLAG is implied by the atomic directive
S-25     ELSE IF(OMP_GET_THREAD_NUM() .EQ. 2) THEN
S-26         ! Loop until we see that FLAG reaches 2
S-27         !$OMP FLUSH(FLAG, DATA)
S-28         DO WHILE(FLAG .LT. 2)
S-29             !$OMP FLUSH(FLAG, DATA)
S-30         ENDDO
S-31
S-32         PRINT *, 'Thread 2 awoken'
S-33     ENDIF
S-34     !$OMP END PARALLEL
S-35 END

```

1 CHAPTER 3

2

Conditional Compilation

C / C++

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

5 Example `cond_comp.1c`

```
S-1 #include <stdio.h>
S-2
S-3 int main()
S-4 {
S-5
S-6 # ifdef _OPENMP
S-7     printf("Compiled by an OpenMP-compliant implementation.\n");
S-8 # endif
S-9
S-10    return 0;
S-11 }
```

C / C++

Fortran

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

9 Example `cond_comp.1f`

```
S-1      PROGRAM EXAMPLE
S-2
S-3      C234567890
S-4      !$      PRINT *, "Compiled by an OpenMP-compliant implementation."
S-5
S-6      END PROGRAM EXAMPLE
```

Fortran

1 CHAPTER 4

2 Internal Control Variables (ICVs)

3 According to Section 2.3 of the OpenMP 4.0 specification, an OpenMP implementation must act as
4 if there are ICVs that control the behavior of the program. This example illustrates two ICVs,
5 *nthreads-var* and *max-active-levels-var*. The *nthreads-var* ICV controls the number of threads
6 requested for encountered parallel regions; there is one copy of this ICV per task. The
7 *max-active-levels-var* ICV controls the maximum number of nested active parallel regions; there is
8 one copy of this ICV for the whole program.

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

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

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

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

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

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

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

3 The print statement in an inner **parallel** region is also executed by only one of the threads in the
4 team. Since we have a total of two inner **parallel** regions, the print statement will be executed
5 twice – once per inner **parallel** region.

C / C++

6 Example icv.1c

```
S-1 #include <stdio.h>
S-2 #include <omp.h>
S-3
S-4 int main (void)
S-5 {
S-6     omp_set_nested(1);
S-7     omp_set_max_active_levels(8);
S-8     omp_set_dynamic(0);
S-9     omp_set_num_threads(2);
S-10    #pragma omp parallel
S-11    {
S-12        omp_set_num_threads(3);
S-13
S-14        #pragma omp parallel
S-15        {
S-16            omp_set_num_threads(4);
S-17            #pragma omp single
S-18            {
S-19                /*
S-20                 * The following should print:
S-21                 * Inner: max_act_lev=8, num_thds=3, max_thds=4
S-22                 * Inner: max_act_lev=8, num_thds=3, max_thds=4
S-23                 */
S-24                 printf ("Inner: max_act_lev=%d, num_thds=%d, max_thds=%d\n",
S-25                 omp_get_max_active_levels(), omp_get_num_threads(),
S-26                 omp_get_max_threads());
S-27             }
S-28         }
S-29
S-30         #pragma omp barrier
S-31         #pragma omp single
S-32         {
S-33             /*
S-34             * The following should print:
S-35             * Outer: max_act_lev=8, num_thds=2, max_thds=3
S-36             */
S-37             printf ("Outer: max_act_lev=%d, num_thds=%d, max_thds=%d\n",

```

```

S-38             omp_get_max_active_levels(), omp_get_num_threads(),
S-39             omp_get_max_threads());
S-40         }
S-41     }
S-42     return 0;
S-43 }
```

C / C++
Fortran

1 *Example icv.If*

```

S-1      program icv
S-2      use omp_lib
S-3
S-4      call omp_set_nested(.true.)
S-5      call omp_set_max_active_levels(8)
S-6      call omp_set_dynamic(.false.)
S-7      call omp_set_num_threads(2)
S-8
S-9      !$omp parallel
S-10     call omp_set_num_threads(3)
S-11
S-12     !$omp parallel
S-13     call omp_set_num_threads(4)
S-14     !$omp single
S-15     !       The following should print:
S-16     !       Inner: max_act_lev= 8 , num_thds= 3 , max_thds= 4
S-17     !       Inner: max_act_lev= 8 , num_thds= 3 , max_thds= 4
S-18     print *, "Inner: max_act_lev=", omp_get_max_active_levels(),
S-19     &           ", num_thds=", omp_get_num_threads(),
S-20     &           ", max_thds=", omp_get_max_threads()
S-21     !$omp end single
S-22     !$omp end parallel
S-23
S-24     !$omp barrier
S-25     !$omp single
S-26     !       The following should print:
S-27     !       Outer: max_act_lev= 8 , num_thds= 2 , max_thds= 3
S-28     print *, "Outer: max_act_lev=", omp_get_max_active_levels(),
S-29     &           ", num_thds=", omp_get_num_threads(),
S-30     &           ", max_thds=", omp_get_max_threads()
S-31     !$omp end single
S-32     !$omp end parallel
S-33     end
```

Fortran

1 CHAPTER 5

2 **The parallel Construct**

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

C / C++

6 Example parallel.1c

```
S-1 #include <omp.h>
S-2
S-3 void subdomain(float *x, int istart, int ipoints)
S-4 {
S-5     int i;
S-6
S-7     for (i = 0; i < ipoints; i++)
S-8         x[istart+i] = 123.456;
S-9 }
S-10
S-11 void sub(float *x, int npoints)
S-12 {
S-13     int iam, nt, ipoints, istart;
S-14
S-15 #pragma omp parallel default(shared) private(iam,nt,ipoints,istart)
S-16 {
S-17     iam = omp_get_thread_num();
S-18     nt = omp_get_num_threads();
S-19     ipoints = npoints / nt; /* size of partition */
S-20     istart = iam * ipoints; /* starting array index */
S-21     if (iam == nt-1) /* last thread may do more */
S-22         ipoints = npoints - istart;
S-23     subdomain(x, istart, ipoints);
S-24 }
S-25 }
```

```

S-26
S-27     int main()
S-28 {
S-29     float array[10000];
S-30
S-31     sub(array, 10000);
S-32
S-33     return 0;
S-34 }
```

C / C++
Fortran

1 Example parallel. If

```

S-1      SUBROUTINE SUBDOMAIN(X, ISTART, IPOINTS)
S-2          INTEGER ISTART, IPOINTS
S-3          REAL X(*)
S-4
S-5          INTEGER I
S-6
S-7          DO 100 I=1, IPOINTS
S-8              X(ISTART+I) = 123.456
S-9          100      CONTINUE
S-10
S-11         END SUBROUTINE SUBDOMAIN
S-12
S-13         SUBROUTINE SUB(X, NPOINTS)
S-14             INCLUDE "omp_lib.h"      ! or USE OMP_LIB
S-15
S-16             REAL X(*)
S-17             INTEGER NPOINTS
S-18             INTEGER IAM, NT, IPOINTS, ISTART
S-19
S-20             !$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X, NPOINTS)
S-21
S-22                 IAM = OMP_GET_THREAD_NUM()
S-23                 NT = OMP_GET_NUM_THREADS()
S-24                 IPOINTS = NPOINTS/NT
S-25                 ISTART = IAM * IPOINTS
S-26                 IF (IAM .EQ. NT-1) THEN
S-27                     IPOINTS = NPOINTS - ISTART
S-28                 ENDIF
S-29                 CALL SUBDOMAIN(X, ISTART, IPOINTS)
S-30
S-31             !$OMP END PARALLEL
S-32             END SUBROUTINE SUB
S-33
```

```
S-34      PROGRAM PAREXAMPLE
S-35          REAL ARRAY(10000)
S-36          CALL SUB(ARRAY, 10000)
S-37      END PROGRAM PAREXAMPLE
```

Fortran

1 CHAPTER 6

2

3 Controlling the Number of Threads on Multiple Nesting Levels

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

C / C++

6 Example *nths_nesting.1c*

```
S-1 #include <stdio.h>
S-2 #include <omp.h>
S-3 int main (void)
S-4 {
S-5     omp_set_nested(1);
S-6     omp_set_dynamic(0);
S-7     #pragma omp parallel
S-8     {
S-9         #pragma omp parallel
S-10        {
S-11            #pragma omp single
S-12            {
S-13                /*
S-14                 * If OMP_NUM_THREADS=2,3 was set, the following should print:
S-15                 * Inner: num_thds=3
S-16                 * Inner: num_thds=3
S-17                 *
S-18                 * If nesting is not supported, the following should print:
S-19                 * Inner: num_thds=1
S-20                 * Inner: num_thds=1
S-21                 */
S-22             printf ("Inner: num_thds=%d\n", omp_get_num_threads());
S-23         }
S-24     }
```

```

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 }
```

C / C++

Fortran

1

Example nthrs_nesting.f

```

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 !$omp single
S-8         ! If OMP_NUM_THREADS=2,3 was set, the following should print:
S-9         ! Inner: num_thds= 3
S-10        ! Inner: num_thds= 3
S-11        ! If nesting is not supported, the following should print:
S-12        ! Inner: num_thds= 1
S-13        ! Inner: num_thds= 1
S-14         print *, "Inner: num_thds=", omp_get_num_threads()
S-15 !$omp end single
```

```
S-16    !$omp end parallel
S-17    !$omp barrier
S-18        call omp_set_nested(.false.)
S-19    !$omp parallel
S-20    !$omp single
S-21        ! Even if OMP_NUM_THREADS=2,3 was set, the following should print,
S-22        ! because nesting is disabled:
S-23        ! Inner: num_thds= 1
S-24        ! Inner: num_thds= 1
S-25        print *, "Inner: num_thds=", omp_get_num_threads()
S-26    !$omp end single
S-27    !$omp end parallel
S-28    !$omp barrier
S-29    !$omp single
S-30        ! If OMP_NUM_THREADS=2,3 was set, the following should print:
S-31        ! Outer: num_thds= 2
S-32        print *, "Outer: num_thds=", omp_get_num_threads()
S-33    !$omp end single
S-34    !$omp end parallel
S-35        end
```

Fortran

1 CHAPTER 7

2

Interaction Between the `num_threads` 3 Clause and `omp_set_dynamic`

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

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

C / C++

10 Example `nths_dynamic.1c`

```
S-1 #include <omp.h>
S-2 int main()
S-3 {
S-4     omp_set_dynamic(0);
S-5     #pragma omp parallel num_threads(10)
S-6     {
S-7         /* do work here */
S-8     }
S-9     return 0;
S-10 }
```

C / C++

Fortran

1 Example *nths_dynamic.If*

```
S-1      PROGRAM EXAMPLE
S-2          INCLUDE "omp_lib.h"      ! or USE OMP_LIB
S-3          CALL OMP_SET_DYNAMIC(.FALSE.)
S-4      !$OMP    PARALLEL NUM_THREADS(10)
S-5          ! do work here
S-6      !$OMP    END PARALLEL
S-7      END PROGRAM EXAMPLE
```

Fortran

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

C / C++

4 Example *nths_dynamic.2c*

```
S-1 #include <omp.h>
S-2 int main()
S-3 {
S-4     omp_set_dynamic(1);
S-5     #pragma omp parallel num_threads(10)
S-6     {
S-7         /* do work here */
S-8     }
S-9     return 0;
S-10 }
```

C / C++

Fortran

5 Example *nths_dynamic.2f*

```
S-1      PROGRAM EXAMPLE
S-2          INCLUDE "omp_lib.h"      ! or USE OMP_LIB
S-3          CALL OMP_SET_DYNAMIC(.TRUE.)
S-4      !$OMP    PARALLEL NUM_THREADS(10)
S-5          ! do work here
S-6      !$OMP    END PARALLEL
S-7      END PROGRAM EXAMPLE
```

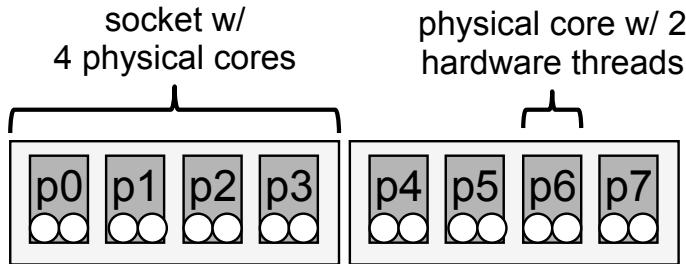
Fortran

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

1 **CHAPTER 8**

2 **The proc_bind Clause**

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



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

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

11 or

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

13 **8.1 Spread Affinity Policy**

14 The following example shows the result of the **spread** affinity policy on the partition list when the
15 number of threads is less than or equal to the number of places in the parent's place partition, for

1 the machine architecture depicted above. Note that the threads are bound to the first place of each
2 subpartition.

C / C++

3 *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

4 *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
```

Fortran

5 It is unspecified on which place the master thread is initially started. If the master thread is initially
6 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

11 If the master thread would initially be started on p2, the placement of threads and distribution of the
12 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

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

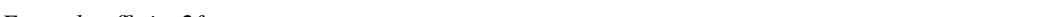
3 Let T be the number of threads in the team, and P be the number of places in the parent's place
4 partition. The first T/P threads of the team (including the master thread) execute on the parent's
5 place. The next T/P threads execute on the next place in the place partition, and so on, with wrap
6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

7  C / C++ 

8 *Example affinity.2c*

```
S-1 void work();  
S-2 void foo()  
S-3 {  
S-4     #pragma omp parallel num_threads(16) proc_bind(spread)  
S-5     {  
S-6         work();  
S-7     }  
S-8 }
```

9  C / C++ 

10 Fortran 

8 *Example affinity.2f*

```
S-1 subroutine foo  
S-2 !$omp parallel num_threads(16) proc_bind(spread)  
S-3     call work()  
S-4 !$omp end parallel  
S-5 end subroutine
```

9  Fortran 

10 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:

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

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

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

- ```

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

```

## 8 8.2 Close Affinity Policy

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



C / C++

12     *Example affinity.3c*

```

S-1 void work();
S-2 int main()
S-3 {
S-4 #pragma omp parallel proc_bind(close) num_threads(4)
S-5 {
S-6 work();
S-7 }
S-8 return 0;
S-9 }
```



C / C++



Fortran

13     *Example affinity.3f*

```

S-1 PROGRAM EXAMPLE
S-2 !$OMP PARALLEL PROC_BIND(CLOSE) NUM_THREADS(4)
S-3 CALL WORK()
S-4 !$OMP END PARALLEL
S-5 END PROGRAM EXAMPLE
```

## Fortran

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

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

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

- 9 • thread 0 executes on p2 with the place partition p0-p7
- 10 • thread 1 executes on p3 with the place partition p0-p7
- 11 • thread 2 executes on p4 with the place partition p0-p7
- 12 • thread 3 executes on p5 with the place partition p0-p7

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

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

## C / C++

19 Example affinity.4c

```
S-1 void work();
S-2 void foo()
S-3 {
S-4 #pragma omp parallel num_threads(16) proc_bind(close)
S-5 {
S-6 work();
S-7 }
S-8 }
```

## C / C++

## Fortran

20 Example affinity.4f

```
S-1 subroutine foo
S-2 !$omp parallel num_threads(16) proc_bind(close)
S-3 call work()
S-4 !$omp end parallel
S-5 end subroutine
```

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

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

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

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

## 21 8.3 Master Affinity Policy

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

C / C++

1      *Example affinity.5c*

```
S-1 void work();
S-2 int main()
S-3 {
S-4 #pragma omp parallel proc_bind(master) num_threads(4)
S-5 {
S-6 work();
S-7 }
S-8 return 0;
S-9 }
```

C / C++

Fortran

2      *Example affinity.5f*

```
S-1 PROGRAM EXAMPLE
S-2 !$OMP PARALLEL PROC_BIND(MASTER) NUM_THREADS(4)
S-3 CALL WORK()
S-4 !$OMP END PARALLEL
S-5 END PROGRAM EXAMPLE
```

Fortran

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

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

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

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

1 CHAPTER 9

2

## 3 Fortran Restrictions on the do Construct

---

Fortran

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

7 Example fort\_do\_1f

```
S-1 SUBROUTINE WORK(I, J)
S-2 INTEGER I,J
S-3 END SUBROUTINE WORK
S-4
S-5 SUBROUTINE DO_GOOD ()
S-6 INTEGER I, J
S-7 REAL A(1000)
S-8
S-9 DO 100 I = 1,10
S-10 !$OMP DO
S-11 DO 100 J = 1,10
S-12 CALL WORK(I,J)
S-13 100 CONTINUE ! !$OMP ENDDO implied here
S-14
S-15 !$OMP DO
S-16 DO 200 J = 1,10
S-17 200 A(I) = I + 1
S-18 !$OMP ENDDO
S-19
S-20 !$OMP DO
S-21 DO 300 I = 1,10
S-22 DO 300 J = 1,10
S-23 CALL WORK(I,J)
```

```
S-24 300 CONTINUE
S-25 !$OMP ENDDO
S-26 END SUBROUTINE DO_GOOD
```

1 The following example is non-conforming because the matching `do` directive for the `end do` does  
2 not precede the outermost loop:

3 Example for `fort_do.2f`

```
S-1 SUBROUTINE WORK(I, J)
S-2 INTEGER I, J
S-3 END SUBROUTINE WORK
S-4
S-5 SUBROUTINE DO_WRONG
S-6 INTEGER I, J
S-7
S-8 DO 100 I = 1,10
S-9 !$OMP DO
S-10 DO 100 J = 1,10
S-11 CALL WORK(I,J)
S-12 100 CONTINUE
S-13 !$OMP ENDDO
S-14 END SUBROUTINE DO_WRONG
```

Fortran

## 1 CHAPTER 10

# 2 Fortran Private Loop Iteration Variables

---

### Fortran

3 In general loop iteration variables will be private, when used in the *do-loop* of a **do** and  
4 **parallel do** construct or in sequential loops in a **parallel** construct (see Section 2.7.1 and  
5 Section 2.14.1 of the OpenMP 4.0 specification). In the following example of a sequential loop in a  
6 **parallel** construct the loop iteration variable *I* will be private.

7 Example *fort\_loopvar.1f*

```
S-1 SUBROUTINE PLOOP_1(A,N)
S-2 INCLUDE "omp_lib.h" ! or USE OMP_LIB
S-3
S-4 REAL A(*)
S-5 INTEGER I, MYOFFSET, N
S-6
S-7 !$OMP PARALLEL PRIVATE(MYOFFSET)
S-8 MYOFFSET = OMP_GET_THREAD_NUM()*N
S-9 DO I = 1, N
S-10 A(MYOFFSET+I) = FLOAT(I)
S-11 ENDDO
S-12 !$OMP END PARALLEL
S-13
S-14 END SUBROUTINE PLOOP_1
```

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

9 Example *fort\_loopvar.2f*

```
S-1 SUBROUTINE PLOOP_2(A,B,N,I1,I2)
S-2 REAL A(*), B(*)
S-3 INTEGER I1, I2, N
S-4
S-5 !$OMP PARALLEL SHARED(A,B,I1,I2)
S-6 !$OMP SECTIONS
```

```
S-7 !$OMP SECTION
S-8 DO I1 = I1, N
S-9 IF (A(I1).NE.0.0) EXIT
S-10 ENDDO
S-11 !$OMP SECTION
S-12 DO I2 = I2, N
S-13 IF (B(I2).NE.0.0) EXIT
S-14 ENDDO
S-15 !$OMP END SECTIONS
S-16 !$OMP SINGLE
S-17 IF (I1.LE.N) PRINT *, 'ITEMS IN A UP TO ', I1, 'ARE ALL ZERO.'
S-18 IF (I2.LE.N) PRINT *, 'ITEMS IN B UP TO ', I2, 'ARE ALL ZERO.'
S-19 !$OMP END SINGLE
S-20 !$OMP END PARALLEL
S-21
S-22 END SUBROUTINE PLOOP_2
```

1

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

Fortran

1 CHAPTER 11

2 **The nowait Clause**

---

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

C / C++

5 Example *nowait.1c*

```
S-1 #include <math.h>
S-2
S-3 void nowait_example(int n, int m, float *a, float *b, float *y, float *z)
S-4 {
S-5 int i;
S-6 #pragma omp parallel
S-7 {
S-8 #pragma omp for nowait
S-9 for (i=1; i<n; i++)
S-10 b[i] = (a[i] + a[i-1]) / 2.0;
S-11
S-12 #pragma omp for nowait
S-13 for (i=0; i<m; i++)
S-14 y[i] = sqrt(z[i]);
S-15 }
S-16 }
```

C / C++

## Fortran

1      Example nowait.If

```
S-1 SUBROUTINE NOWAIT_EXAMPLE(N, M, A, B, Y, Z)
S-2
S-3 INTEGER N, M
S-4 REAL A(*), B(*), Y(*), Z(*)
S-5
S-6 INTEGER I
S-7
S-8 !$OMP PARALLEL
S-9
S-10 !$OMP DO
S-11 DO I=2,N
S-12 B(I) = (A(I) + A(I-1)) / 2.0
S-13 ENDDO
S-14 !$OMP END DO NOWAIT
S-15
S-16 !$OMP DO
S-17 DO I=1,M
S-18 Y(I) = SQRT(Z(I))
S-19 ENDDO
S-20 !$OMP END DO NOWAIT
S-21
S-22 !$OMP END PARALLEL
S-23
S-24 END SUBROUTINE NOWAIT_EXAMPLE
```

## Fortran

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

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

1

*Example nowait.2c*

```

S-1
S-2 #include <math.h>
S-3 void nowait_example2(int n, float *a, float *b, float *c, float *y, float
S-4 *z)
S-5 {
S-6 int i;
S-7 #pragma omp parallel
S-8 {
S-9 #pragma omp for schedule(static) nowait
S-10 for (i=0; i<n; i++)
S-11 c[i] = (a[i] + b[i]) / 2.0f;
S-12 #pragma omp for schedule(static) nowait
S-13 for (i=0; i<n; i++)
S-14 z[i] = sqrtf(c[i]);
S-15 #pragma omp for schedule(static) nowait
S-16 for (i=1; i<=n; i++)
S-17 y[i] = z[i-1] + a[i];
S-18 }
S-19 }
```

2

*Example nowait.2f*

```

S-1 SUBROUTINE NOWAIT_EXAMPLE2(N, A, B, C, Y, Z)
S-2 INTEGER N
S-3 REAL A(*), B(*), C(*), Y(*), Z(*)
S-4 INTEGER I
S-5 !$OMP PARALLEL
S-6 !$OMP DO SCHEDULE(STATIC)
S-7 DO I=1,N
S-8 C(I) = (A(I) + B(I)) / 2.0
S-9 ENDDO
S-10 !$OMP END DO NOWAIT
S-11 !$OMP DO SCHEDULE(STATIC)
S-12 DO I=1,N
S-13 Z(I) = SQRT(C(I))
S-14 ENDDO
S-15 !$OMP END DO NOWAIT
S-16 !$OMP DO SCHEDULE(STATIC)
S-17 DO I=2,N+1
S-18 Y(I) = Z(I-1) + A(I)
S-19 ENDDO
S-20 !$OMP END DO NOWAIT
```

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

Fortran

1 CHAPTER 12

2 

## The collapse Clause

---

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

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

C / C++

12 Example collapse.1c

```
S-1 void bar(float *a, int i, int j, int k);
S-2 int kl, ku, ks, jl, ju, js, il, iu, is;
S-3 void sub(float *a)
S-4 {
S-5 int i, j, k;
S-6 #pragma omp for collapse(2) private(i, k, j)
S-7 for (k=kl; k<=ku; k+=ks)
S-8 for (j=jl; j<=ju; j+=js)
S-9 for (i=il; i<=iu; i+=is)
S-10 bar(a, i, j, k);
S-11 }
```

C / C++

## Fortran

1      Example collapse.1f

```
S-1 subroutine sub(a)
S-2 real a(*)
S-3 integer kl, ku, ks, jl, ju, js, il, iu, is
S-4 common /csub/ kl, ku, ks, jl, ju, js, il, iu, is
S-5 integer i, j, k
S-6 !$omp do collapse(2) private(i,j,k)
S-7 do k = kl, ku, ks
S-8 do j = jl, ju, js
S-9 do i = il, iu, is
S-10 call bar(a,i,j,k)
S-11 enddo
S-12 enddo
S-13 enddo
S-14 !$omp end do
S-15 end subroutine
```

## Fortran

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

5      The sequential execution of the iterations in the **k** and **j** loops determines the order of the iterations  
6      in the collapsed iteration space. This implies that in the sequentially last iteration of the collapsed  
7      iteration space, **k** will have the value **2** and **j** will have the value **3**. Since **klast** and **jlast** are  
8      **lastprivate**, their values are assigned by the sequentially last iteration of the collapsed **k** and **j**  
9      loop. This example prints: **2 3**.

## C / C++

10     Example collapse.2c

```
S-1 #include <stdio.h>
S-2 void test()
S-3 {
S-4 int j, k, jlast, klast;
S-5 #pragma omp parallel
S-6 {
S-7 #pragma omp for collapse(2) lastprivate(jlast, klast)
S-8 for (k=1; k<=2; k++)
S-9 for (j=1; j<=3; j++)
S-10 {
S-11 jlast=j;
S-12 klast=k;
S-13 }
S-14 #pragma omp single
```

```
S-15 printf("%d %d\n", klast, jlast);
S-16 }
S-17 }
```



1       *Example collapse.2f*

```
S-1 program test
S-2 !$omp parallel
S-3 !$omp do private(j,k) collapse(2) lastprivate(jlast, klast)
S-4 do k = 1,2
S-5 do j = 1,3
S-6 jlast=j
S-7 klast=k
S-8 enddo
S-9 enddo
S-10 !$omp end do
S-11 !$omp single
S-12 print *, klast, jlast
S-13 !$omp end single
S-14 !$omp end parallel
S-15 end program test
```



2       The next example illustrates the interaction of the **collapse** and **ordered** clauses.

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

8       According to Section 2.12.8 of the OpenMP 4.0 specification, a thread must not execute more than  
9       one ordered region that binds to the same loop region. So the **collapse** clause is required for the  
10      example to be conforming. With the **collapse** clause, the iterations of the **k** and **j** loops are  
11      collapsed into one loop, and therefore only one ordered region will bind to the collapsed **k** and **j**  
12      loop. Without the **collapse** clause, there would be two ordered regions that bind to each  
13      iteration of the **k** loop (one arising from the first iteration of the **j** loop, and the other arising from  
14      the second iteration of the **j** loop).

15      The code prints

```
16 0 1 1
17 0 1 2
18 0 2 1
19 1 2 2
```

```
1 1 3 1
2 1 3 2
```

C / C++

3 Example collapse.3c

```
S-1 #include <omp.h>
S-2 #include <stdio.h>
S-3 void work(int a, int j, int k);
S-4 void sub()
{
 S-5 int j, k, a;
 S-6 #pragma omp parallel num_threads(2)
 S-7 {
 S-8 #pragma omp for collapse(2) ordered private(j,k) schedule(static,3)
 S-9 for (k=1; k<=3; k++)
 S-10 for (j=1; j<=2; j++)
 S-11 {
 S-12 #pragma omp ordered
 S-13 printf("%d %d %d\n", omp_get_thread_num(), k, j);
 S-14 /* end ordered */
 S-15 work(a, j, k);
 S-16 }
 S-17 }
 S-18 }
S-19}
```

C / C++

Fortran

4 Example collapse.3f

```
S-1 program test
S-2 include 'omp_lib.h'
S-3 !$omp parallel num_threads(2)
S-4 !$omp do collapse(2) ordered private(j,k) schedule(static,3)
S-5 do k = 1,3
S-6 do j = 1,2
S-7 !$omp ordered
S-8 print *, omp_get_thread_num(), k, j
S-9 !$omp end ordered
S-10 call work(a, j, k)
S-11 enddo
S-12 enddo
S-13 !$omp end do
S-14 !$omp end parallel
S-15 end program test
```

Fortran

1 CHAPTER 13

2 **The parallel sections Construct**

---

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

C / C++

6 Example psections.1c

```
S-1 void XAXIS();
S-2 void YAXIS();
S-3 void ZAXIS();
S-4
S-5 void sect_example()
S-6 {
S-7 #pragma omp parallel sections
S-8 {
S-9 #pragma omp section
S-10 XAXIS();
S-11
S-12 #pragma omp section
S-13 YAXIS();
S-14
S-15 #pragma omp section
S-16 ZAXIS();
S-17 }
S-18 }
```

C / C++



## Fortran

1

*Example psections.If*

```
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
```



## Fortran

1 CHAPTER 14

2

## 3 The **firstprivate** Clause and the **sections** Construct

---

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

C / C++

11 Example fpriv\_sections.1c

```
S-1 #include <omp.h>
S-2 #include <stdio.h>
S-3 #define NT 4
S-4 int main() {
S-5 int section_count = 0;
S-6 omp_set_dynamic(0);
S-7 omp_set_num_threads(NT);
S-8 #pragma omp parallel
S-9 #pragma omp sections firstprivate(section_count)
S-10 {
S-11 #pragma omp section
S-12 {
S-13 section_count++;
S-14 /* may print the number one or two */
S-15 printf("section_count %d\n", section_count);
S-16 }
S-17 #pragma omp section
S-18 {
```

```
S-19 section_count++;
S-20 /* may print the number one or two */
S-21 printf("section_count %d\n", section_count);
S-22 }
S-23 }
S-24 return 0;
S-25 }
```

C / C++

Fortran

1

*Example fpriv\_sections.If*

```
S-1 program section
S-2 use omp_lib
S-3 integer :: section_count = 0
S-4 integer, parameter :: NT = 4
S-5 call omp_set_dynamic(.false.)
S-6 call omp_set_num_threads(NT)
S-7 !$omp parallel
S-8 !$omp sections firstprivate (section_count)
S-9 !$omp section
S-10 section_count = section_count + 1
S-11 ! may print the number one or two
S-12 print *, 'section_count', section_count
S-13 !$omp section
S-14 section_count = section_count + 1
S-15 ! may print the number one or two
S-16 print *, 'section_count', section_count
S-17 !$omp end sections
S-18 !$omp end parallel
S-19 end program section
```

Fortran

## 1 CHAPTER 15

# 2 The **single** Construct

---

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

C / C++

9 Example *single.1c*

```
S-1 #include <stdio.h>
S-2
S-3 void work1() {}
S-4 void work2() {}
S-5
S-6 void single_example()
S-7 {
S-8 #pragma omp parallel
S-9 {
S-10 #pragma omp single
S-11 printf("Beginning work1.\n");
S-12
S-13 work1();
S-14
S-15 #pragma omp single
S-16 printf("Finishing work1.\n");
S-17
S-18 #pragma omp single nowait
S-19 printf("Finished work1 and beginning work2.\n");
S-20
S-21 work2();
S-22 }
S-23 }
```



C / C++

Fortran

1

*Example single.If*

```
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
```



Fortran

## 1 CHAPTER 16

# 2 The task and taskwait Constructs

---

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

C / C++

8 Example tasking.1c

```
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 }
```

C / C++

## Fortran

1      Example tasking.1f

```
S-1 RECURSIVE SUBROUTINE traverse (P)
S-2 TYPE Node
S-3 TYPE (Node) , POINTER :: left, right
S-4 END TYPE Node
S-5 TYPE (Node) :: P
S-6 IF (associated(P%left)) THEN
S-7 !$OMP TASK ! P is firstprivate by default
S-8 call traverse(P%left)
S-9 !$OMP END TASK
S-10 ENDIF
S-11 IF (associated(P%right)) THEN
S-12 !$OMP TASK ! P is firstprivate by default
S-13 call traverse(P%right)
S-14 !$OMP END TASK
S-15 ENDIF
S-16 CALL process (P)
S-17 END SUBROUTINE
```

## Fortran

2      In the next example, we force a postorder traversal of the tree by adding a **taskwait** directive.  
3      Now, we can safely assume that the left and right sons have been executed before we process the  
4      current node.

## C / C++

5      Example tasking.2c

```
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 postorder_traverse(struct node *p) {
S-7 if (p->left)
S-8 #pragma omp task // p is firstprivate by default
S-9 postorder_traverse(p->left);
S-10 if (p->right)
S-11 #pragma omp task // p is firstprivate by default
S-12 postorder_traverse(p->right);
S-13 #pragma omp taskwait
S-14 process(p);
S-15 }
```

## C / C++

---

Fortran

---

1       *Example tasking.2f*

```
S-1 RECURSIVE SUBROUTINE traverse (P)
S-2 TYPE Node
S-3 TYPE(Node), POINTER :: left, right
S-4 END TYPE Node
S-5 TYPE(Node) :: P
S-6 IF (associated(P%left)) THEN
S-7 !$OMP TASK ! P is firstprivate by default
S-8 call traverse(P%left)
S-9 !$OMP END TASK
S-10 ENDIF
S-11 IF (associated(P%right)) THEN
S-12 !$OMP TASK ! P is firstprivate by default
S-13 call traverse(P%right)
S-14 !$OMP END TASK
S-15 ENDIF
S-16 !$OMP TASKWAIT
S-17 CALL process (P)
S-18 END SUBROUTINE
```

---

Fortran

---

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

---

C / C++

---

6       *Example tasking.3c*

```
S-1 typedef struct node node;
S-2 struct node {
S-3 int data;
S-4 node * next;
S-5 };
S-6
S-7 void process(node * p)
S-8 {
S-9 /* do work here */
S-10 }
S-11 void increment_list_items(node * head)
S-12 {
S-13 #pragma omp parallel
S-14 {
S-15 #pragma omp single
S-16 {
```

```

S-17 node * p = head;
S-18 while (p) {
S-19 #pragma omp task
S-20 // p is firstprivate by default
S-21 process(p);
S-22 p = p->next;
S-23 }
S-24 }
S-25 }
S-26 }
```

C / C++

Fortran

1

*Example tasking.3f*

```

S-1 MODULE LIST
S-2 TYPE NODE
S-3 INTEGER :: PAYLOAD
S-4 TYPE (NODE), POINTER :: NEXT
S-5 END TYPE NODE
S-6 CONTAINS
S-7 SUBROUTINE PROCESS(P)
S-8 TYPE (NODE), POINTER :: P
S-9 ! do work here
S-10 END SUBROUTINE
S-11 SUBROUTINE INCREMENT_LIST_ITEMS (HEAD)
S-12 TYPE (NODE), POINTER :: HEAD
S-13 TYPE (NODE), POINTER :: P
S-14 !$OMP PARALLEL PRIVATE(P)
S-15 !$OMP SINGLE
S-16 P => HEAD
S-17 DO
S-18 !$OMP TASK
S-19 ! P is firstprivate by default
S-20 CALL PROCESS(P)
S-21 !$OMP END TASK
S-22 P => P%NEXT
S-23 IF (.NOT. ASSOCIATED (P)) EXIT
S-24 END DO
S-25 !$OMP END SINGLE
S-26 !$OMP END PARALLEL
S-27 END SUBROUTINE
S-28 END MODULE
```



## Fortran

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



## C / C++

4      *Example tasking.4c*

```
S-1 int fib(int n) {
S-2 int i, j;
S-3 if (n<2)
S-4 return n;
S-5 else {
S-6 #pragma omp task shared(i)
S-7 i=fib(n-1);
S-8 #pragma omp task shared(j)
S-9 j=fib(n-2);
S-10 #pragma omp taskwait
S-11 return i+j;
S-12 }
S-13 }
```



## C / C++



## Fortran

5      *Example tasking.4f*

```
S-1 RECURSIVE INTEGER FUNCTION fib(n) RESULT(res)
S-2 INTEGER n, i, j
S-3 IF (n .LT. 2) THEN
S-4 res = n
S-5 ELSE
S-6 !$OMP TASK SHARED(i)
S-7 i = fib(n-1)
S-8 !$OMP END TASK
S-9 !$OMP TASK SHARED(j)
S-10 j = fib(n-2)
S-11 !$OMP END TASK
S-12 !$OMP TASKWAIT
S-13 res = i+j
S-14 END IF
S-15 END FUNCTION
```

## Fortran

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

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

## C / C++

9 *Example tasking.5c*

```
S-1 #define LARGE_NUMBER 10000000
S-2 double item[LARGE_NUMBER];
S-3 extern void process(double);
S-4
S-5 int main() {
S-6 #pragma omp parallel
S-7 {
S-8 #pragma omp single
S-9 {
S-10 int i;
S-11 for (i=0; i<LARGE_NUMBER; i++)
S-12 #pragma omp task // i is firstprivate, item is shared
S-13 process(item[i]);
S-14 }
S-15 }
S-16 }
```

## C / C++

---

Fortran

---

1       *Example tasking.5f*

```
S-1 real*8 item(10000000)
S-2 integer i
S-3
S-4 !$omp parallel
S-5 !$omp single ! loop iteration variable i is private
S-6 do i=1,10000000
S-7 !$omp task
S-8 ! i is firstprivate, item is shared
S-9 call process(item(i))
S-10 !$omp end task
S-11 end do
S-12 !$omp end single
S-13 !$omp end parallel
S-14 end
```

---

Fortran

---

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

8       In this case, since the loop is in an untied task, any other thread is eligible to resume the task  
9       generating loop. In the previous examples, the other threads would be forced to idle until the  
10      generating thread finishes its long task, since the task generating loop was in a tied task.

---

C / C++

---

11      *Example tasking.6c*

```
S-1 #define LARGE_NUMBER 10000000
S-2 double item[LARGE_NUMBER];
S-3 extern void process(double);
S-4 int main() {
S-5 #pragma omp parallel
S-6 {
S-7 #pragma omp single
S-8 {
S-9 int i;
S-10 #pragma omp task untied
S-11 // i is firstprivate, item is shared
S-12 {
S-13 for (i=0; i<LARGE_NUMBER; i++)
S-14 #pragma omp task
```

```
S-15 process(item[i]);
S-16 }
S-17 }
S-18 }
S-19 return 0;
S-20 }
```

C / C++

Fortran

1 Example tasking.6f

```
S-1 real*8 item(10000000)
S-2 !$omp parallel
S-3 !$omp single
S-4 !$omp task untied
S-5 ! loop iteration variable i is private
S-6 do i=1,10000000
S-7 !$omp task ! i is firstprivate, item is shared
S-8 call process(item(i))
S-9 !$omp end task
S-10 end do
S-11 !$omp end task
S-12 !$omp end single
S-13 !$omp end parallel
S-14 end
```

Fortran

2 The following two examples demonstrate how the scheduling rules illustrated in Section 2.11.3 of  
3 the OpenMP 4.0 specification affect the usage of **threadprivate** variables in tasks. A  
4 **threadprivate** variable can be modified by another task that is executed by the same thread.  
5 Thus, the value of a **threadprivate** variable cannot be assumed to be unchanged across a task  
6 scheduling point. In untied tasks, task scheduling points may be added in any place by the  
7 implementation.

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

## 1 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 }
S-17 var = tp; //value of tp can be 1 or 2
S-18 }
S-19 tp = 2;
S-20 }
S-21 }
S-22 }
```

## 2 Example tasking.7f

```

S-1 module example
S-2 integer tp
S-3 !$omp threadprivate(tp)
S-4 integer var
S-5 contains
S-6 subroutine work
S-7 !$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
```

```
S-18 !$omp end task
S-19 end subroutine
S-20 end module
```

Fortran

1 In this example, scheduling constraints prohibit a thread in the team from executing a new task that  
2 modifies **tp** while another such task region tied to the same thread is suspended. Therefore, the  
3 value written will persist across the task scheduling point.

C / C++

4 *Example tasking.8c*

```
S-1
S-2 int tp;
S-3 #pragma omp threadprivate(tp)
S-4 int var;
S-5 void work()
S-6 {
S-7 #pragma omp parallel
S-8 {
S-9 /* do work here */
S-10 #pragma omp task
S-11 {
S-12 tp++;
S-13 /* do work here */
S-14 #pragma omp task
S-15 {
S-16 /* do work here but don't modify tp */
S-17 }
S-18 var = tp; //Value does not change after write above
S-19 }
S-20 }
S-21 }
```

C / C++

1       *Example tasking.8f*

```

S-1 module example
S-2 integer tp
S-3 !$omp threadprivate(tp)
S-4 integer var
S-5 contains
S-6 subroutine work
S-7 !$omp parallel
S-8 ! do work here
S-9 !$omp task
S-10 tp = tp + 1
S-11 ! do work here
S-12 !$omp task
S-13 ! do work here but don't modify tp
S-14 !$omp end task
S-15 var = tp ! value does not change after write above
S-16 !$omp end task
S-17 !$omp end parallel
S-18 end subroutine
S-19 end module

```

2       The following two examples demonstrate how the scheduling rules illustrated in Section 2.11.3 of  
 3       the OpenMP 4.0 specification affect the usage of locks and critical sections in tasks. If a lock is  
 4       held across a task scheduling point, no attempt should be made to acquire the same lock in any code  
 5       that may be interleaved. Otherwise, a deadlock is possible.

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

9       *Example tasking.9c*

```

S-1 void work()
S-2 {
S-3 #pragma omp task
S-4 { //Task 1
S-5 #pragma omp task
S-6 { //Task 2
S-7 #pragma omp critical //Critical region 1
S-8 /*do work here */
S-9 }
S-10 #pragma omp critical //Critical Region 2
S-11 {

```

```

S-12 //Capture data for the following task
S-13 #pragma omp task
S-14 { /* do work here */ } //Task 3
S-15 }
S-16 }
S-17 }
```

C / C++

Fortran

1

*Example tasking.9f*

```

S-1 module example
S-2 contains
S-3 subroutine work
S-4 !$omp task
S-5 ! Task 1
S-6 !$omp task
S-7 ! Task 2
S-8 !$omp critical
S-9 ! Critical region 1
S-10 ! do work here
S-11 !$omp end critical
S-12 !$omp end task
S-13 !$omp critical
S-14 ! Critical region 2
S-15 ! Capture data for the following task
S-16 !$omp task
S-17 !Task 3
S-18 ! do work here
S-19 !$omp end task
S-20 !$omp end critical
S-21 !$omp end task
S-22 end subroutine
S-23 end module
```

Fortran

2  
3  
4

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.

1      Example tasking.10c

```

S-1 #include <omp.h>
S-2 void work() {
S-3 omp_lock_t lock;
S-4 omp_init_lock(&lock);
S-5 #pragma omp parallel
S-6 {
S-7 int i;
S-8 #pragma omp for
S-9 for (i = 0; i < 100; i++) {
S-10 #pragma omp task
S-11 {
S-12 // lock is shared by default in the task
S-13 omp_set_lock(&lock);
S-14 // Capture data for the following task
S-15 #pragma omp task
S-16 // Task Scheduling Point 1
S-17 { /* do work here */ }
S-18 omp_unset_lock(&lock);
S-19 }
S-20 }
S-21 }
S-22 omp_destroy_lock(&lock);
S-23 }
```

2      Example tasking.10f

```

S-1 module example
S-2 include 'omp_lib.h'
S-3 integer (kind=omp_lock_kind) lock
S-4 integer i
S-5 contains
S-6 subroutine work
S-7 call omp_init_lock(lock)
S-8 !$omp parallel
S-9 !$omp do
S-10 do i=1,100
S-11 !$omp task
S-12 ! Outer task
S-13 call omp_set_lock(lock) ! lock is shared by
S-14 ! default in the task
S-15 ! Capture data for the following task
S-16 !$omp task ! Task Scheduling Point 1
```

```

S-17 ! do work here
S-18 !$omp end task
S-19 call omp_unset_lock(lock)
S-20 !$omp end task
S-21 end do
S-22 !$omp end parallel
S-23 call omp_destroy_lock(lock)
S-24 end subroutine
S-25 end module

```

Fortran

1 The following examples illustrate the use of the **mergeable** clause in the **task** construct. In this  
 2 first example, the **task** construct has been annotated with the **mergeable** clause. The addition  
 3 of this clause allows the implementation to reuse the data environment (including the ICVs) of the  
 4 parent task for the task inside **foo** if the task is included or undeferred. Thus, the result of the  
 5 execution may differ depending on whether the task is merged or not. Therefore the mergeable  
 6 clause needs to be used with caution. In this example, the use of the mergeable clause is safe. As **x**  
 7 is a shared variable the outcome does not depend on whether or not the task is merged (that is, the  
 8 task will always increment the same variable and will always compute the same value for **x**).

C / C++

9 *Example tasking.11c*

```

S-1 #include <stdio.h>
S-2 void foo ()
S-3 {
S-4 int x = 2;
S-5 #pragma omp task shared(x) mergeable
S-6 {
S-7 x++;
S-8 }
S-9 #pragma omp taskwait
S-10 printf("%d\n",x); // prints 3
S-11 }

```

C / C++

Forran

1       *Example tasking.11f*

```
S-1 subroutine foo()
S-2 integer :: x
S-3 x = 2
S-4 !$omp task shared(x) mergeable
S-5 x = x + 1
S-6 !$omp end task
S-7 !$omp taskwait
S-8 print *, x ! prints 3
S-9 end subroutine
```

Forran

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

C / C++

7       *Example tasking.12c*

```
S-1 #include <stdio.h>
S-2 void foo ()
S-3 {
S-4 int x = 2;
S-5 #pragma omp task mergeable
S-6 {
S-7 x++;
S-8 }
S-9 #pragma omp taskwait
S-10 printf("%d\n",x); // prints 2 or 3
S-11 }
```

C / C++

## Fortran

1      Example tasking.12f

```
S-1 subroutine foo()
S-2 integer :: x
S-3 x = 2
S-4 !$omp task mergeable
S-5 x = x + 1
S-6 !$omp end task
S-7 !$omp taskwait
S-8 print *, x ! prints 2 or 3
S-9 end subroutine
```

## Fortran

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

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

## C / C++

14     Example tasking.13c

```
S-1 #include <string.h>
S-2 #include <omp.h>
S-3 #define LIMIT 3 /* arbitrary limit on recursion depth */
S-4 void check_solution(char *);
S-5 void bin_search (int pos, int n, char *state)
S-6 {
S-7 if (pos == n) {
S-8 check_solution(state);
S-9 return;
S-10 }
S-11 #pragma omp task final(pos > LIMIT) mergeable
S-12 {
S-13 char new_state[n];
S-14 if (!omp_in_final()) {
S-15 memcpy(new_state, state, pos);
```

```

S-16 state = new_state;
S-17 }
S-18 state[pos] = 0;
S-19 bin_search(pos+1, n, state);
S-20 }
S-21 #pragma omp task final(pos > LIMIT) mergeable
S-22 {
S-23 char new_state[n];
S-24 if (! omp_in_final()) {
S-25 memcpy(new_state, state, pos);
S-26 state = new_state;
S-27 }
S-28 state[pos] = 1;
S-29 bin_search(pos+1, n, state);
S-30 }
S-31 #pragma omp taskwait
S-32 }
```

C / C++

Fortran

### *Example tasking.13f*

```

S-1 recursive subroutine bin_search(pos, n, state)
S-2 use omp_lib
S-3 integer :: pos, n
S-4 character, pointer :: state(:)
S-5 character, target, dimension(n) :: new_state1, new_state2
S-6 integer, parameter :: LIMIT = 3
S-7 if (pos .eq. n) then
S-8 call check_solution(state)
S-9 return
S-10 endif
S-11 !$omp task final(pos > LIMIT) mergeable
S-12 if (.not. omp_in_final()) then
S-13 new_state1(1:pos) = state(1:pos)
S-14 state => new_state1
S-15 endif
S-16 state(pos+1) = 'z'
S-17 call bin_search(pos+1, n, state)
S-18 !$omp end task
S-19 !$omp task final(pos > LIMIT) mergeable
S-20 if (.not. omp_in_final()) then
S-21 new_state2(1:pos) = state(1:pos)
S-22 state => new_state2
S-23 endif
S-24 state(pos+1) = 'y'
S-25 call bin_search(pos+1, n, state)
```

```
S-26 !$omp end task
S-27 !$omp taskwait
S-28 end subroutine
```

Fortran

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

C / C++

7 Example tasking.14c

```
S-1 void foo ()
S-2 {
S-3 int i;
S-4 #pragma omp task if(0) // This task is undefined
S-5 {
S-6 #pragma omp task // This task is a regular task
S-7 for (i = 0; i < 3; i++) {
S-8 #pragma omp task // This task is a regular task
S-9 bar();
S-10 }
S-11 }
S-12 #pragma omp task final(1) // This task is a regular task
S-13 {
S-14 #pragma omp task // This task is included
S-15 for (i = 0; i < 3; i++) {
S-16 #pragma omp task // This task is also included
S-17 bar();
S-18 }
S-19 }
S-20 }
```

C / C++

1      *Example tasking.14f*

```
S-1 subroutine foo()
S-2 integer i
S-3 !$omp task if(.FALSE.) ! This task is undeferred
S-4 !$omp task ! This task is a regular task
S-5 do i = 1, 3
S-6 !$omp task ! This task is a regular task
S-7 call bar()
S-8 !$omp end task
S-9 enddo
S-10 !$omp end task
S-11 !$omp end task
S-12 !$omp task final(.TRUE.) ! This task is a regular task
S-13 !$omp task ! This task is included
S-14 do i = 1, 3
S-15 !$omp task ! This task is also included
S-16 call bar()
S-17 !$omp end task
S-18 enddo
S-19 !$omp end task
S-20 !$omp end task
S-21 end subroutine
```

1 CHAPTER 17

2

## Task Dependencies

---

3 17.1 Flow Dependence

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

C / C++

6 Example task\_dep.1c

```
S-1 #include <stdio.h>
S-2 int main()
S-3 {
S-4 int x = 1;
S-5 #pragma omp parallel
S-6 #pragma omp single
S-7 {
S-8 #pragma omp task shared(x) depend(out: x)
S-9 x = 2;
S-10 #pragma omp task shared(x) depend(in: x)
S-11 printf("x = %d\n", x);
S-12 }
S-13 return 0;
S-14 }
```

C / C++

## Fortran

1       *Example task\_dep.f90*

```
S-1 program example
S-2 integer :: x
S-3 x = 1
S-4 !$omp parallel
S-5 !$omp single
S-6 !$omp task shared(x) depend(out: x)
S-7 x = 2
S-8 !$omp end task
S-9 !$omp task shared(x) depend(in: x)
S-10 print*, "x = ", x
S-11 !$omp end task
S-12 !$omp end single
S-13 !$omp end parallel
S-14 end program
```

## Fortran

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

## 5   17.2 Anti-dependence

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

## C / C++

8       *Example task\_dep.2c*

```
S-1 #include <stdio.h>
S-2 int main()
S-3 {
S-4 int x = 1;
S-5 #pragma omp parallel
S-6 #pragma omp single
S-7 {
S-8 #pragma omp task shared(x) depend(in: x)
S-9 printf("x = %d\n", x);
S-10 #pragma omp task shared(x) depend(out: x)
S-11 x = 2;
```

```
S-12 }
S-13 return 0;
S-14 }
```

C / C++

Fortran

1      Example task\_dep.2f

```
S-1 program example
S-2 integer :: x
S-3 x = 1
S-4 !$omp parallel
S-5 !$omp single
S-6 !$omp task shared(x) depend(in: x)
S-7 print*, "x = ", x
S-8 !$omp end task
S-9 !$omp task shared(x) depend(out: x)
S-10 x = 2
S-11 !$omp end task
S-12 !$omp end single
S-13 !$omp end parallel
S-14 end program
```

Fortran

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

## 5    17.3 Output Dependence

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

---

C / C++

1      Example task\_dep.3c

```
S-1 #include <stdio.h>
S-2 int main()
S-3 {
S-4 int x;
S-5 #pragma omp parallel
S-6 #pragma omp single
S-7 {
S-8 #pragma omp task shared(x) depend(out: x)
S-9 x = 1;
S-10 #pragma omp task shared(x) depend(out: x)
S-11 x = 2;
S-12 #pragma omp taskwait
S-13 printf("x = %d\n", x);
S-14 }
S-15 return 0;
S-16 }
```

---

C / C++

---

Fortran

2      Example task\_dep.3f

```
S-1 program example
S-2 integer :: x
S-3 !$omp parallel
S-4 !$omp single
S-5 !$omp task shared(x) depend(out: x)
S-6 x = 1
S-7 !$omp end task
S-8 !$omp task shared(x) depend(out: x)
S-9 x = 2
S-10 !$omp end task
S-11 !$omp taskwait
S-12 print*, "x = ", x
S-13 !$omp end single
S-14 !$omp end parallel
S-15 end program
```

---

Fortran

3      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

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

C / C++

4 Example task\_dep.4c

```
S-1 #include <stdio.h>
S-2 int main()
S-3 {
S-4 int x = 1;
S-5 #pragma omp parallel
S-6 #pragma omp single
S-7 {
S-8 #pragma omp task shared(x) depend(out: x)
S-9 x = 2;
S-10 #pragma omp task shared(x) depend(in: x)
S-11 printf("x + 1 = %d. ", x+1);
S-12 #pragma omp task shared(x) depend(in: x)
S-13 printf("x + 2 = %d\n", x+2);
S-14 }
S-15 return 0;
S-16 }
```

C / C++

Fortran

5 Example task\_dep.4f

```
S-1 program example
S-2 integer :: x
S-3 x = 1
S-4 !$omp parallel
S-5 !$omp single
S-6 !$omp task shared(x) depend(out: x)
S-7 x = 2
S-8 !$omp end task
S-9 !$omp task shared(x) depend(in: x)
S-10 print*, "x + 1 = ", x+1, "."
S-11 !$omp end task
S-12 !$omp task shared(x) depend(in: x)
S-13 print*, "x + 2 = ", x+2, "."
S-14 !$omp end task
S-15 !$omp end single
S-16 !$omp end parallel
S-17 end program
```

---

**Fortran**

---

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

## 6 17.5 Matrix multiplication

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

---

**C / C++**

---

9 *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
S-3 C[N][N])
S-4 {
S-5 int i, j, k, ii, jj, kk;
S-6 for (i = 0; i < N; i+=BS) {
S-7 for (j = 0; j < N; j+=BS) {
S-8 for (k = 0; k < N; k+=BS) {
S-9 // Note 1: i, j, k, A, B, C are firstprivate by default
S-10 // Note 2: A, B and C are just pointers
S-11 #pragma omp task private(ii, jj, kk) \
S-12 depend (in: A[i:BS][k:BS], B[k:BS][j:BS]) \
S-13 depend (inout: C[i:BS][j:BS])
S-14 for (ii = i; ii < i+BS; ii++)
S-15 for (jj = j; jj < j+BS; jj++)
S-16 for (kk = k; kk < k+BS; kk++)
S-17 C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
S-18 }
S-19 }
S-20 }
S-21 }
```

---

**C / C++**

---

Forran

1

*Example task\_dep.f*

```
S-1 ! Assume BS divides N perfectly
S-2 subroutine matmul_depend (N, BS, A, B, C)
S-3 implicit none
S-4 integer :: N, BS, BM
S-5 real, dimension(N, N) :: A, B, C
S-6 integer :: i, j, k, ii, jj, kk
S-7 BM = BS - 1
S-8 do i = 1, N, BS
S-9 do j = 1, N, BS
S-10 do k = 1, N, BS
S-11 !$omp task shared(A,B,C) private(ii,jj,kk) & ! I,J,K are firstprivate by default
S-12 !$omp depend (in: A(i:i+BM, k:k+BM), B(k:k+BM, j:j+BM)) &
S-13 !$omp depend (inout: C(i:i+BM, j:j+BM))
S-14 do ii = i, i+BM
S-15 do jj = j, j+BM
S-16 do kk = k, k+BM
S-17 C(jj,ii) = C(jj,ii) + A(kk,ii) * B(jj,kk)
S-18 end do
S-19 end do
S-20 end do
S-21 !$omp end task
S-22 end do
S-23 end do
S-24 end do
S-25 end subroutine
```

Forran

## 1 CHAPTER 18

# 2 The taskgroup Construct

---

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

4 Initially, one task (the task executing the **start\_background\_work()** call) is created in the  
5 **parallel** region, and later a parallel tree traversal is started (the task executing the root of the  
6 recursive **compute\_tree()** calls). While synchronizing tasks at the end of each tree traversal,  
7 using the **taskgroup** construct ensures that the formerly started background task does not  
8 participate in the synchronization, and is left free to execute in parallel. This is opposed to the  
9 behaviour of the **taskwait** construct, which would include the background tasks in the  
10 synchronization.

C / C++

11 Example taskgroup.1c

```
S-1 extern void start_background_work(void);
S-2 extern void check_step(void);
S-3 extern void print_results(void);
S-4 struct tree_node
S-5 {
S-6 struct tree_node *left;
S-7 struct tree_node *right;
S-8 };
S-9 typedef struct tree_node* tree_type;
S-10 extern void init_tree(tree_type);
S-11 #define max_steps 100
S-12 void compute_something(tree_type tree)
S-13 {
S-14 // some computation
S-15 }
S-16 void compute_tree(tree_type tree)
S-17 {
S-18 if (tree->left)
S-19 {
```

```

S-20 #pragma omp task
S-21 compute_tree(tree->left);
S-22 }
S-23 if (tree->right)
S-24 {
S-25 #pragma omp task
S-26 compute_tree(tree->right);
S-27 }
S-28 #pragma omp task
S-29 compute_something(tree);
S-30 }
S-31 int main()
S-32 {
S-33 int i;
S-34 tree_type tree;
S-35 init_tree(tree);
S-36 #pragma omp parallel
S-37 #pragma omp single
S-38 {
S-39 #pragma omp task
S-40 start_background_work();
S-41 for (i = 0; i < max_steps; i++)
S-42 {
S-43 #pragma omp taskgroup
S-44 {
S-45 #pragma omp task
S-46 compute_tree(tree);
S-47 } // wait on tree traversal in this step
S-48 check_step();
S-49 }
S-50 } // only now is background work required to be complete
S-51 print_results();
S-52 return 0;
S-53 }
```

C / C++

Fortran

1

*Example taskgroup.If*

```

S-1 module tree_type_mod
S-2 integer, parameter :: max_steps=100
S-3 type tree_type
S-4 type(tree_type), pointer :: left, right
S-5 end type
S-6 contains
S-7 subroutine compute_something(tree)
S-8 type(tree_type), pointer :: tree
```

```

S-9 ! some computation
S-10 end subroutine
S-11 recursive subroutine compute_tree(tree)
S-12 type(tree_type), pointer :: tree
S-13 if (associated(tree%left)) then
S-14 !$omp task
S-15 call compute_tree(tree%left)
S-16 !$omp end task
S-17 endif
S-18 if (associated(tree%right)) then
S-19 !$omp task
S-20 call compute_tree(tree%right)
S-21 !$omp end task
S-22 endif
S-23 !$omp task
S-24 call compute_something(tree)
S-25 !$omp end task
S-26 end subroutine
S-27 end module
S-28 program main
S-29 use tree_type_mod
S-30 type(tree_type), pointer :: tree
S-31 call init_tree(tree);
S-32 !$omp parallel
S-33 !$omp single
S-34 !$omp task
S-35 call start_background_work()
S-36 !$omp end task
S-37 do i=1, max_steps
S-38 !$omp taskgroup
S-39 !$omp task
S-40 call compute_tree(tree)
S-41 !$omp end task
S-42 !$omp end taskgroup ! wait on tree traversal in this step
S-43 call check_step()
S-44 enddo
S-45 !$omp end single
S-46 !$omp end parallel ! only now is background work required to be complete
S-47 call print_results()
S-48 end program

```

Fortran

1 CHAPTER 19

2 **The `taskyield` Construct**

---

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

C / C++

7 Example `taskyield.1c`

```
S-1 #include <omp.h>
S-2
S-3 void something_useful (void);
S-4 void something_critical (void);
S-5 void foo (omp_lock_t * lock, int n)
S-6 {
S-7 int i;
S-8
S-9 for (i = 0; i < n; i++)
S-10 #pragma omp task
S-11 {
S-12 something_useful();
S-13 while (!omp_test_lock(lock)) {
S-14 #pragma omp taskyield
S-15 }
S-16 something_critical();
S-17 omp_unset_lock(lock);
S-18 }
S-19 }
```

C / C++

Forran

1      Example taskyield. If

```
S-1 subroutine foo (lock, n)
S-2 use omp_lib
S-3 integer (kind=omp_lock_kind) :: lock
S-4 integer n
S-5 integer i
S-6
S-7 do i = 1, n
S-8 !$omp task
S-9 call something_useful()
S-10 do while (.not. omp_test_lock(lock))
S-11 !$omp taskyield
S-12 end do
S-13 call something_critical()
S-14 call omp_unset_lock(lock)
S-15 !$omp end task
S-16 end do
S-17
S-18 end subroutine
```

Forran

1 CHAPTER 20

2 

# The workshare Construct

---

Fortran

3 The following are examples of the **workshare** construct.

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

7 *Example workshare.1f*

```
S-1 SUBROUTINE WSHARE1(AA, BB, CC, DD, EE, FF, N)
S-2 INTEGER N
S-3 REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N), EE(N,N), FF(N,N)
S-4
S-5 !$OMP PARALLEL
S-6 !$OMP WORKSHARE
S-7 AA = BB
S-8 CC = DD
S-9 EE = FF
S-10 !$OMP END WORKSHARE
S-11 !$OMP END PARALLEL
S-12
S-13 END SUBROUTINE WSHARE1
```

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

11 *Example workshare.2f*

## Fortran (cont.)

```
S-1 SUBROUTINE WSHARE2(AA, BB, CC, DD, EE, FF, 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
S-6 !$OMP PARALLEL
S-7 !$OMP WORKSHARE
S-8 AA = BB
S-9 CC = DD
S-10 !$OMP END WORKSHARE NOWAIT
S-11 !$OMP WORKSHARE
S-12 EE = FF
S-13 !$OMP END WORKSHARE
S-14 !$OMP END PARALLEL
S-15 END SUBROUTINE WSHARE2
```

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

*Example workshare.3f*

```
S-1 SUBROUTINE WSHARE3(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 REAL R
S-5 R=0
S-6 !$OMP PARALLEL
S-7 !$OMP WORKSHARE
S-8 AA = BB
S-9 !$OMP ATOMIC UPDATE
S-10 R = R + SUM(AA)
S-11 CC = DD
S-12 !$OMP END WORKSHARE
S-13 !$OMP END PARALLEL
S-14 END SUBROUTINE WSHARE3
```

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

Each task gets worked on in order by the threads:

```
AA = BB then
CC = DD then
EE .ne. 0 then
```

## Fortran (cont.)

```
1 FF = 1 / EE then
2 GG = HH
```

3      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
```

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

6      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
```

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

1       *Example workshare.6f*

```
S-1 SUBROUTINE WSHARE6_WRONG(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 PRI
S-6
S-7 !$OMP PARALLEL PRIVATE(PRI)
S-8 !$OMP WORKSHARE
S-9 AA = BB
S-10 PRI = 1
S-11 CC = DD * PRI
S-12 !$OMP END WORKSHARE
S-13 !$OMP END PARALLEL
S-14
S-15 END SUBROUTINE WSHARE6_WRONG
```

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

5       *Example workshare.7f*

```
S-1 SUBROUTINE WSHARE7(AA, BB, CC, N)
S-2 INTEGER N
S-3 REAL AA(N), BB(N), CC(N)
S-4
S-5 !$OMP PARALLEL
S-6 !$OMP WORKSHARE
S-7 AA(1:50) = BB(11:60)
S-8 CC(11:20) = AA(1:10)
S-9 !$OMP END WORKSHARE
S-10 !$OMP END PARALLEL
S-11
S-12 END SUBROUTINE WSHARE7
```

Fortran

1 CHAPTER 21

2 **The master Construct**

---

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

C / C++

6 Example master.1c

```
S-1 #include <stdio.h>
S-2
S-3 extern float average(float,float,float);
S-4
S-5 void master_example(float* x, float* xold, int n, float tol)
S-6 {
S-7 int c, i, toobig;
S-8 float error, y;
S-9 c = 0;
S-10 #pragma omp parallel
S-11 {
S-12 do{
S-13 #pragma omp for private(i)
S-14 for(i = 1; i < n-1; ++i){
S-15 xold[i] = x[i];
S-16 }
S-17 #pragma omp single
S-18 {
S-19 toobig = 0;
S-20 }
S-21 #pragma omp for private(i,y,error) reduction(+:toobig)
S-22 for(i = 1; i < n-1; ++i){
S-23 y = x[i];
S-24 x[i] = average(xold[i-1], x[i], xold[i+1]);
S-25 error = y - x[i];
```

```

S-26 if(error > tol || error < -tol) ++toobig;
S-27 }
S-28 #pragma omp master
S-29 {
S-30 ++c;
S-31 printf("iteration %d, toobig=%d\n", c, toobig);
S-32 }
S-33 }while(toobig > 0);
S-34 }
S-35 }
```

C / C++

Fortran

1

*Example master. If*

```

S-1 SUBROUTINE MASTER_EXAMPLE(X, XOLD, N, TOL)
S-2 REAL X(*), XOLD(*), TOL
S-3 INTEGER N
S-4 INTEGER C, I, TOOBIG
S-5 REAL ERROR, Y, AVERAGE
S-6 EXTERNAL AVERAGE
S-7 C = 0
S-8 TOOBIG = 1
S-9 !$OMP PARALLEL
S-10 DO WHILE(TOOBIG > 0)
S-11 !$OMP DO PRIVATE(I)
S-12 DO I = 2, N-1
S-13 XOLD(I) = X(I)
S-14 ENDDO
S-15 !$OMP SINGLE
S-16 TOOBIG = 0
S-17 !$OMP END SINGLE
S-18 !$OMP DO PRIVATE(I,Y,ERROR), REDUCTION(+:TOOBIG)
S-19 DO I = 2, N-1
S-20 Y = X(I)
S-21 X(I) = AVERAGE(XOLD(I-1), X(I), XOLD(I+1))
S-22 ERROR = Y-X(I)
S-23 IF(ERROR > TOL .OR. ERROR < -TOL) TOOBIG = TOOBIG+1
S-24 ENDDO
S-25 !$OMP MASTER
S-26 C = C + 1
S-27 PRINT *, 'Iteration ', C, 'TOOBIG=', TOOBIG
S-28 !$OMP END MASTER
S-29 ENDDO
S-30 !$OMP END PARALLEL
S-31 END SUBROUTINE MASTER_EXAMPLE
```

Fortran

1 CHAPTER 22

2 **The critical Construct**

---

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

C / C++

8 Example critical.1c

```
S-1 int dequeue(float *a);
S-2 void work(int i, float *a);
S-3
S-4 void critical_example(float *x, float *y)
S-5 {
S-6 int ix_next, iy_next;
S-7
S-8 #pragma omp parallel shared(x, y) private(ix_next, iy_next)
S-9 {
S-10 #pragma omp critical (xaxis)
S-11 ix_next = dequeue(x);
S-12 work(ix_next, x);
S-13
S-14 #pragma omp critical (yaxis)
S-15 iy_next = dequeue(y);
S-16 work(iy_next, y);
S-17 }
S-18
S-19 }
```

C / C++

Forran

1      *Example critical.If*

```
S-1 SUBROUTINE CRITICAL_EXAMPLE(X, Y)
S-2
S-3 REAL X(*), Y(*)
S-4 INTEGER IX_NEXT, IY_NEXT
S-5
S-6 !$OMP PARALLEL SHARED(X, Y) PRIVATE(IX_NEXT, IY_NEXT)
S-7
S-8 !$OMP CRITICAL(XAXIS)
S-9 CALL DEQUEUE(IX_NEXT, X)
S-10 !$OMP END CRITICAL(XAXIS)
S-11 CALL WORK(IX_NEXT, X)
S-12
S-13 !$OMP CRITICAL(YAXIS)
S-14 CALL DEQUEUE(IY_NEXT, Y)
S-15 !$OMP END CRITICAL(YAXIS)
S-16 CALL WORK(IY_NEXT, Y)
S-17
S-18 !$OMP END PARALLEL
S-19
S-20 END SUBROUTINE CRITICAL_EXAMPLE
```

Forran

1 CHAPTER 23

2

## Worksharing Constructs Inside 3 a **critical** Construct

---

4 The following example demonstrates using a worksharing construct inside a **critical** construct.  
5 This example is conforming because the worksharing **single** region is not closely nested inside  
6 the **critical** region. A single thread executes the one and only section in the **sections**  
7 region, and executes the **critical** region. The same thread encounters the nested **parallel**  
8 region, creates a new team of threads, and becomes the master of the new team. One of the threads  
9 in the new team enters the **single** region and increments **i** by 1. At the end of this example **i** is  
10 equal to 2.

C / C++

11 Example *worksharing\_critical.1c*

```
S-1 void critical_work()
S-2 {
S-3 int i = 1;
S-4 #pragma omp parallel sections
S-5 {
S-6 #pragma omp section
S-7 {
S-8 #pragma omp critical (name)
S-9 {
S-10 #pragma omp parallel
S-11 {
S-12 #pragma omp single
S-13 {
S-14 i++;
S-15 }
S-16 }
S-17 }
S-18 }
S-19 }
S-20 }
```

1

*Example worksharing\_critical.If*

```
S-1 SUBROUTINE CRITICAL_WORK()
S-2
S-3 INTEGER I
S-4 I = 1
S-5
S-6 !$OMP PARALLEL SECTIONS
S-7 !$OMP SECTION
S-8 !$OMP CRITICAL (NAME)
S-9 !$OMP PARALLEL
S-10 !$OMP SINGLE
S-11 I = I + 1
S-12 !$OMP END SINGLE
S-13 !$OMP END PARALLEL
S-14 !$OMP END CRITICAL (NAME)
S-15 !$OMP END PARALLEL SECTIONS
S-16 END SUBROUTINE CRITICAL_WORK
```

C / C++

Fortran

Fortran

1 CHAPTER 24

2 **Binding of barrier Regions**

---

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

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

8 The call from the main program to *sub3* is conforming because the **barrier** region binds to the  
9 implicit inactive **parallel** region enclosing the sequential part. Also note that the **barrier**  
10 region in *sub3* when called from *sub2* only synchronizes the team of threads in the enclosing  
11 **parallel** region and not all the threads created in *sub1*.

C / C++

12 Example *barrier\_regions.1c*

```
S-1 void work(int n) {}
S-2
S-3 void sub3(int n)
S-4 {
S-5 work(n);
S-6 #pragma omp barrier
S-7 work(n);
S-8 }
S-9
S-10 void sub2(int k)
S-11 {
S-12 #pragma omp parallel shared(k)
S-13 sub3(k);
S-14 }
S-15
S-16 void sub1(int n)
S-17 {
```

```

S-18 int i;
S-19 #pragma omp parallel private(i) shared(n)
S-20 {
S-21 #pragma omp for
S-22 for (i=0; i<n; i++)
S-23 sub2(i);
S-24 }
S-25 }
S-26
S-27 int main()
S-28 {
S-29 sub1(2);
S-30 sub2(2);
S-31 sub3(2);
S-32 return 0;
S-33 }
```

C / C++

Fortran

1       *Example barrier\_regions.If*

```

S-1 SUBROUTINE WORK(N)
S-2 INTEGER N
S-3 END SUBROUTINE WORK
S-4
S-5 SUBROUTINE SUB3(N)
S-6 INTEGER N
S-7 CALL WORK(N)
S-8 !$OMP BARRIER
S-9 CALL WORK(N)
S-10 END SUBROUTINE SUB3
S-11
S-12 SUBROUTINE SUB2(K)
S-13 INTEGER K
S-14 !$OMP PARALLEL SHARED(K)
S-15 CALL SUB3(K)
S-16 !$OMP END PARALLEL
S-17 END SUBROUTINE SUB2
S-18
S-19
S-20 SUBROUTINE SUB1(N)
S-21 INTEGER N
S-22 INTEGER I
S-23 !$OMP PARALLEL PRIVATE(I) SHARED(N)
S-24 !$OMP DO
S-25 DO I = 1, N
S-26 CALL SUB2(I)
```

```
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
```

Fortran

1 CHAPTER 25

2 

## The atomic Construct

---

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

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

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

C / C++

10 Example atomic.1c

```
S-1 float work1(int i)
S-2 {
S-3 return 1.0 * i;
S-4 }
S-5
S-6 float work2(int i)
S-7 {
S-8 return 2.0 * i;
S-9 }
S-10
S-11 void atomic_example(float *x, float *y, int *index, int n)
S-12 {
S-13 int i;
S-14
S-15 #pragma omp parallel for shared(x, y, index, n)
S-16 for (i=0; i<n; i++) {
S-17 #pragma omp atomic update
S-18 x[index[i]] += work1(i);
S-19 y[i] += work2(i);
```

```

S-20 }
S-21 }
S-22
S-23 int main()
S-24 {
S-25 float x[1000];
S-26 float y[10000];
S-27 int index[10000];
S-28 int i;
S-29
S-30 for (i = 0; i < 10000; i++) {
S-31 index[i] = i % 1000;
S-32 y[i]=0.0;
S-33 }
S-34 for (i = 0; i < 1000; i++)
S-35 x[i] = 0.0;
S-36 atomic_example(x, y, index, 10000);
S-37 return 0;
S-38 }

```

C / C++  
Fortran

1

*Example atomic.If*

```

S-1 REAL FUNCTION WORK1(I)
S-2 INTEGER I
S-3 WORK1 = 1.0 * I
S-4 RETURN
S-5 END FUNCTION WORK1
S-6
S-7 REAL FUNCTION WORK2(I)
S-8 INTEGER I
S-9 WORK2 = 2.0 * I
S-10 RETURN
S-11 END FUNCTION WORK2
S-12
S-13 SUBROUTINE SUB(X, Y, INDEX, N)
S-14 REAL X(*), Y(*)
S-15 INTEGER INDEX(*), N
S-16
S-17 INTEGER I
S-18
S-19 !$OMP PARALLEL DO SHARED(X, Y, INDEX, N)
S-20 DO I=1,N
S-21 !$OMP ATOMIC UPDATE
S-22 X(INDEX(I)) = X(INDEX(I)) + WORK1(I)
S-23 Y(I) = Y(I) + WORK2(I)

```

```

S-24 ENDDO
S-25
S-26 END SUBROUTINE SUB
S-27
S-28 PROGRAM ATOMIC_EXAMPLE
S-29 REAL X(1000), Y(10000)
S-30 INTEGER INDEX(10000)
S-31 INTEGER I
S-32
S-33 DO I=1,10000
S-34 INDEX(I) = MOD(I, 1000) + 1
S-35 Y(I) = 0.0
S-36 ENDDO
S-37
S-38 DO I = 1,1000
S-39 X(I) = 0.0
S-40 ENDDO
S-41
S-42 CALL SUB(X, Y, INDEX, 10000)
S-43
S-44 END PROGRAM ATOMIC_EXAMPLE

```

Fortran

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

C / C++

*Example atomic.2c*

```

S-1 int atomic_read(const int *p)
S-2 {
S-3 int value;
S-4 /* Guarantee that the entire value of *p is read atomically. No part of
S-5 * *p can change during the read operation.
S-6 */
S-7 #pragma omp atomic read
S-8 value = *p;
S-9 return value;
S-10 }
S-11
S-12 void atomic_write(int *p, int value)
S-13 {
S-14 /* Guarantee that value is stored atomically into *p. No part of *p can

```

```

S-15 change
S-16 * until after the entire write operation is completed.
S-17 */
S-18 #pragma omp atomic write
S-19 *p = value;
S-20 }

```

C / C++

Fortran

1 Example atomic.2f

```

S-1 function atomic_read(p)
S-2 integer :: atomic_read
S-3 integer, intent(in) :: p
S-4 ! Guarantee that the entire value of p is read atomically. No part of
S-5 ! p can change during the read operation.
S-6
S-7 !$omp atomic read
S-8 atomic_read = p
S-9 return
S-10 end function atomic_read
S-11
S-12 subroutine atomic_write(p, value)
S-13 integer, intent(out) :: p
S-14 integer, intent(in) :: value
S-15 ! Guarantee that value is stored atomically into p. No part of p can change
S-16 ! until after the entire write operation is completed.
S-17 !$omp atomic write
S-18 p = value
S-19 end subroutine atomic_write

```

Fortran

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

1

*Example atomic.3c*

```

S-1 int fetch_and_add(int *p)
S-2 {
S-3 /* Atomically read the value of *p and then increment it. The previous value
S-4 is
S-5 * returned. This can be used to implement a simple lock as shown below.
S-6 */
S-7 int old;
S-8 #pragma omp atomic capture
S-9 { old = *p; (*p)++; }
S-10 return old;
S-11 }
S-12 /*
S-13 * Use fetch_and_add to implement a lock
S-14 */
S-15 struct locktype {
S-16 int ticketnumber;
S-17 int turn;
S-18 };
S-19 void do_locked_work(struct locktype *lock)
S-20 {
S-21 int atomic_read(const int *p);
S-22 void work();
S-23
S-24 // Obtain the lock
S-25 int myturn = fetch_and_add(&lock->ticketnumber);
S-26 while (atomic_read(&lock->turn) != myturn)
S-27 ;
S-28 // Do some work. The flush is needed to ensure visibility of
S-29 // variables not involved in atomic directives
S-30
S-31 #pragma omp flush
S-32 work();
S-33 #pragma omp flush
S-34 // Release the lock
S-35 fetch_and_add(&lock->turn);
S-36 }

```

1

*Example atomic.3f*

```

S-1 function fetch_and_add(p)
S-2 integer:: fetch_and_add
S-3 integer, intent(inout) :: p
S-4
S-5 ! Atomically read the value of p and then increment it. The previous value is
S-6 ! returned. This can be used to implement a simple lock as shown below.
S-7 !$omp atomic capture
S-8 fetch_and_add = p
S-9 p = p + 1
S-10 !$omp end atomic
S-11 end function fetch_and_add
S-12 module m
S-13 interface
S-14 function fetch_and_add(p)
S-15 integer :: fetch_and_add
S-16 integer, intent(inout) :: p
S-17 end function
S-18 function atomic_read(p)
S-19 integer :: atomic_read
S-20 integer, intent(in) :: p
S-21 end function
S-22 end interface
S-23 type locktype
S-24 integer ticketnumber
S-25 integer turn
S-26 end type
S-27 contains
S-28 subroutine do_locked_work(lock)
S-29 type(locktype), intent(inout) :: lock
S-30 integer myturn
S-31 integer junk
S-32 ! obtain the lock
S-33 myturn = fetch_and_add(lock%ticketnumber)
S-34 do while (atomic_read(lock%turn) .ne. myturn)
S-35 continue
S-36 enddo
S-37 ! Do some work. The flush is needed to ensure visibility of variables
S-38 ! not involved in atomic directives
S-39 !$omp flush
S-40 call work
S-41 !$omp flush
S-42 ! Release the lock
S-43 junk = fetch_and_add(lock%turn)

```

S-44  
S-45

```
end subroutine
end module
```

Fortran

## CHAPTER 26

# Restrictions on the `atomic` Construct

---

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

C / C++

*Example atomic\_restrict.1c*

```
S-1 void atomic_wrong ()
S-2 {
S-3 union {int n; float x;} u;
S-4
S-5 #pragma omp parallel
S-6 {
S-7 #pragma omp atomic update
S-8 u.n++;
S-9
S-10 #pragma omp atomic update
S-11 u.x += 1.0;
S-12
S-13 /* Incorrect because the atomic constructs reference the same location
S-14 through incompatible types */
S-15 }
S-16 }
```

C / C++

Fortran

1      Example atomic\_restrict.1f

```
S-1 SUBROUTINE ATOMIC_WRONG()
S-2 INTEGER:: I
S-3 REAL:: R
S-4 EQUIVALENCE (I,R)
S-5
S-6 !$OMP PARALLEL
S-7 !$OMP ATOMIC UPDATE
S-8 I = I + 1
S-9 !$OMP ATOMIC UPDATE
S-10 R = R + 1.0
S-11 ! incorrect because I and R reference the same location
S-12 ! but have different types
S-13 !$OMP END PARALLEL
S-14 END SUBROUTINE ATOMIC_WRONG
```

Fortran

C / C++

2      Example atomic\_restrict.2c

```
S-1 void atomic_wrong2 ()
S-2 {
S-3 int x;
S-4 int *i;
S-5 float *r;
S-6
S-7 i = &x;
S-8 r = (float *)&x;
S-9
S-10 #pragma omp parallel
S-11 {
S-12 #pragma omp atomic update
S-13 *i += 1;
S-14
S-15 #pragma omp atomic update
S-16 *r += 1.0;
S-17
S-18 /* Incorrect because the atomic constructs reference the same location
S-19 through incompatible types */
S-20
S-21 }
S-22 }
```

C / C++

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

3 *Example atomic\_restrict.2f*

```
S-1 SUBROUTINE SUB()
S-2 COMMON /BLK/ R
S-3 REAL R
S-4
S-5 !$OMP ATOMIC UPDATE
S-6 R = R + 1.0
S-7 END SUBROUTINE SUB
S-8
S-9 SUBROUTINE ATOMIC_WRONG2()
S-10 COMMON /BLK/ I
S-11 INTEGER I
S-12
S-13 !$OMP PARALLEL
S-14
S-15 !$OMP ATOMIC UPDATE
S-16 I = I + 1
S-17 CALL SUB()
S-18 !$OMP END PARALLEL
S-19 END SUBROUTINE ATOMIC_WRONG2
```

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

5 *Example atomic\_restrict.3f*

```
S-1 SUBROUTINE ATOMIC_WRONG3
S-2 INTEGER:: I
S-3 REAL:: R
S-4 EQUIVALENCE (I,R)
S-5
S-6 !$OMP PARALLEL
S-7 !$OMP ATOMIC UPDATE
S-8 I = I + 1
S-9 ! incorrect because I and R reference the same location
S-10 ! but have different types
S-11 !$OMP END PARALLEL
S-12
S-13 !$OMP PARALLEL
S-14 !$OMP ATOMIC UPDATE
S-15 R = R + 1.0
S-16 ! incorrect because I and R reference the same location
S-17 ! but have different types
S-18 !$OMP END PARALLEL
```

S-19  
S-20

END SUBROUTINE ATOMIC\_WRONG3

Fortran

1 CHAPTER 27

2

## The flush Construct without a 3 List

---

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

C / C++

6 Example *flush\_nolist.c*

```
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)
```

```

S-25 {
S-26 int i = 1, j, sum = 0;
S-27 *p = 1;
S-28 #pragma omp parallel reduction(+: sum) num_threads(10)
S-29 {
S-30 f1(&j);
S-31
S-32 /* i, n and sum were not flushed */
S-33 /* because they were not accessible in f1 */
S-34 /* j was flushed because it was accessible */
S-35 sum += j;
S-36
S-37 f2(&j);
S-38
S-39 /* i, n, and sum were not flushed */
S-40 /* because they were not accessible in f2 */
S-41 /* j was flushed because it was accessible */
S-42 sum += i + j + *p + n;
S-43 }
S-44 return sum;
S-45 }
S-46
S-47 int main()
S-48 {
S-49 int result = g(7);
S-50 return result;
S-51 }

```

C / C++

Fortran

*Example flush\_nolist.If*

```

S-1 SUBROUTINE F1(Q)
S-2 COMMON /DATA/ X, P
S-3 INTEGER, TARGET :: X
S-4 INTEGER, POINTER :: P
S-5 INTEGER Q
S-6
S-7 Q = 1
S-8 !$OMP FLUSH
S-9 ! X, P and Q are flushed
S-10 ! because they are shared and accessible
S-11 END SUBROUTINE F1
S-12
S-13 SUBROUTINE F2(Q)
S-14 COMMON /DATA/ X, P
S-15 INTEGER, TARGET :: X

```

```

S-16 INTEGER, POINTER :: P
S-17 INTEGER Q
S-18
S-19 !$OMP BARRIER
S-20 Q = 2
S-21 !$OMP BARRIER
S-22 ! a barrier implies a flush
S-23 ! X, P and Q are flushed
S-24 ! because they are shared and accessible
S-25 END SUBROUTINE F2
S-26
S-27 INTEGER FUNCTION G(N)
S-28 COMMON /DATA/ X, P
S-29 INTEGER, TARGET :: X
S-30 INTEGER, POINTER :: P
S-31 INTEGER N
S-32 INTEGER I, J, SUM
S-33
S-34 I = 1
S-35 SUM = 0
S-36 P = 1
S-37 !$OMP PARALLEL REDUCTION(+: SUM) NUM_THREADS(10)
S-38 CALL F1(J)
S-39 ! I, N and SUM were not flushed
S-40 ! because they were not accessible in F1
S-41 ! J was flushed because it was accessible
S-42 SUM = SUM + J
S-43
S-44 CALL F2(J)
S-45 ! I, N, and SUM were not flushed
S-46 ! because they were not accessible in f2
S-47 ! J was flushed because it was accessible
S-48 SUM = SUM + I + J + P + N
S-49 !$OMP END PARALLEL
S-50
S-51 G = SUM
S-52 END FUNCTION G
S-53
S-54 PROGRAM FLUSH_NOLIST
S-55 COMMON /DATA/ X, P
S-56 INTEGER, TARGET :: X
S-57 INTEGER, POINTER :: P
S-58 INTEGER RESULT, G
S-59
S-60 P => X
S-61 RESULT = G(7)
S-62 PRINT *, RESULT

```



1 CHAPTER 28

2

## Placement of flush, barrier, taskwait and taskyield Directives

3

---

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

C / C++

7 Example standalone.1c

```
S-1
S-2 void standalone_wrong()
S-3 {
S-4 int a = 1;
S-5
S-6 if (a != 0)
S-7 #pragma omp flush(a)
S-8 /* incorrect as flush cannot be immediate substatement
S-9 of if statement */
S-10
S-11 if (a != 0)
S-12 #pragma omp barrier
S-13 /* incorrect as barrier cannot be immediate substatement
S-14 of if statement */
S-15
S-16 if (a!=0)
S-17 #pragma omp taskyield
S-18 /* incorrect as taskyield cannot be immediate substatement of if statement
S-19 */
S-20
S-21 if (a != 0)
S-22 #pragma omp taskwait
S-23 /* incorrect as taskwait cannot be immediate substatement
```

```
S-24 of if statement */
S-25
S-26 }
```

C / C++

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

Fortran

4 Example standalone.f90

```
S-1 SUBROUTINE STANDALONE_WRONG()
S-2 INTEGER A
S-3 A = 1
S-4 ! the FLUSH directive must not be the action statement
S-5 ! in an IF statement
S-6 IF (A .NE. 0) !$OMP FLUSH(A)
S-7
S-8 ! the BARRIER directive must not be the action statement
S-9 ! in an IF statement
S-10 IF (A .NE. 0) !$OMP BARRIER
S-11
S-12 ! the TASKWAIT directive must not be the action statement
S-13 ! in an IF statement
S-14 IF (A .NE. 0) !$OMP TASKWAIT
S-15
S-16 ! the TASKYIELD directive must not be the action statement
S-17 ! in an IF statement
S-18 IF (A .NE. 0) !$OMP TASKYIELD
S-19
S-20 GOTO 100
S-21
S-22 ! the FLUSH directive must not be a labeled branch target
S-23 ! statement
S-24 100 !$OMP FLUSH(A)
S-25 GOTO 200
S-26
S-27 ! the BARRIER directive must not be a labeled branch target
S-28 ! statement
S-29 200 !$OMP BARRIER
S-30 GOTO 300
S-31
S-32 ! the TASKWAIT directive must not be a labeled branch target
S-33 ! statement
S-34 300 !$OMP TASKWAIT
S-35 GOTO 400
```

```
S-36 ! the TASKYIELD directive must not be a labeled branch target
S-37 ! statement
S-38 400 !$OMP TASKYIELD
S-40
S-41 END SUBROUTINE
```

Fortran

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

C / C++

3 Example standalone.2c

```
S-1 void standalone_ok()
S-2 {
S-3 int a = 1;
S-4
S-5 #pragma omp parallel
S-6 {
S-7 if (a != 0) {
S-8 #pragma omp flush(a)
S-9 }
S-10 if (a != 0) {
S-11 #pragma omp barrier
S-12 }
S-13 if (a != 0) {
S-14 #pragma omp taskwait
S-15 }
S-16 if (a != 0) {
S-17 #pragma omp taskyield
S-18 }
S-19 }
```

C / C++

4 The following example is conforming because the **flush**, **barrier**, **taskwait**, and  
5 **taskyield** directives are enclosed in an **if** construct or follow the labeled branch target.

1

*Example standalone.2f*

```
S-1 SUBROUTINE STANDALONE_OK()
S-2 INTEGER A
S-3 A = 1
S-4 IF (A .NE. 0) THEN
S-5 !$OMP FLUSH(A)
S-6 ENDIF
S-7 IF (A .NE. 0) THEN
S-8 !$OMP BARRIER
S-9 ENDIF
S-10 IF (A .NE. 0) THEN
S-11 !$OMP TASKWAIT
S-12 ENDIF
S-13 IF (A .NE. 0) THEN
S-14 !$OMP TASKYIELD
S-15 ENDIF
S-16 GOTO 100
S-17 100 CONTINUE
S-18 !$OMP FLUSH(A)
S-19 GOTO 200
S-20 200 CONTINUE
S-21 !$OMP BARRIER
S-22 GOTO 300
S-23 300 CONTINUE
S-24 !$OMP TASKWAIT
S-25 GOTO 400
S-26 400 CONTINUE
S-27 !$OMP TASKYIELD
S-28 END SUBROUTINE
```

1 CHAPTER 29

2

## 3 The ordered Clause and the ordered Construct

---

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

C / C++

6 Example *ordered.1c*

```
S-1 #include <stdio.h>
S-2
S-3 void work(int k)
S-4 {
S-5 #pragma omp ordered
S-6 printf("%d\n", k);
S-7 }
S-8
S-9 void ordered_example(int lb, int ub, int stride)
S-10 {
S-11 int i;
S-12
S-13 #pragma omp parallel for ordered schedule(dynamic)
S-14 for (i=lb; i<ub; i+=stride)
S-15 work(i);
S-16 }
S-17
S-18 int main()
S-19 {
S-20 ordered_example(0, 100, 5);
S-21 return 0;
S-22 }
```

C / C++

## Fortran

1       *Example ordered.1f*

```
S-1 SUBROUTINE WORK(K)
S-2 INTEGER k
S-3
S-4 !$OMP ORDERED
S-5 WRITE(*,*) K
S-6 !$OMP END ORDERED
S-7
S-8 END SUBROUTINE WORK
S-9
S-10 SUBROUTINE SUB(LB, UB, STRIDE)
S-11 INTEGER LB, UB, STRIDE
S-12 INTEGER I
S-13
S-14 !$OMP PARALLEL DO ORDERED SCHEDULE(DYNAMIC)
S-15 DO I=LB,UB,STRIDE
S-16 CALL WORK(I)
S-17 END DO
S-18 !$OMP END PARALLEL DO
S-19
S-20 END SUBROUTINE SUB
S-21
S-22 PROGRAM ORDERED_EXAMPLE
S-23 CALL SUB(1,100,5)
S-24 END PROGRAM ORDERED_EXAMPLE
```

## Fortran

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

## C / C++

5       *Example ordered.2c*

```
S-1 void work(int i) {}
S-2
S-3 void ordered_wrong(int n)
S-4 {
S-5 int i;
S-6 #pragma omp for ordered
S-7 for (i=0; i<n; i++) {
S-8 /* incorrect because an iteration may not execute more than one
S-9 ordered region */
S-10 #pragma omp ordered
S-11 work(i);
```

```
S-12 #pragma omp ordered
S-13 work(i+1);
S-14 }
S-15 }
```

C / C++  
Fortran

1 Example ordered.2f

```
S-1 SUBROUTINE WORK(I)
S-2 INTEGER I
S-3 END SUBROUTINE WORK
S-4
S-5 SUBROUTINE ORDERED_WRONG(N)
S-6 INTEGER N
S-7
S-8 INTEGER I
S-9 !$OMP DO ORDERED
S-10 DO I = 1, N
S-11 ! incorrect because an iteration may not execute more than one
S-12 ! ordered region
S-13 !$OMP ORDERED
S-14 CALL WORK(I)
S-15 !$OMP END ORDERED
S-16
S-17 !$OMP ORDERED
S-18 CALL WORK(I+1)
S-19 !$OMP END ORDERED
S-20 END DO
S-21 END SUBROUTINE ORDERED_WRONG
```

Fortran

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

## 1 Example ordered.3c

```

S-1 void work(int i) {}
S-2 void ordered_good(int n)
S-3 {
S-4 int i;
S-5 #pragma omp for ordered
S-6 for (i=0; i<n; i++) {
S-7 if (i <= 10) {
S-8 #pragma omp ordered
S-9 work(i);
S-10 }
S-11 if (i > 10) {
S-12 #pragma omp ordered
S-13 work(i+1);
S-14 }
S-15 }
S-16 }
```

## 2 Example ordered.3f

```

S-1 SUBROUTINE ORDERED_GOOD(N)
S-2 INTEGER N
S-3
S-4 !$OMP DO ORDERED
S-5 DO I = 1,N
S-6 IF (I <= 10) THEN
S-7 ORDERED
S-8 CALL WORK(I)
S-9 END ORDERED
S-10 ENDIF
S-11
S-12 IF (I > 10) THEN
S-13 ORDERED
S-14 CALL WORK(I+1)
S-15 END ORDERED
S-16 ENDIF
S-17 ENDDO
S-18 END SUBROUTINE ORDERED_GOOD
```

## 1 CHAPTER 30

# 2 Cancellation Constructs

---

3 The following example shows how the **cancel** directive can be used to terminate an OpenMP  
4 region. Although the **cancel** construct terminates the OpenMP worksharing region, programmers  
5 must still track the exception through the pointer ex and issue a cancellation for the **parallel**  
6 region if an exception has been raised. The master thread checks the exception pointer to make sure  
7 that the exception is properly handled in the sequential part. If cancellation of the **parallel**  
8 region has been requested, some threads might have executed **phase\_1()**. However, it is  
9 guaranteed that none of the threads executed **phase\_2()**.

C / C++

10 Example cancellation.1c

```
S-1 #include <iostream>
S-2 #include <exception>
S-3
S-4 #define N 10000
S-5
S-6 extern void causes_an_exception();
S-7 extern void phase_1();
S-8 extern void phase_2();
S-9
S-10 void example() {
S-11 std::exception *ex = NULL;
S-12 #pragma omp parallel shared(ex)
S-13 {
S-14 #pragma omp for
S-15 for (int i = 0; i < N; i++) {
S-16 // no 'if' that prevents compiler optimizations
S-17 try {
S-18 causes_an_exception();
S-19 }
S-20 catch (std::exception *e) {
```

```

S-21 // still must remember exception for later handling
S-22 #pragma omp atomic write
S-23 ex = e;
S-24 // cancel worksharing construct
S-25 #pragma omp cancel for
S-26 }
S-27 }
S-28 // if an exception has been raised, cancel parallel region
S-29 if (ex) {
S-30 #pragma omp cancel parallel
S-31 }
S-32 phase_1();
S-33 #pragma omp barrier
S-34 phase_2();
S-35 }
S-36 // continue here if an exception has been thrown in the worksharing loop
S-37 if (ex) {
S-38 // handle exception stored in ex
S-39 }
S-40 }
```

 C / C++ 

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

 Fortran 

5 Example cancellation. If

```

S-1 subroutine example(n, dim)
S-2 integer, intent(in) :: n, dim(n)
S-3 integer :: i, s, err
S-4 real, allocatable :: B(:)
S-5 err = 0
S-6 !$omp parallel shared(err)
S-7 ! ...
S-8 !$omp do private(s, B)
S-9 do i=1, n
S-10 !$omp cancellation point do
S-11 allocate(B(dim(i)), stat=s)
S-12 if (s .gt. 0) then
S-13 !$omp atomic write
S-14 err = s
S-15 !$omp cancel do
S-16 endif
S-17 ! ...
```

```

S-18 ! deallocate private array B
S-19 if (allocated(B)) then
S-20 deallocate(B)
S-21 endif
S-22 enddo
S-23 !$omp end parallel
S-24 end subroutine

```

Fortran

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

C / C++

8 Example cancellation.2c

```

S-1 typedef struct binary_tree_s {
S-2 int value;
S-3 struct binary_tree_s *left, *right;
S-4 } binary_tree_t;
S-5
S-6 binary_tree_t *search_tree(binary_tree_t *tree, int value, int level) {
S-7 binary_tree_t *found = NULL;
S-8 if (tree) {
S-9 if (tree->value == value) {
S-10 found = tree;
S-11 }
S-12 else {
S-13 #pragma omp task shared(found) if(level < 10)
S-14 {
S-15 binary_tree_t *found_left = NULL;
S-16 found_left = search_tree(tree->left, value, level + 1);
S-17 if (found_left) {
S-18 #pragma omp atomic write
S-19 found = found_left;
S-20 #pragma omp cancel taskgroup
S-21 }
S-22 }
S-23 #pragma omp task shared(found) if(level < 10)
S-24 {
S-25 binary_tree_t *found_right = NULL;
S-26 found_right = search_tree(tree->right, value, level + 1);
S-27 if (found_right) {

```

```

S-28 #pragma omp atomic write
S-29 found = found_right;
S-30 #pragma omp cancel taskgroup
S-31 }
S-32 }
S-33 #pragma omp taskwait
S-34 }
S-35 }
S-36 return found;
S-37 }
S-38 binary_tree_t *search_tree_parallel(binary_tree_t *tree, int value) {
S-39 binary_tree_t *found = NULL;
S-40 #pragma omp parallel shared(found, tree, value)
S-41 {
S-42 #pragma omp master
S-43 {
S-44 #pragma omp taskgroup
S-45 {
S-46 found = search_tree(tree, value, 0);
S-47 }
S-48 }
S-49 }
S-50 return found;
S-51 }
```

C / C++

1 The following is the equivalent parallel search example in Fortran.

Fortran

2 *Example cancellation.2f*

```

S-1 module parallel_search
S-2 type binary_tree
S-3 integer :: value
S-4 type(binary_tree), pointer :: right
S-5 type(binary_tree), pointer :: left
S-6 end type
S-7
S-8 contains
S-9 recursive subroutine search_tree(tree, value, level, found)
S-10 type(binary_tree), intent(in), pointer :: tree
S-11 integer, intent(in) :: value, level
S-12 type(binary_tree), pointer :: found
S-13 type(binary_tree), pointer :: found_left => NULL(), found_right => NULL()
S-14
S-15 if (associated(tree)) then
S-16 if (tree%value .eq. value) then
```

```

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



1 CHAPTER 31

2 **The `threadprivate` Directive**

---

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

C / C++

5 Example `threadprivate.1c`

```
S-1 int counter = 0;
S-2 #pragma omp threadprivate(counter)
S-3
S-4 int increment_counter()
S-5 {
S-6 counter++;
S-7 return(counter);
S-8 }
```

C / C++

Fortran

6 Example `threadprivate.1f`

```
S-1 INTEGER FUNCTION INCREMENT_COUNTER()
S-2 COMMON/INC_COMMON/COUNTER
S-3 !$OMP THREADPRIVATE (/INC_COMMON/)
S-4
S-5 COUNTER = COUNTER +1
S-6 INCREMENT_COUNTER = COUNTER
S-7 RETURN
S-8 END FUNCTION INCREMENT_COUNTER
```

Fortran

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

2 *Example threadprivate.2c*

```
S-1 int increment_counter_2()
S-2 {
S-3 static int counter = 0;
S-4 #pragma omp threadprivate(counter)
S-5 counter++;
S-6 return(counter);
S-7 }
```

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

8 *Example threadprivate.3c*

```
S-1 class T {
S-2 public:
S-3 int val;
S-4 T (int);
S-5 T (const T&);
S-6 };
S-7
S-8 T :: T (int v){
S-9 val = v;
S-10 }
S-11
S-12 T :: T (const T& t) {
S-13 val = t.val;
S-14 }
S-15
S-16 void g(T a, T b){
S-17 a.val += b.val;
S-18 }
S-19
S-20 int x = 1;
S-21 T a(x);
S-22 const T b_aux(x); /* Capture value of x = 1 */
S-23 T b(b_aux);
S-24 #pragma omp threadprivate(a, b)
S-25
S-26 void f(int n) {
S-27 x++;
```

```

S-28 #pragma omp parallel for
S-29 /* In each thread:
S-30 * a is constructed from x (with value 1 or 2?)
S-31 * b is copy-constructed from b_aux
S-32 */
S-33
S-34 for (int i=0; i<n; i++) {
S-35 g(a, b); /* Value of a is unspecified. */
S-36 }
S-37 }
```

C / C++

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

Fortran

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

*Example threadprivate.2f*

```

S-1 MODULE INC_MODULE
S-2 COMMON /T/ A
S-3 END MODULE INC_MODULE
S-4
S-5 SUBROUTINE INC_MODULE_WRONG()
S-6 USE INC_MODULE
S-7 !$OMP THREADPRIVATE(/T/)
S-8 !non-conforming because /T/ not declared in INC_MODULE_WRONG
S-9 END SUBROUTINE INC_MODULE_WRONG
```

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

*Example threadprivate.3f*

```

S-1 SUBROUTINE INC_WRONG()
S-2 COMMON /T/ A
S-3 !$OMP THREADPRIVATE(/T/)
S-4
S-5 CONTAINS
S-6 SUBROUTINE INC_WRONG_SUB()
S-7 !$OMP PARALLEL COPYIN(/T/)
S-8 !non-conforming because /T/ not declared in INC_WRONG_SUB
S-9 !$OMP END PARALLEL
S-10 END SUBROUTINE INC_WRONG_SUB
S-11 END SUBROUTINE INC_WRONG
```

## Fortran (cont.)

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

2 *Example threadprivate.4f*

```
S-1 SUBROUTINE INC_GOOD()
S-2 COMMON /T/ A
S-3 !$OMP THREADPRIVATE (/T/)
S-4
S-5 CONTAINS
S-6 SUBROUTINE INC_GOOD_SUB()
S-7 COMMON /T/ A
S-8 !$OMP THREADPRIVATE (/T/)
S-9
S-10 !$OMP PARALLEL COPYIN(/T/)
S-11 !$OMP END PARALLEL
S-12 END SUBROUTINE INC_GOOD_SUB
S-13 END SUBROUTINE INC_GOOD
```

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

4 *Example threadprivate.5f*

```
S-1 PROGRAM INC_GOOD2
S-2 INTEGER, ALLOCATABLE, SAVE :: A(:)
S-3 INTEGER, POINTER, SAVE :: PTR
S-4 INTEGER, SAVE :: I
S-5 INTEGER, TARGET :: TARG
S-6 LOGICAL :: FIRSTIN = .TRUE.
S-7 !$OMP THREADPRIVATE (A, I, PTR)
S-8
S-9 ALLOCATE (A(3))
S-10 A = (/1,2,3/)
S-11 PTR => TARG
S-12 I = 5
S-13
S-14 !$OMP PARALLEL COPYIN(I, PTR)
S-15 !$OMP CRITICAL
S-16 IF (FIRSTIN) THEN
S-17 TARG = 4 ! Update target of ptr
S-18 I = I + 10
S-19 IF (ALLOCATED(A)) A = A + 10
S-20 FIRSTIN = .FALSE.
S-21 END IF
S-22
S-23 IF (ALLOCATED(A)) THEN
S-24 PRINT *, 'a = ', A
```

## Fortran (cont.)

```
S-25 ELSE
S-26 PRINT *, 'A is not allocated'
S-27 END IF
S-28
S-29 PRINT *, 'ptr = ', PTR
S-30 PRINT *, 'i = ', I
S-31 PRINT *
S-32
S-33 !$OMP END CRITICAL
S-34 !$OMP END PARALLEL
S-35 END PROGRAM INC_GOOD2
```

1 The above program, if executed by two threads, will print one of the following two sets of output:

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

5 A is not allocated
6 ptr = 4
7 i = 5

8 or

9 A is not allocated
10 ptr = 4
11 i = 15

12 a = 1 2 3
13 ptr = 4
14 i = 5
```

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

16 Example *threadprivate.f90*

```
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)
```

```

S-12 WRITE(*,*) THE_SUM
S-13 !$OMP END PARALLEL
S-14 END SUBROUTINE SUB1
S-15
S-16 SUBROUTINE SUB2 (THE_SUM)
S-17 USE INC_MODULE_GOOD3
S-18 WORK(:) = 10
S-19 THE_SUM=SUM(WORK)
S-20 END SUBROUTINE SUB2
S-21
S-22 PROGRAM INC_GOOD3
S-23 N = 10
S-24 CALL SUB1(N)
S-25 END PROGRAM INC_GOOD3

```

Fortran  
C++

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

4 Example *threadprivate.4c*

```

S-1 static T t1;
S-2 #pragma omp threadprivate(t1)
S-3 static T t2(23);
S-4 #pragma omp threadprivate(t2)
S-5 static T t3 = f();
S-6 #pragma omp threadprivate(t3)

```

5 The following example illustrates the use of **threadprivate** for static class members. The  
 6 **threadprivate** directive for a static class member must be placed inside the class definition.

7 Example *threadprivate.5c*

```

S-1 class T {
S-2 public:
S-3 static int i;
S-4 #pragma omp threadprivate(i)
S-5 };

```

C++

1 CHAPTER 32

2

## Parallel Random Access Iterator Loop

---

4 The following example shows a parallel random access iterator loop.

5 Example *pra\_iterator.1c*

```
S-1 #include <vector>
S-2 void iterator_example()
S-3 {
S-4 std::vector<int> vec(23);
S-5 std::vector<int>::iterator it;
S-6 #pragma omp parallel for default(none) shared(vec)
S-7 for (it = vec.begin(); it < vec.end(); it++)
S-8 {
S-9 // do work with *it //
S-10 }
S-11 }
```

C++

C++

1    **CHAPTER 33**

2

## Fortran Restrictions on shared 3    and private Clauses with Common 4    Blocks

---

5

6                                  **Fortran**

7

8    When a named common block is specified in a **private**, **firstprivate**, or **lastprivate**  
9    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.

10   The following example is conforming:

11   *Example fort\_sp\_common.1f*

```
S-1 SUBROUTINE COMMON_GOOD ()
S-2 COMMON /C/ X, Y
S-3 REAL X, Y
S-4
S-5 !$OMP PARALLEL PRIVATE (/C/)
S-6 ! do work here
S-7 !$OMP END PARALLEL
S-8 !$OMP PARALLEL SHARED (X, Y)
S-9 ! do work here
S-10 !$OMP END PARALLEL
S-11 END SUBROUTINE COMMON_GOOD
```

10   The following example is also conforming:

11   *Example fort\_sp\_common.2f*

## Fortran (cont.)

```
S-1 SUBROUTINE COMMON_GOOD2()
S-2 COMMON /C/ X, Y
S-3 REAL X, Y
S-4 INTEGER I
S-5 !$OMP PARALLEL
S-6 !$OMP DO PRIVATE(/C/)
S-7 DO I=1,1000
S-8 ! do work here
S-9 ENDDO
S-10 !$OMP END DO
S-11 !$OMP DO PRIVATE(X)
S-12 DO I=1,1000
S-13 ! do work here
S-14 ENDDO
S-15 !$OMP END DO
S-16 !$OMP END PARALLEL
S-17 END SUBROUTINE COMMON_GOOD2
```

1 The following example is conforming:

2 *Example fort\_sp\_common.3f*

```
S-1 SUBROUTINE COMMON_GOOD3()
S-2 COMMON /C/ X, Y
S-3 !$OMP PARALLEL PRIVATE (/C/)
S-4 ! do work here
S-5 !$OMP END PARALLEL
S-6 !$OMP PARALLEL SHARED (/C/)
S-7 ! do work here
S-8 !$OMP END PARALLEL
S-9 END SUBROUTINE COMMON_GOOD3
```

3 The following example is non-conforming because **x** is a constituent element of **c**:

4 *Example fort\_sp\_common.4f*

```
S-1 SUBROUTINE COMMON_WRONG()
S-2 COMMON /C/ X, Y
S-3 ! Incorrect because X is a constituent element of C
S-4 !$OMP PARALLEL PRIVATE(/C/), SHARED(X)
S-5 ! do work here
S-6 !$OMP END PARALLEL
S-7 END SUBROUTINE COMMON_WRONG
```

5 The following example is non-conforming because a common block may not be declared both  
6 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
```



Fortran

## 1 CHAPTER 34

# 2 The default (none) Clause

---

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

C / C++

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

8 Example *default\_none.1c*

```
S-1 #include <omp.h>
S-2 int x, y, z[1000];
S-3 #pragma omp threadprivate(x)
S-4
S-5 void default_none(int a) {
S-6 const int c = 1;
S-7 int i = 0;
S-8
S-9 #pragma omp parallel default(none) private(a) shared(z, c)
S-10 {
S-11 int j = omp_get_num_threads();
S-12 /* O.K. - j is declared within parallel region */
S-13 a = z[j]; /* O.K. - a is listed in private clause */
S-14 /* - z is listed in shared clause */
S-15 x = c; /* O.K. - x is threadprivate */
S-16 /* - c has const-qualified type and
S-17 is listed in shared clause */
S-18 z[i] = y; /* Error - cannot reference i or y here */
S-19
S-20 #pragma omp for firstprivate(y)
S-21 /* Error - Cannot reference y in the firstprivate clause */
S-22 for (i=0; i<10 ; i++) {
```

```

S-23 z[i] = i; /* O.K. - i is the loop iteration variable */
S-24 }
S-25
S-26 z[i] = y; /* Error - cannot reference i or y here */
S-27 }
S-28 }
```

C / C++

Fortran

1

*Example default\_none.If*

```

S-1 SUBROUTINE DEFAULT_NONE (A)
S-2 INCLUDE "omp_lib.h" ! or USE OMP_LIB
S-3
S-4 INTEGER A
S-5
S-6 INTEGER X, Y, Z(1000)
S-7 COMMON/BLOCKX/X
S-8 COMMON/BLOCKY/Y
S-9 COMMON/BLOCKZ/Z
S-10 !$OMP THREADPRIVATE (/BLOCKX/)
S-11
S-12 INTEGER I, J
S-13 i = 1
S-14
S-15 !$OMP PARALLEL DEFAULT (NONE) PRIVATE (A) SHARED (Z) PRIVATE (J)
S-16 J = OMP_GET_NUM_THREADS();
S-17 ! O.K. - J is listed in PRIVATE clause
S-18 A = Z(J) ! O.K. - A is listed in PRIVATE clause
S-19 ! - Z is listed in SHARED clause
S-20 X = 1 ! O.K. - X is THREADPRIVATE
S-21 Z(I) = Y ! Error - cannot reference I or Y here
S-22
S-23 !$OMP DO firstprivate(y)
S-24 ! Error - Cannot reference y in the firstprivate clause
S-25 DO I = 1,10
S-26 Z(I) = I ! O.K. - I is the loop iteration variable
S-27 END DO
S-28
S-29
S-30 Z(I) = Y ! Error - cannot reference I or Y here
S-31 !$OMP END PARALLEL
S-32 END SUBROUTINE DEFAULT_NONE
```

Fortran

1 CHAPTER 35

2

## 3 Race Conditions Caused by Implied 4 Copies of Shared Variables in Fortran

---

Fortran

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

11 Example *fort\_race.f90*

```
S-1 SUBROUTINE SHARED_RACE
S-2
S-3 INCLUDE "omp_lib.h" ! or USE OMP_LIB
S-4
S-5 REAL A(20)
S-6 INTEGER MYTHREAD
S-7
S-8 !$OMP PARALLEL SHARED(A) PRIVATE(MYTHREAD)
S-9
S-10 MYTHREAD = OMP_GET_THREAD_NUM()
S-11 IF (MYTHREAD .EQ. 0) THEN
S-12 CALL SUB(A(1:10)) ! compiler may introduce writes to A(6:10)
S-13 ELSE
S-14 A(6:10) = 12
S-15 ENDIF
S-16
S-17 !$OMP END PARALLEL
```

```
S-18
S-19 END SUBROUTINE SHARED_RACE
S-20
S-21 SUBROUTINE SUB(X)
S-22 REAL X(*)
S-23 X(1:5) = 4
S-24 END SUBROUTINE SUB
```

Fortran

## 1 CHAPTER 36

# 2 The private Clause

---

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

C / C++

6 Example *private.1c*

```
S-1 #include <stdio.h>
S-2 #include <assert.h>
S-3
S-4 int main()
S-5 {
S-6 int i, j;
S-7 int *ptr_i, *ptr_j;
S-8
S-9 i = 1;
S-10 j = 2;
S-11
S-12 ptr_i = &i;
S-13 ptr_j = &j;
S-14
S-15 #pragma omp parallel private(i) firstprivate(j)
S-16 {
S-17 i = 3;
S-18 j = j + 2;
S-19 assert (*ptr_i == 1 && *ptr_j == 2);
S-20 }
S-21
S-22 assert(i == 1 && j == 2);
S-23
S-24 return 0;
S-25 }
```



C / C++

Fortran

1

*Example private.1f*

```
S-1 PROGRAM PRIV_EXAMPLE
S-2 INTEGER I, J
S-3
S-4 I = 1
S-5 J = 2
S-6
S-7 !$OMP PARALLEL PRIVATE(I) FIRSTPRIVATE(J)
S-8 I = 3
S-9 J = J + 2
S-10 !$OMP END PARALLEL
S-11
S-12 PRINT *, I, J ! I .eq. 1 .and. J .eq. 2
S-13 END PROGRAM PRIV_EXAMPLE
```



Fortran

2

3

4

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



C / C++

5

*Example private.2c*

```
S-1 int a;
S-2
S-3 void g(int k) {
S-4 a = k; /* Accessed in the region but outside of the construct;
S-5 * therefore unspecified whether original or private list
S-6 * item is modified. */
S-7 }
S-8
S-9
S-10 void f(int n) {
S-11 int a = 0;
S-12
S-13 #pragma omp parallel for private(a)
S-14 for (int i=1; i<n; i++) {
S-15 a = i;
S-16 g(a*2); /* Private copy of "a" */
S-17 }
S-18 }
```



C / C++

---

Fortran

---

1       *Example private.2f*

```
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
```

---

Fortran

---

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

---

C / C++

---

5       *Example private.3c*

```
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)
```

```
S-10 for (i=0; i<10; i++)
S-11 {
S-12 a = 2;
S-13 }
S-14 assert(a == 1);
S-15 }
S-16 }
```

C / C++  
Fortran

1

*Example private.3f*

```
S-1 SUBROUTINE PRIV_EXAMPLE3()
S-2 INTEGER I, A
S-3
S-4 !$OMP PARALLEL PRIVATE(A)
S-5 A = 1
S-6 !$OMP PARALLEL DO PRIVATE(A)
S-7 DO I = 1, 10
S-8 A = 2
S-9 END DO
S-10 !$OMP END PARALLEL DO
S-11 PRINT *, A ! Outer A still has value 1
S-12 !$OMP END PARALLEL
S-13 END SUBROUTINE PRIV_EXAMPLE3
```

Fortran

1 CHAPTER 37

2

## 3 Fortran Restrictions on Storage Association with the **private** Clause

---

Fortran

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

6 Example *fort\_sa\_private.1f*

```
S-1 SUBROUTINE SUB()
S-2 COMMON /BLOCK/ X
S-3 PRINT *,X ! X is undefined
S-4 END SUBROUTINE SUB
S-5
S-6 PROGRAM PRIV_RESTRICT
S-7 COMMON /BLOCK/ X
S-8 X = 1.0
S-9 !$OMP PARALLEL PRIVATE (X)
S-10 X = 2.0
S-11 CALL SUB()
S-12 !$OMP END PARALLEL
S-13 END PROGRAM PRIV_RESTRICT
```

7 Example *fort\_sa\_private.2f*

```
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
```

## Fortran (cont.)

```
S-9
S-10 CONTAINS
S-11
S-12 SUBROUTINE SUB()
S-13 COMMON /BLOCK2/ Y
S-14
S-15 PRINT *,X ! X is undefined
S-16 PRINT *,Y ! Y is undefined
S-17 END SUBROUTINE SUB
S-18
S-19 END PROGRAM PRIV_RESTRICT2
```

1      Example *fort\_sa\_private.3f*

```
S-1 PROGRAM PRIV_RESTRICT3
S-2 EQUIVALENCE (X,Y)
S-3 X = 1.0
S-4
S-5 !$OMP PARALLEL PRIVATE(X)
S-6 PRINT *,Y ! Y is undefined
S-7 Y = 10
S-8 PRINT *,X ! X is undefined
S-9 !$OMP END PARALLEL
S-10 END PROGRAM PRIV_RESTRICT3
```

2      Example *fort\_sa\_private.4f*

```
S-1 PROGRAM PRIV_RESTRICT4
S-2 INTEGER I, J
S-3 INTEGER A(100), B(100)
S-4 EQUIVALENCE (A(51), B(1))
S-5
S-6 !$OMP PARALLEL DO DEFAULT(PRIVATE) PRIVATE(I,J) LASTPRIVATE(A)
S-7 DO I=1,100
S-8 DO J=1,100
S-9 B(J) = J - 1
S-10 ENDDO
S-11
S-12 DO J=1,100
S-13 A(J) = J ! B becomes undefined at this point
S-14 ENDDO
S-15
S-16 DO J=1,50
S-17 B(J) = B(J) + 1 ! B is undefined
S-18 ! A becomes undefined at this point
S-19 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

```

1      Example for *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

```



Fortran

1    **CHAPTER 38**

2

## C/C++ Arrays in a **firstprivate** Clause

3

---

C / C++

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

7    In this example:

- 8    • The type of **A** is array of two arrays of two ints.  
9    • The type of **B** is adjusted to pointer to array of **n** ints, because it is a function parameter.  
10   • The type of **C** is adjusted to pointer to int, because it is a function parameter.  
11   • The type of **D** is array of two arrays of two ints.  
12   • The type of **E** is array of **n** arrays of **n** ints.

13   Note that **B** and **E** involve variable length array types.

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

17   Example *carrays\_fpriv.1c*

```
S-1 #include <assert.h>
S-2
S-3 int A[2][2] = {1, 2, 3, 4};
S-4
S-5 void f(int n, int B[n][n], int C[])
S-6 {
S-7 int D[2][2] = {1, 2, 3, 4};
S-8 int E[n][n];
S-9 }
```

```
S-10 assert(n >= 2);
S-11 E[1][1] = 4;
S-12
S-13 #pragma omp parallel firstprivate(B, C, D, E)
S-14 {
S-15 assert(sizeof(B) == sizeof(int (*)[n]));
S-16 assert(sizeof(C) == sizeof(int *));
S-17 assert(sizeof(D) == 4 * sizeof(int));
S-18 assert(sizeof(E) == n * n * sizeof(int));
S-19
S-20 /* Private B and C have values of original B and C. */
S-21 assert(&B[1][1] == &A[1][1]);
S-22 assert(&C[3] == &A[1][1]);
S-23 assert(D[1][1] == 4);
S-24 assert(E[1][1] == 4);
S-25 }
S-26 }
S-27
S-28 int main() {
S-29 f(2, A, A[0]);
S-30 return 0;
S-31 }
```

C / C++

1 CHAPTER 39

2 **The lastprivate Clause**

---

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

C / C++

6 Example *lastprivate.1c*

```
S-1 void lastpriv (int n, float *a, float *b)
S-2 {
S-3 int i;
S-4
S-5 #pragma omp parallel
S-6 {
S-7 #pragma omp for lastprivate(i)
S-8 for (i=0; i<n-1; i++)
S-9 a[i] = b[i] + b[i+1];
S-10 }
S-11
S-12 a[i]=b[i]; /* i == n-1 here */
S-13 }
```

C / C++

Forran

1 Example lastprivate. If

```
S-1 SUBROUTINE LASTPRIV(N, A, B)
S-2
S-3 INTEGER N
S-4 REAL A(*), B(*)
S-5 INTEGER I
S-6 !$OMP PARALLEL
S-7 !$OMP DO LASTPRIVATE(I)
S-8
S-9 DO I=1,N-1
S-10 A(I) = B(I) + B(I+1)
S-11 ENDDO
S-12
S-13 !$OMP END PARALLEL
S-14 A(I) = B(I) ! I has the value of N here
S-15
S-16 END SUBROUTINE LASTPRIV
```

Forran

1 CHAPTER 40

2 **The reduction Clause**

---

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

5 C / C++

*Example reduction.1c*

```
S-1 #include <math.h>
S-2 void reduction1(float **x, int *y, int n)
S-3 {
S-4 int i, b, c;
S-5 float a, d;
S-6 a = 0.0;
S-7 b = 0;
S-8 c = y[0];
S-9 d = x[0];
S-10 #pragma omp parallel for private(i) shared(x, y, n) \
S-11 reduction(+:a) reduction(^:b) \
S-12 reduction(min:c) reduction(max:d)
S-13 for (i=0; i<n; i++) {
S-14 a += x[i];
S-15 b ^= y[i];
S-16 if (c > y[i]) c = y[i];
S-17 d = fmaxf(d,x[i]);
S-18 }
S-19 }
```

C / C++

---

Fortran

---

1      Example reduction.1f

```
S-1 SUBROUTINE REDUCTION1(A, B, C, D, X, Y, N)
S-2 REAL :: X(*), A, D
S-3 INTEGER :: Y(*), N, B, C
S-4 INTEGER :: I
S-5 A = 0
S-6 B = 0
S-7 C = Y(1)
S-8 D = X(1)
S-9 !$OMP PARALLEL DO PRIVATE(I) SHARED(X, Y, N) REDUCTION(+:A) &
S-10 !$OMP& REDUCTION(IEOR:B) REDUCTION(MIN:C) REDUCTION(MAX:D)
S-11 DO I=1,N
S-12 A = A + X(I)
S-13 B = IEOR(B, Y(I))
S-14 C = MIN(C, Y(I))
S-15 IF (D < X(I)) D = X(I)
S-16 END DO
S-17
S-18 END SUBROUTINE REDUCTION1
```

---

Fortran

---

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

---

C / C++

---

4      Example reduction.2c

```
S-1 #include <limits.h>
S-2 #include <math.h>
S-3 void reduction2(float *x, int *y, int n)
S-4 {
S-5 int i, b, b_p, c, c_p;
S-6 float a, a_p, d, d_p;
S-7 a = 0.0f;
S-8 b = 0;
S-9 c = y[0];
S-10 d = x[0];
S-11 #pragma omp parallel shared(a, b, c, d, x, y, n) \
S-12 private(a_p, b_p, c_p, d_p)
S-13 {
S-14 a_p = 0.0f;
S-15 b_p = 0;
S-16 c_p = INT_MAX;
S-17 d_p = -HUGE_VALF;
S-18 #pragma omp for private(i)
```

```

S-19 for (i=0; i<n; i++) {
S-20 a_p += x[i];
S-21 b_p ^= y[i];
S-22 if (c_p > y[i]) c_p = y[i];
S-23 d_p = fmaxf(d_p, x[i]);
S-24 }
S-25 #pragma omp critical
S-26 {
S-27 a += a_p;
S-28 b ^= b_p;
S-29 if(c > c_p) c = c_p;
S-30 d = fmaxf(d, d_p);
S-31 }
S-32 }
S-33 }
```

C / C++

Fortran

1

*Example reduction.2f*

```

S-1 SUBROUTINE REDUCTION2(A, B, C, D, X, Y, N)
S-2 REAL :: X(*), A, D
S-3 INTEGER :: Y(*), N, B, C
S-4 REAL :: A_P, D_P
S-5 INTEGER :: I, B_P, C_P
S-6 A = 0
S-7 B = 0
S-8 C = Y(1)
S-9 D = X(1)
S-10 !$OMP PARALLEL SHARED(X, Y, A, B, C, D, N) &
S-11 !$OMP& PRIVATE(A_P, B_P, C_P, D_P)
S-12 A_P = 0.0
S-13 B_P = 0
S-14 C_P = HUGE(C_P)
S-15 D_P = -HUGE(D_P)
S-16 !$OMP DO PRIVATE(I)
S-17 DO I=1,N
S-18 A_P = A_P + X(I)
S-19 B_P = IEOR(B_P, Y(I))
S-20 C_P = MIN(C_P, Y(I))
S-21 IF (D_P < X(I)) D_P = X(I)
S-22 END DO
S-23 !$OMP CRITICAL
S-24 A = A + A_P
S-25 B = IEOR(B, B_P)
S-26 C = MIN(C, C_P)
S-27 D = MAX(D, D_P)
```

## Fortran (cont.)

```
S-28 !$OMP END CRITICAL
S-29 !$OMP END PARALLEL
S-30 END SUBROUTINE REDUCTION2
```

1 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**.  
2

3 *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
```

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

6 *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
```

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

9 *Example reduction.5f*

```

S-1 MODULE MOD
S-2 INTRINSIC MAX, MIN
S-3 END MODULE MOD
S-4
S-5 PROGRAM REDUCTION4
S-6 USE MOD, MIN=>MAX, MAX=>MIN
S-7 REAL :: R
S-8 R = -HUGE(0.0)
S-9
S-10 !$OMP PARALLEL DO REDUCTION(MIN: R) ! still does MAX
S-11 DO I = 1, 1000
S-12 R = MIN(R, SIN(REAL(I)))
S-13 END DO
S-14 PRINT *, R
S-15 END PROGRAM REDUCTION4

```

Fortran

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

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

C / C++

8 *Example reduction.3c*

```

S-1 #include <stdio.h>
S-2
S-3 int main (void)
S-4 {
S-5 int a, i;
S-6
S-7 #pragma omp parallel shared(a) private(i)
S-8 {
S-9 #pragma omp master
S-10 a = 0;
S-11
S-12 // To avoid race conditions, add a barrier here.
S-13
S-14 #pragma omp for reduction(+:a)
S-15 for (i = 0; i < 10; i++) {
S-16 a += i;
S-17 }
S-18

```

```
S-19 #pragma omp single
S-20 printf ("Sum is %d\n", a);
S-21 }
S-22 return 0;
S-23 }
```

C / C++

Fortran

1      *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
```

Fortran

1 CHAPTER 41

2 **The copyin Clause**

---

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

C / C++

6 Example *copyin.1c*

```
S-1 #include <stdlib.h>
S-2
S-3 float* work;
S-4 int size;
S-5 float tol;
S-6
S-7 #pragma omp threadprivate(work,size,tol)
S-8
S-9 void build()
S-10 {
S-11 int i;
S-12 work = (float*)malloc(sizeof(float)*size);
S-13 for(i = 0; i < size; ++i) work[i] = tol;
S-14 }
S-15
S-16 void copyin_example(float t, int n)
S-17 {
S-18 tol = t;
S-19 size = n;
S-20 #pragma omp parallel copyin(tol,size)
S-21 {
S-22 build();
S-23 }
S-24 }
```

C / C++  
Fortran

1       *Example copyin.If*

```

S-1 MODULE M
S-2 REAL, POINTER, SAVE :: WORK(:)
S-3 INTEGER :: SIZE
S-4 REAL :: TOL
S-5 !$OMP THREADPRIVATE(WORK,SIZE,TOL)
S-6 END MODULE M
S-7
S-8 SUBROUTINE COPYIN_EXAMPLE(T, N)
S-9 USE M
S-10 REAL :: T
S-11 INTEGER :: N
S-12 TOL = T
S-13 SIZE = N
S-14 !$OMP PARALLEL COPYIN(TOL,SIZE)
S-15 CALL BUILD
S-16 !$OMP END PARALLEL
S-17 END SUBROUTINE COPYIN_EXAMPLE
S-18
S-19 SUBROUTINE BUILD
S-20 USE M
S-21 ALLOCATE(WORK(SIZE))
S-22 WORK = TOL
S-23 END SUBROUTINE BUILD

```

Fortran

1 CHAPTER 42

2 

## The copyprivate Clause

---

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

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

C / C++ ▾

12 Example *copyprivate.1c*

```
S-1 #include <stdio.h>
S-2 float x, y;
S-3 #pragma omp threadprivate(x, y)
S-4
S-5 void init(float a, float b) {
S-6 #pragma omp single copyprivate(a,b,x,y)
S-7 {
S-8 scanf("%f %f %f %f", &a, &b, &x, &y);
S-9 }
S-10 }
```

▲ C / C++

Fortran

1       *Example copyprivate.If*

```
S-1 SUBROUTINE INIT(A,B)
S-2 REAL A, B
S-3 COMMON /XY/ X,Y
S-4 !$OMP THREADPRIVATE (/XY/)
S-5
S-6 !$OMP SINGLE
S-7 READ (11) A,B,X,Y
S-8 !$OMP END SINGLE COPYPRIVATE (A,B,/XY/)
S-9
S-10 END SUBROUTINE INIT
```

Fortran

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

C / C++

5       *Example copyprivate.2c*

```
S-1 #include <stdio.h>
S-2 #include <stdlib.h>
S-3
S-4 float read_next() {
S-5 float * tmp;
S-6 float return_val;
S-7
S-8 #pragma omp single copyprivate(tmp)
S-9 {
S-10 tmp = (float *) malloc(sizeof(float));
S-11 } /* copies the pointer only */
S-12
S-13
S-14 #pragma omp master
S-15 {
S-16 scanf("%f", tmp);
S-17 }
S-18
S-19 #pragma omp barrier
S-20 return_val = *tmp;
S-21 #pragma omp barrier
S-22
S-23 #pragma omp single nowait
S-24 {
S-25 free(tmp);
```

```
S-26 }
S-27
S-28 return return_val;
S-29 }
```

C / C++  
Fortran

1 Example copyprivate.2f

```
S-1 REAL FUNCTION READ_NEXT()
S-2 REAL, POINTER :: TMP
S-3
S-4 !$OMP SINGLE
S-5 ALLOCATE (TMP)
S-6 END SINGLE COPYPRIVATE (TMP) ! copies the pointer only
S-7
S-8 !$OMP MASTER
S-9 READ (11) TMP
S-10 !$OMP END MASTER
S-11
S-12 !$OMP BARRIER
S-13 READ_NEXT = TMP
S-14 !$OMP BARRIER
S-15
S-16 !$OMP SINGLE
S-17 DEALLOCATE (TMP)
S-18 !$OMP END SINGLE NOWAIT
S-19 END FUNCTION READ_NEXT
```

Fortran

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

## 1 Example copyprivate.3c

```

S-1 #include <stdio.h>
S-2 #include <stdlib.h>
S-3 #include <omp.h>
S-4
S-5 omp_lock_t *new_lock()
S-6 {
S-7 omp_lock_t *lock_ptr;
S-8
S-9 #pragma omp single copyprivate(lock_ptr)
S-10 {
S-11 lock_ptr = (omp_lock_t *) malloc(sizeof(omp_lock_t));
S-12 omp_init_lock(lock_ptr);
S-13 }
S-14
S-15 return lock_ptr;
S-16 }
```

## 2 Example copyprivate.3f

```

S-1 FUNCTION NEW_LOCK()
S-2 USE OMP_LIB ! or INCLUDE "omp_lib.h"
S-3 INTEGER(OMP_LOCK_KIND), POINTER :: NEW_LOCK
S-4
S-5 !$OMP SINGLE
S-6 ALLOCATE(NEW_LOCK)
S-7 CALL OMP_INIT_LOCK(NEW_LOCK)
S-8 !$OMP END SINGLE COPYPRIVATE(NEW_LOCK)
S-9 END FUNCTION NEW_LOCK
```

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

## 7 Example copyprivate.4f

```
S-1 SUBROUTINE S (N)
S-2 INTEGER N
S-3
S-4 REAL, DIMENSION(:), ALLOCATABLE :: A
S-5 REAL, DIMENSION(:), POINTER :: B
S-6
S-7 ALLOCATE (A (N))
S-8 !$OMP SINGLE
S-9 ALLOCATE (B (N))
S-10 READ (11) A,B
S-11 !$OMP END SINGLE COPYPRIVATE (A,B)
S-12 ! Variable A is private and is
S-13 ! assigned the same value in each thread
S-14 ! Variable B is shared
S-15
S-16 !$OMP BARRIER
S-17 !$OMP SINGLE
S-18 DEALLOCATE (B)
S-19 !$OMP END SINGLE NOWAIT
S-20 END SUBROUTINE S
```

Fortran

1 CHAPTER 43

2 

## Nested Loop Constructs

---

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

C / C++

5 *Example nested\_loop.lc*

```
S-1 void work(int i, int j) {}
S-2
S-3 void good_nesting(int n)
S-4 {
S-5 int i, j;
S-6 #pragma omp parallel default(shared)
S-7 {
S-8 #pragma omp for
S-9 for (i=0; i<n; i++) {
S-10 #pragma omp parallel shared(i, n)
S-11 {
S-12 #pragma omp for
S-13 for (j=0; j < n; j++)
S-14 work(i, j);
S-15 }
S-16 }
S-17 }
S-18}
```

C / C++

Forran

1 Example nested\_loop.1f

```
S-1 SUBROUTINE WORK(I, J)
S-2 INTEGER I, J
S-3 END SUBROUTINE WORK
S-4
S-5 SUBROUTINE GOOD_NESTING(N)
S-6 INTEGER N
S-7
S-8 INTEGER I
S-9 !$OMP PARALLEL DEFAULT(SHARED)
S-10 DO
S-11 DO I = 1, N
S-12 !$OMP PARALLEL SHARED(I, N)
S-13 DO
S-14 DO J = 1, N
S-15 CALL WORK(I, J)
S-16 END DO
S-17 !$OMP END PARALLEL
S-18 END DO
S-19 !$OMP END PARALLEL
S-20 END SUBROUTINE GOOD_NESTING
```

Forran

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

C / C++

3 Example nested\_loop.2c

```
S-1 void work(int i, int j) {}
S-2
S-3
S-4 void work1(int i, int n)
S-5 {
S-6 int j;
S-7 #pragma omp parallel default(shared)
S-8 {
S-9 #pragma omp for
S-10 for (j=0; j<n; j++)
S-11 work(i, j);
S-12 }
S-13 }
S-14
S-15
S-16 void good_nesting2(int n)
S-17 {
```

```

S-18 int i;
S-19 #pragma omp parallel default(shared)
S-20 {
S-21 #pragma omp for
S-22 for (i=0; i<n; i++)
S-23 work1(i, n);
S-24 }
S-25 }
```

C / C++  
Fortran

1      *Example nested\_loop.2f*

```

S-1 SUBROUTINE WORK(I, J)
S-2 INTEGER I, J
S-3 END SUBROUTINE WORK
S-4
S-5 SUBROUTINE WORK1(I, N)
S-6 INTEGER J
S-7 !$OMP PARALLEL DEFAULT(SHARED)
S-8 !$OMP DO
S-9 DO J = 1, N
S-10 CALL WORK(I, J)
S-11 END DO
S-12 !$OMP END PARALLEL
S-13 END SUBROUTINE WORK1
S-14
S-15 SUBROUTINE GOOD_NESTING2(N)
S-16 INTEGER N
S-17 !$OMP PARALLEL DEFAULT(SHARED)
S-18 !$OMP DO
S-19 DO I = 1, N
S-20 CALL WORK1(I, N)
S-21 END DO
S-22 !$OMP END PARALLEL
S-23 END SUBROUTINE GOOD_NESTING2
```

Fortran

1 CHAPTER 44

2

## Restrictions on Nesting of Regions

---

3 The examples in this section illustrate the region nesting rules.

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

C / C++

6 Example *nesting\_restrict.1c*

```
S-1 void work(int i, int j) {}
S-2 void wrong1(int n)
S-3 {
S-4 #pragma omp parallel default(shared)
S-5 {
S-6 int i, j;
S-7 #pragma omp for
S-8 for (i=0; i<n; i++) {
S-9 /* incorrect nesting of loop regions */
S-10 #pragma omp for
S-11 for (j=0; j<n; j++)
S-12 work(i, j);
S-13 }
S-14 }
S-15 }
```

C / C++

Forran

1 Example nesting\_restrict.1f

```
S-1 SUBROUTINE WORK(I, J)
S-2 INTEGER I, J
S-3 END SUBROUTINE WORK
S-4 SUBROUTINE WRONG1(N)
S-5 INTEGER N
S-6 INTEGER I, J
S-7 !$OMP PARALLEL DEFAULT(SHARED)
S-8 DO
S-9 DO I = 1, N
S-10 DO ! incorrect nesting of loop regions
S-11 DO J = 1, N
S-12 CALL WORK(I, J)
S-13 END DO
S-14 END DO
S-15 END PARALLEL
S-16 END SUBROUTINE WRONG1
```

Forran

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

C / C++

3 Example nesting\_restrict.2c

```
S-1 void work(int i, int j) {}
S-2 void work1(int i, int n)
S-3 {
S-4 int j;
S-5 /* incorrect nesting of loop regions */
S-6 #pragma omp for
S-7 for (j=0; j<n; j++)
S-8 work(i, j);
S-9 }
S-10
S-11 void wrong2(int n)
S-12 {
S-13 #pragma omp parallel default(shared)
S-14 {
S-15 int i;
S-16 #pragma omp for
S-17 for (i=0; i<n; i++)
S-18 work1(i, n);
S-19 }
S-20 }
```

C / C++

Fortran

1

*Example nesting\_restrict.2f*

```
S-1 SUBROUTINE WORK1(I,N)
S-2 INTEGER I, N
S-3 INTEGER J
S-4 !$OMP DO ! incorrect nesting of loop regions
S-5 DO J = 1, N
S-6 CALL WORK(I,J)
S-7 END DO
S-8 END SUBROUTINE WORK1
S-9 SUBROUTINE WRONG2(N)
S-10 INTEGER N
S-11 INTEGER I
S-12 !$OMP PARALLEL DEFAULT(SHARED)
S-13 !$OMP DO
S-14 DO I = 1, N
S-15 CALL WORK1(I,N)
S-16 END DO
S-17 !$OMP END PARALLEL
S-18 END SUBROUTINE WRONG2
```

Fortran

2

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

C / C++

3

*Example nesting\_restrict.3c*

```
S-1 void work(int i, int j) {}
S-2 void wrong3(int n)
S-3 {
S-4 #pragma omp parallel default(shared)
S-5 {
S-6 int i;
S-7 #pragma omp for
S-8 for (i=0; i<n; i++) {
S-9 /* incorrect nesting of regions */
S-10 #pragma omp single
S-11 work(i, 0);
S-12 }
S-13 }
S-14 }
```

C / C++

Forran

1       *Example nesting\_restrict.3f*

```
S-1 SUBROUTINE WRONG3(N)
S-2 INTEGER N
S-3
S-4 INTEGER I
S-5 !$OMP PARALLEL DEFAULT(SHARED)
S-6 DO
S-7 DO I = 1, N
S-8 SINGLE ! incorrect nesting of regions
S-9 CALL WORK(I, 1)
S-10 END SINGLE
S-11 END DO
S-12 END PARALLEL
S-13 END SUBROUTINE WRONG3
```

Forran

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

C / C++

4       *Example nesting\_restrict.4c*

```
S-1 void work(int i, int j) {}
S-2 void wrong4(int n)
S-3 {
S-4
S-5 #pragma omp parallel default(shared)
S-6 {
S-7 int i;
S-8 #pragma omp for
S-9 for (i=0; i<n; i++) {
S-10 work(i, 0);
S-11 /* incorrect nesting of barrier region in a loop region */
S-12 #pragma omp barrier
S-13 work(i, 1);
S-14 }
S-15 }
S-16 }
```

C / C++



## Fortran

1      Example *nesting\_restrict.4f*

```
S-1 SUBROUTINE WRONG4 (N)
S-2 INTEGER N
S-3
S-4 INTEGER I
S-5 !$OMP PARALLEL DEFAULT(SHARED)
S-6 !$OMP DO
S-7 DO I = 1, N
S-8 CALL WORK(I, 1)
S-9 ! incorrect nesting of barrier region in a loop region
S-10 !$OMP BARRIER
S-11 CALL WORK(I, 2)
S-12 END DO
S-13 !$OMP END PARALLEL
S-14 END SUBROUTINE WRONG4
```



## Fortran

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



## C / C++

5      Example *nesting\_restrict.5c*

```
S-1 void work(int i, int j) {}
S-2 void wrong5(int n)
S-3 {
S-4 #pragma omp parallel
S-5 {
S-6 #pragma omp critical
S-7 {
S-8 work(n, 0);
S-9 /* incorrect nesting of barrier region in a critical region */
S-10 #pragma omp barrier
S-11 work(n, 1);
S-12 }
S-13 }
S-14 }
```



## C / C++

Forran

```
1 Example nesting_restrict.5f
S-1 SUBROUTINE WRONG5 (N)
S-2 INTEGER N
S-3
S-4 !$OMP PARALLEL DEFAULT(SHARED)
S-5 !$OMP CRITICAL
S-6 CALL WORK(N,1)
S-7 ! incorrect nesting of barrier region in a critical region
S-8 !$OMP BARRIER
S-9 CALL WORK(N,2)
S-10 !$OMP END CRITICAL
S-11 !$OMP END PARALLEL
S-12 END SUBROUTINE WRONG5
```

Forran

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

C / C++

5 Example nesting\_restrict.6c

```
S-1 void work(int i, int j) {}
S-2 void wrong6(int n)
S-3 {
S-4 #pragma omp parallel
S-5 {
S-6 #pragma omp single
S-7 {
S-8 work(n, 0);
S-9 /* incorrect nesting of barrier region in a single region */
S-10 #pragma omp barrier
S-11 work(n, 1);
S-12 }
S-13 }
S-14 }
```

C / C++

Forran

1 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
```

Forran

1 CHAPTER 45

2

## The `omp_set_dynamic` and 3 `omp_set_num_threads` Routines

---

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

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

C / C++

15 Example `set_dynamic_nthrs.1c`

```
S-1 #include <omp.h>
S-2 #include <stdlib.h>
S-3
S-4 void do_by_16(float *x, int iam, int ipoints) {}
S-5
S-6 void dynthreads(float *x, int npoints)
S-7 {
S-8 int iam, ipoints;
S-9
S-10 omp_set_dynamic(0);
S-11 omp_set_num_threads(16);
S-12
S-13 #pragma omp parallel shared(x, npoints) private(iam, ipoints)
```

```

S-14 {
S-15 if (omp_get_num_threads() != 16)
S-16 abort();
S-17
S-18 iam = omp_get_thread_num();
S-19 ipoints = npoints/16;
S-20 do_by_16(x, iam, ipoints);
S-21 }
S-22 }
```

C / C++  
Fortran

1 Example set\_dynamic\_nthrs.If

```

S-1 SUBROUTINE DO_BY_16(X, IAM, IPOINTS)
S-2 REAL X(*)
S-3 INTEGER IAM, IPOINTS
S-4 END SUBROUTINE DO_BY_16
S-5
S-6 SUBROUTINE DYNTHREADS(X, NPOINTS)
S-7
S-8 INCLUDE "omp_lib.h" ! or USE OMP_LIB
S-9
S-10 INTEGER NPOINTS
S-11 REAL X(NPOINTS)
S-12
S-13 INTEGER IAM, IPOINTS
S-14
S-15 CALL OMP_SET_DYNAMIC(.FALSE.)
S-16 CALL OMP_SET_NUM_THREADS(16)
S-17
S-18 !$OMP PARALLEL SHARED(X,NPOINTS) PRIVATE(IAM, IPOINTS)
S-19
S-20 IF (OMP_GET_NUM_THREADS() .NE. 16) THEN
S-21 STOP
S-22 ENDIF
S-23
S-24 IAM = OMP_GET_THREAD_NUM()
S-25 IPOINTS = NPOINTS/16
S-26 CALL DO_BY_16(X, IAM, IPOINTS)
S-27
S-28 !$OMP END PARALLEL
S-29
S-30 END SUBROUTINE DYNTHREADS
```

Fortran

2 

## The `omp_get_num_threads` Routine

---

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

C / C++

6 Example `get_nthrds.1c`

```
S-1 #include <omp.h>
S-2 void work(int i);
S-3
S-4 void incorrect()
S-5 {
S-6 int np, i;
S-7
S-8 np = omp_get_num_threads(); /* misplaced */
S-9
S-10 #pragma omp parallel for schedule(static)
S-11 for (i=0; i < np; i++)
S-12 work(i);
S-13 }
```

C / C++

Forran

1 Example *get\_nthrds.If*

```
S-1 SUBROUTINE WORK(I)
S-2 INTEGER I
S-3 I = I + 1
S-4 END SUBROUTINE WORK
S-5
S-6 SUBROUTINE INCORRECT()
S-7 INCLUDE "omp_lib.h" ! or USE OMP_LIB
S-8 INTEGER I, NP
S-9
S-10 NP = OMP_GET_NUM_THREADS() !misplaced: will return 1
S-11 !$OMP PARALLEL DO SCHEDULE(STATIC)
S-12 DO I = 0, NP-1
S-13 CALL WORK(I)
S-14 ENDDO
S-15 !$OMP END PARALLEL DO
S-16 END SUBROUTINE INCORRECT
```

Forran

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

C / C++

4 Example *get\_nthrds.2c*

```
S-1 #include <omp.h>
S-2 void work(int i);
S-3
S-4 void correct()
S-5 {
S-6 int i;
S-7
S-8 #pragma omp parallel private(i)
S-9 {
S-10 i = omp_get_thread_num();
S-11 work(i);
S-12 }
S-13 }
```

C / C++

1       *Example get\_nthsrs.2f*

```
S-1 SUBROUTINE WORK(I)
S-2 INTEGER I
S-3
S-4 I = I + 1
S-5
S-6 END SUBROUTINE WORK
S-7
S-8 SUBROUTINE CORRECT()
S-9 INCLUDE "omp_lib.h" ! or USE OMP_LIB
S-10 INTEGER I
S-11
S-12 !$OMP PARALLEL PRIVATE(I)
S-13 I = OMP_GET_THREAD_NUM()
S-14 CALL WORK(I)
S-15 !$OMP END PARALLEL
S-16
S-17 END SUBROUTINE CORRECT
```

1 CHAPTER 47

2 **The `omp_init_lock` Routine**

---

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

C / C++

5 Example `init_lock.c`

```
S-1 #include <omp.h>
S-2
S-3 omp_lock_t *new_locks()
S-4 {
S-5 int i;
S-6 omp_lock_t *lock = new omp_lock_t[1000];
S-7
S-8 #pragma omp parallel for private(i)
S-9 for (i=0; i<1000; i++)
S-10 {
S-11 omp_init_lock(&lock[i]);
S-12 }
S-13 return lock;
S-14 }
```

C / C++

Forran

1 Example init\_lock.f90

```
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
```

Forran

1 CHAPTER 48

2 

## Ownership of Locks

---

3 Ownership of locks has changed since OpenMP 2.5. In OpenMP 2.5, locks are owned by threads;  
4 so a lock released by the **omp\_unset\_lock** routine must be owned by the same thread executing  
5 the routine. Beginning with OpenMP 3.0, locks are owned by task regions; so a lock released by the  
6 **omp\_unset\_lock** routine in a task region must be owned by the same task region.

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

C / C++ 

13 Example *lock\_owner.c*

```
S-1 #include <stdlib.h>
S-2 #include <stdio.h>
S-3 #include <omp.h>
S-4
S-5 int main()
S-6 {
S-7 int x;
S-8 omp_lock_t lck;
S-9
S-10 omp_init_lock (&lck);
S-11 omp_set_lock (&lck);
S-12 x = 0;
S-13
S-14 #pragma omp parallel shared (x)
S-15 {
S-16 #pragma omp master
```

```

S-17 {
S-18 x = x + 1;
S-19 omp_unset_lock (&lck);
S-20 }
S-21
S-22 /* Some more stuff. */
S-23 }
S-24 omp_destroy_lock (&lck);
S-25 return 0;
S-26 }
```

C / C++

Fortran

*Example lock\_owner.If*

```

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
```

Fortran

1 CHAPTER 49

2

## Simple Lock Routines

---

3 In the following example, the lock routines cause the threads to be idle while waiting for entry to  
4 the first critical section, but to do other work while waiting for entry to the second. The  
5 **omp\_set\_lock** function blocks, but the **omp\_test\_lock** function does not, allowing the work  
6 in **skip** to be done.

7 Note that the argument to the lock routines should have type **omp\_lock\_t**, and that there is no  
8 need to flush it.

C / C++

9 Example *simple\_lock.1c*

```
S-1 #include <stdio.h>
S-2 #include <omp.h>
S-3 void skip(int i) {}
S-4 void work(int i) {}
S-5 int main()
S-6 {
S-7 omp_lock_t lck;
S-8 int id;
S-9 omp_init_lock(&lck);
S-10
S-11 #pragma omp parallel shared(lck) private(id)
S-12 {
S-13 id = omp_get_thread_num();
S-14
S-15 omp_set_lock(&lck);
S-16 /* only one thread at a time can execute this printf */
S-17 printf("My thread id is %d.\n", id);
S-18 omp_unset_lock(&lck);
S-19
S-20 while (!omp_test_lock(&lck)) {
S-21 skip(id); /* we do not yet have the lock,
```

```

S-22 so we must do something else */
S-23 }
S-24
S-25 work(id); /* we now have the lock
S-26 and can do the work */
S-27
S-28 omp_unset_lock(&lck);
S-29 }
S-30 omp_destroy_lock(&lck);
S-31
S-32 return 0;
S-33 }
```

C / C++

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

Fortran

2 Example simple\_lock.If

```

S-1 SUBROUTINE SKIP(ID)
S-2 END SUBROUTINE SKIP
S-3
S-4 SUBROUTINE WORK(ID)
S-5 END SUBROUTINE WORK
S-6
S-7 PROGRAM SIMPLELOCK
S-8
S-9 INCLUDE "omp_lib.h" ! or USE OMP_LIB
S-10
S-11 INTEGER(OMP_LOCK_KIND) LCK
S-12 INTEGER ID
S-13
S-14 CALL OMP_INIT_LOCK(LCK)
S-15
S-16 !$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
S-17 ID = OMP_GET_THREAD_NUM()
S-18 CALL OMP_SET_LOCK(LCK)
S-19 PRINT *, 'My thread id is ', ID
S-20 CALL OMP_UNSET_LOCK(LCK)
S-21
S-22 DO WHILE (.NOT. OMP_TEST_LOCK(LCK))
S-23 CALL SKIP(ID) ! We do not yet have the lock
S-24 ! so we must do something else
S-25 END DO
S-26
S-27 CALL WORK(ID) ! We now have the lock
S-28 ! and can do the work
S-29
```

```
S-30 CALL OMP_UNSET_LOCK(LCK)
S-31
S-32 !$OMP END PARALLEL
S-33
S-34 CALL OMP_DESTROY_LOCK(LCK)
S-35
S-36 END PROGRAM SIMPLELOCK
```

Fortran

1 CHAPTER 50

2 **Nestable Lock Routines**

---

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

C / C++

5 Example *nestable\_lock.1c*

```
S-1 #include <omp.h>
S-2 typedef struct {
S-3 int a,b;
S-4 omp_nest_lock_t lck; } pair;
S-5
S-6 int work1();
S-7 int work2();
S-8 int work3();
S-9 void incr_a(pair *p, int a)
S-10 {
S-11 /* Called only from incr_pair, no need to lock. */
S-12 p->a += a;
S-13 }
S-14 void incr_b(pair *p, int b)
S-15 {
S-16 /* Called both from incr_pair and elsewhere, */
S-17 /* so need a nestable lock. */
S-18
S-19 omp_set_nest_lock(&p->lck);
S-20 p->b += b;
S-21 omp_unset_nest_lock(&p->lck);
S-22 }
S-23 void incr_pair(pair *p, int a, int b)
S-24 {
S-25 omp_set_nest_lock(&p->lck);
S-26 incr_a(p, a);
```

```

S-27 incr_b(p, b);
S-28 omp_unset_nest_lock(&p->lck);
S-29 }
S-30 void nestlock(pair *p)
S-31 {
S-32 #pragma omp parallel sections
S-33 {
S-34 #pragma omp section
S-35 incr_pair(p, work1(), work2());
S-36 #pragma omp section
S-37 incr_b(p, work3());
S-38 }
S-39 }
```

C / C++

Fortran

1

*Example nestable\_lock.If*

```

S-1 MODULE DATA
S-2 USE OMP_LIB, ONLY: OMP_NEST_LOCK_KIND
S-3 TYPE LOCKED_PAIR
S-4 INTEGER A
S-5 INTEGER B
S-6 INTEGER (OMP_NEST_LOCK_KIND) LCK
S-7 END TYPE
S-8 END MODULE DATA
S-9
S-10 SUBROUTINE INCR_A(P, A)
S-11 ! called only from INCR_PAIR, no need to lock
S-12 USE DATA
S-13 TYPE(LOCKED_PAIR) :: P
S-14 INTEGER A
S-15 P%A = P%A + A
S-16 END SUBROUTINE INCR_A
S-17
S-18 SUBROUTINE INCR_B(P, B)
S-19 ! called from both INCR_PAIR and elsewhere,
S-20 ! so we need a nestable lock
S-21 USE OMP_LIB ! or INCLUDE "omp_lib.h"
S-22 USE DATA
S-23 TYPE(LOCKED_PAIR) :: P
S-24 INTEGER B
S-25 CALL OMP_SET_NEST_LOCK(P%LCK)
S-26 P%B = P%B + B
S-27 CALL OMP_UNSET_NEST_LOCK(P%LCK)
S-28 END SUBROUTINE INCR_B
S-29
```

```

S-30 SUBROUTINE INCR_PAIR(P, A, B)
S-31 USE OMP_LIB ! or INCLUDE "omp_lib.h"
S-32 USE DATA
S-33 TYPE(LOCKED_PAIR) :: P
S-34 INTEGER A
S-35 INTEGER B
S-36
S-37 CALL OMP_SET_NEST_LOCK(P%LCK)
S-38 CALL INCR_A(P, A)
S-39 CALL INCR_B(P, B)
S-40 CALL OMP_UNSET_NEST_LOCK(P%LCK)
S-41 END SUBROUTINE INCR_PAIR
S-42
S-43 SUBROUTINE NESTLOCK(P)
S-44 USE OMP_LIB ! or INCLUDE "omp_lib.h"
S-45 USE DATA
S-46 TYPE(LOCKED_PAIR) :: P
S-47 INTEGER WORK1, WORK2, WORK3
S-48 EXTERNAL WORK1, WORK2, WORK3
S-49
S-50 !$OMP PARALLEL SECTIONS
S-51
S-52 !$OMP SECTION
S-53 CALL INCR_PAIR(P, WORK1(), WORK2())
S-54 !$OMP SECTION
S-55 CALL INCR_B(P, WORK3())
S-56 !$OMP END PARALLEL SECTIONS
S-57
S-58 END SUBROUTINE NESTLOCK

```

Fortran

1 CHAPTER 51

2 SIMD Constructs

---

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

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

C / C++

7 Example SIMD.1c

```
S-1 void star(double *a, double *b, double *c, int n, int *ioff)
S-2 {
S-3 int i;
S-4 #pragma omp simd
S-5 for (i = 0; i < n; i++)
S-6 a[i] *= b[i] * c[i+ *ioff];
S-7 }
```

C / C++

Fortran

8 Example SIMD.1f

```
S-1 subroutine star(a,b,c,n,ioff_ptr)
S-2 implicit none
S-3 double precision :: a(*),b(*),c(*)
S-4 integer :: n, i
S-5 integer, pointer :: ioff_ptr
S-6
S-7 !$omp simd
S-8 do i = 1,n
S-9 a(i) = a(i) * b(i) * c(i+ioff_ptr)
S-10 end do
S-11
S-12 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 the each vector operation has its own *tmp* variable.

### Example SIMD.2c

```
S-1 #include <stdio.h>
S-2
S-3 #pragma omp declare simd uniform(fact)
S-4 double add1(double a, double b, double fact)
S-5 {
S-6 double c;
S-7 c = a + b + fact;
S-8 return c;
S-9 }
S-10
S-11 #pragma omp declare simd uniform(a,b,fact) linear(i:1)
S-12 double add2(double *a, double *b, int i, double fact)
S-13 {
S-14 double c;
S-15 c = a[i] + b[i] + fact;
S-16 return c;
S-17 }
S-18
S-19 #pragma omp declare simd uniform(fact) linear(a,b:1)
S-20 double add3(double *a, double *b, double fact)
S-21 {
```

```

S-22 double c;
S-23 c = *a + *b + fact;
S-24 return c;
S-25 }
S-26
S-27 void work(double *a, double *b, int n)
S-28 {
S-29 int i;
S-30 double tmp;
S-31 #pragma omp simd private(tmp)
S-32 for (i = 0; i < n; i++) {
S-33 tmp = add1(a[i], b[i], 1.0);
S-34 a[i] = add2(a, b, i, 1.0) + tmp;
S-35 a[i] = add3(&a[i], &b[i], 1.0);
S-36 }
S-37 }
S-38
S-39 int main(){
S-40 int i;
S-41 const int N=32;
S-42 double a[N], b[N];
S-43
S-44 for (i=0; i<N; i++) {
S-45 a[i] = i; b[i] = N-i;
S-46 }
S-47
S-48 work(a, b, N);
S-49
S-50 for (i=0; i<N; i++) {
S-51 printf("%d %f\n", i, a[i]);
S-52 }
S-53
S-54 return 0;
S-55 }
```

C / C++

Fortran

1

*Example SIMD.2f*

```

S-1 program main
S-2 implicit none
S-3 integer, parameter :: N=32
S-4 integer :: i
S-5 double precision :: a(N), b(N)
S-6 do i = 1,N
S-7 a(i) = i-1
S-8 b(i) = N-(i-1)
```

```

S-9 end do
S-10 call work(a, b, N)
S-11 do i = 1,N
S-12 print*, i,a(i)
S-13 end do
S-14 end program
S-15
S-16 function add1(a,b,fact) result(c)
S-17 !$omp declare simd(add1) uniform(fact)
S-18 implicit none
S-19 double precision :: a,b,fact, c
S-20 c = a + b + fact
S-21 end function
S-22
S-23 function add2(a,b,i, fact) result(c)
S-24 !$omp declare simd(add2) uniform(a,b,fact) linear(i:1)
S-25 implicit none
S-26 integer :: i
S-27 double precision :: a(*),b(*),fact, c
S-28 c = a(i) + b(i) + fact
S-29 end function
S-30
S-31 subroutine work(a, b, n)
S-32 implicit none
S-33 double precision :: a(n),b(n), tmp
S-34 integer :: n, i
S-35 double precision, external :: add1, add2
S-36
S-37 !$omp simd private(tmp)
S-38 do i = 1,n
S-39 tmp = add1(a(i), b(i), 1.0d0)
S-40 a(i) = add2(a, b, i, 1.0d0) + tmp
S-41 a(i) = a(i) + b(i) + 1.0d0
S-42 end do
S-43 end subroutine

```



Fortran

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

C / C++

1      Example SIMD.3c

```
S-1 double work(double *a, double *b, int n)
S-2 {
S-3 int i;
S-4 double tmp, sum;
S-5 sum = 0.0;
S-6 #pragma omp simd private(tmp) reduction(+:sum)
S-7 for (i = 0; i < n; i++) {
S-8 tmp = a[i] + b[i];
S-9 sum += tmp;
S-10 }
S-11 return sum;
S-12 }
```

C / C++

Fortran

2      Example SIMD.3f

```
S-1 subroutine work(a, b, n, sum)
S-2 implicit none
S-3 integer :: i, n
S-4 double precision :: a(n), b(n), sum, tmp
S-5
S-6 sum = 0.0d0
S-7 !$omp simd private(tmp) reduction(+:sum)
S-8 do i = 1,n
S-9 tmp = a(i) + b(i)
S-10 sum = sum + tmp
S-11 end do
S-12
S-13 end subroutine work
```

Fortran

3      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.

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

---

C / C++

1        Example SIMD.4c

```
S-1 void work(float *b, int n, int m)
S-2 {
S-3 int i;
S-4 #pragma omp simd safelen(16)
S-5 for (i = m; i < n; i++)
S-6 b[i] = b[i-m] - 1.0f;
S-7 }
```

---

C / C++

---

Fortran

2        Example SIMD.4f

```
S-1 subroutine work(b, n, m)
S-2 implicit none
S-3 real :: b(n)
S-4 integer :: i,n,m
S-5
S-6 !$omp simd safelen(16)
S-7 do i = m+1, n
S-8 b(i) = b(i-m) - 1.0
S-9 end do
S-10 end subroutine work
```

---

Fortran

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

---

C / C++

6        Example SIMD.5c

```
S-1 void work(double **a, double **b, double **c, int n)
S-2 {
S-3 int i, j;
S-4 double tmp;
S-5 #pragma omp for simd collapse(2) private(tmp)
S-6 for (i = 0; i < n; i++) {
S-7 for (j = 0; j < n; j++) {
S-8 tmp = a[i][j] + b[i][j];
S-9 c[i][j] = tmp;
S-10 }
S-11 }
S-12 }
```

C / C++

Fortran

1

*Example SIMD.5f*

```
S-1 subroutine work(a, b, c, n)
S-2 implicit none
S-3 integer :: i,j,n
S-4 double precision :: a(n,n), b(n,n), c(n,n), tmp
S-5
S-6 !$omp for simd collapse(2) private(tmp)
S-7 do j = 1,n
S-8 do i = 1,n
S-9 tmp = a(i,j) + b(i,j)
S-10 c(i,j) = tmp
S-11 end do
S-12 end do
S-13
S-14 end subroutine work
```

Fortran

2

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

C / C++

7

*Example SIMD.6c*

```
S-1 #pragma omp declare simd linear(p:1) notinbranch
S-2 int foo(int *p){
S-3 *p = *p + 10;
S-4 return *p;
S-5 }
S-6
S-7 int myaddint(int *a, int *b, int n)
S-8 {
S-9 #pragma omp simd
S-10 for (int i=0; i<n; i++){
S-11 a[i] = foo(&b[i]); /* foo is not called under a condition */
S-12 }
S-13 return a[n-1];
S-14 }
S-15
S-16 #pragma omp declare simd linear(p:1) inbranch
S-17 float goo(float *p){
```

```

S-18 *p = *p + 18.5f;
S-19 return *p;
S-20 }
S-21
S-22 int myaddfloat(float *x, float *y, int n)
S-23 {
S-24 #pragma omp simd
S-25 for (int i=0; i<n; i++){
S-26 x[i] = (x[i] > y[i]) ? goo(&y[i]) : y[i];
S-27 /* goo is called under the condition (or within a branch) */
S-28 }
S-29 return x[n-1];
S-30 }
```

C / C++

Fortran

### 1 Example SIMD.6f

```

S-1 function foo(p) result(r)
S-2 !$omp declare simd(foo) notinbranch
S-3 implicit none
S-4 integer :: p, r
S-5 p = p + 10
S-6 r = p
S-7 end function foo
S-8
S-9 function myaddint(int *a, int *b, int n) result(r)
S-10 implicit none
S-11 integer :: a(*), b(*), n, r
S-12 integer :: i
S-13 integer, external :: foo
S-14
S-15 !$omp simd
S-16 do i=1, n
S-17 a(i) = foo(b(i)) ! foo is not called under a condition
S-18 end do
S-19 r = a(n)
S-20
S-21 end function myaddint
S-22
S-23 function goo(p) result(r)
S-24 !$omp declare simd(goo) inbranch
S-25 implicit none
S-26 real :: p, r
S-27 p = p + 18.5
S-28 r = p
S-29 end function goo
```

```

S-30
S-31 function myaddfloat(x, y, n) result(r)
S-32 implicit none
S-33 real :: x(*), y(*), r
S-34 integer :: n
S-35 integer :: i
S-36 real, external :: goo
S-37
S-38 !$omp simd
S-39 do i=1, n
S-40 if (x(i) > y(i)) then
S-41 x(i) = goo(y(i))
S-42 ! goo is called under the condition (or within a branch)
S-43 else
S-44 x(i) = y(i)
S-45 endif
S-46 end do
S-47
S-48 r = x(n)
S-49 end function myaddfloat

```

Fortran

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

C / C++

5 Example SIMD.7c

```

S-1 #include <stdio.h>
S-2 #include <stdlib.h>
S-3
S-4 #define N 45
S-5 int a[N], b[N], c[N];
S-6
S-7 #pragma omp declare simd inbranch
S-8 int fib(int n)
S-9 {
S-10 if (n <= 2)
S-11 return n;
S-12 else {
S-13 return fib(n-1) + fib(n-2);
S-14 }
S-15 }
S-16
S-17 int main(void)

```

```

S-18 {
S-19 int i;
S-20
S-21 #pragma omp simd
S-22 for (i=0; i < N; i++) b[i] = i;
S-23
S-24 #pragma omp simd
S-25 for (i=0; i < N; i++) {
S-26 a[i] = fib(b[i]);
S-27 }
S-28 printf("Done a[%d] = %d\n", N-1, a[N-1]);
S-29 return 0;
S-30 }

```

C / C++

Fortran

### Example SIMD.7f

```

S-1 program fibonacci
S-2 implicit none
S-3 integer,parameter :: N=45
S-4 integer :: a(0:N-1), b(0:N-1)
S-5 integer :: i
S-6 integer, external :: fib
S-7
S-8 !$omp simd
S-9 do i = 0,N-1
S-10 b(i) = i
S-11 end do
S-12
S-13 !$omp simd
S-14 do i=0,N-1
S-15 a(i) = fib(b(i))
S-16 end do
S-17
S-18 write(*,*) "Done a(", N-1, ") = ", a(N-1)
S-19 ! 44 1134903168
S-20 end program
S-21
S-22 recursive function fib(n) result(r)
S-23 !$omp declare simd(fib) inbranch
S-24 implicit none
S-25 integer :: n, r
S-26
S-27 if (n <= 2) then
S-28 r = n
S-29 else

```

```
S-30 r = fib(n-1) + fib(n-2)
S-31 endif
S-32
S-33 end function fib
```

Fortran

## 1 CHAPTER 52

## 2 target Construct

---

### 3 52.1 target Construct on parallel Construct

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

C / C++

6 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 }
```

C / C++

## Fortran

1 Example target.1f

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

## Fortran

## 2 52.2 target Construct with map Clause

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

## C / C++

6 Example target.2c

```
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 map(v1, v2, p)
S-9 #pragma omp parallel for
S-10 for (i=0; i<N; i++)
S-11 p[i] = v1[i] * v2[i];
S-12 output(p, N);
S-13 }
```

## C / C++

1      Example target.2f

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

## 2 52.3 map Clause with to/from map-types

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

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

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

## C / C++

1      Example target.3c

```
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 map(to: v1, v2) map(from: p)
S-9 #pragma omp parallel for
S-10 for (i=0; i<N; i++)
S-11 p[i] = v1[i] * v2[i];
S-12 output(p, N);
S-13 }
```

## C / C++

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

## Fortran

5      Example target.3f

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

## Fortran

## 52.4 map Clause with Array Sections

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

C / C++

*Example target.4c*

```
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 map(to: v1[0:N], v2[:N]) map(from: p[0:N])
S-8 #pragma omp parallel for
S-9 for (i=0; i<N; i++)
S-10 p[i] = v1[i] * v2[i];
S-11 output(p, N);
S-12 }
```

C / C++

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

Fortran

*Example target.4f*

```
S-1 module multis
S-2 contains
S-3 subroutine vec_mult(p,v1,v2,N)
S-4 real,pointer,dimension(:) :: p, v1, v2
S-5 integer :: N,i
S-6 call init(v1, v2, N)
S-7 !$omp target map(to: v1(1:N), v2(:N)) map(from: p(1:N))
S-8 !$omp parallel do
S-9 do i=1,N
S-10 p(i) = v1(i) * v2(i)
S-11 end do
S-12 !$omp end target
S-13 call output(p, N)
S-14 end subroutine
S-15 end module
```

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

5 Example target.4bf

```
S-1 module mults
S-2 contains
S-3 subroutine vec_mult(p,v1,v2,N)
S-4 real,dimension(*) :: p, v1, v2
S-5 integer :: N,i
S-6 call init(v1, v2, N)
S-7 !$omp target map(to: v1(1:N), v2(:N)) map(from: p(1:N))
S-8 !$omp parallel do
S-9 do i=1,N
S-10 p(i) = v1(i) * v2(i)
S-11 end do
S-12 call output(p, N)
S-13 !$omp end target
S-14 end subroutine
S-15 end module
```

6 52.5 target Construct with if Clause

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

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

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

1

*Example target.5c*

```

S-1 #define THRESHOLD1 1000000
S-2 #define THRESHOLD2 1000
S-3 extern void init(float*, float*, int);
S-4 extern void output(float*, int);
S-5 void vec_mult(float *p, float *v1, float *v2, int N)
S-6 {
S-7 int i;
S-8 init(v1, v2, N);
S-9 #pragma omp target if(N>THRESHOLD1) map(to: v1[0:N], v2[:N])\
S-10 map(from: p[0:N])
S-11 #pragma omp parallel for if(N>THRESHOLD2)
S-12 for (i=0; i<N; i++)
S-13 p[i] = v1[i] * v2[i];
S-14 output(p, N);
S-15 }
```

2

*Example target.5f*

```

S-1 module params
S-2 integer,parameter :: THRESHOLD1=1000000, THRESHOLD2=1000
S-3 end module
S-4 subroutine vec_mult(p, v1, v2, N)
S-5 use params
S-6 real :: p(N), v1(N), v2(N)
S-7 integer :: i
S-8 call init(v1, v2, N)
S-9 !$omp target if(N>THRESHOLD1) map(to: v1, v2) map(from: p)
S-10 !$omp parallel do if(N>THRESHOLD2)
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
```

1 CHAPTER 53

2 target data Construct

---

3 53.1 Simple target data Construct

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

C / C++

10 Example *target\_data.1c*

```
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 data map(to: v1[0:N], v2[:N]) map(from: p[0:N])
S-8 {
S-9 #pragma omp target
S-10 #pragma omp parallel for
S-11 for (i=0; i<N; i++)
S-12 p[i] = v1[i] * v2[i];
S-13 }
S-14 output(p, N);
S-15 }
```

## C / C++

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

## Fortran

3 Example target\_data.If

```
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 !$omp target data map(to: v1, v2) map(from: p)
S-6 !$omp target
S-7 !$omp parallel do
S-8 do i=1,N
S-9 p(i) = v1(i) * v2(i)
S-10 end do
S-11 !$omp end target
S-12 !$omp end target data
S-13 call output(p, N)
S-14 end subroutine
```

## Fortran

## 53.2 target data Region Enclosing Multiple target Regions

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

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

## C / C++

1 Example target\_data.2c

```
S-1 extern void init(float*, float*, int);
S-2 extern void init_again(float*, float*, int);
S-3 extern void output(float*, int);
S-4 void vec_mult(float *p, float *v1, float *v2, int N)
S-5 {
S-6 int i;
S-7 init(v1, v2, N);
S-8 #pragma omp target data map(from: p[0:N])
S-9 {
S-10 #pragma omp target map(to: v1[:N], v2[:N])
S-11 #pragma omp parallel for
S-12 for (i=0; i<N; i++)
S-13 p[i] = v1[i] * v2[i];
S-14 init_again(v1, v2, N);
S-15 #pragma omp target map(to: v1[:N], v2[:N])
S-16 #pragma omp parallel for
S-17 for (i=0; i<N; i++)
S-18 p[i] = p[i] + (v1[i] * v2[i]);
S-19 }
S-20 output(p, N);
S-21 }
```

## C / C++

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

## Fortran

6 Example target\_data.2f

```
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 !$omp target data map(from: p)
S-6 !$omp target map(to: v1, v2)
S-7 !$omp parallel do
S-8 do i=1,N
S-9 p(i) = v1(i) * v2(i)
S-10 end do
S-11 !$omp end target
S-12 call init_again(v1, v2, N)
```

```

S-13 !$omp target map(to: v1, v2)
S-14 !$omp parallel do
S-15 do i=1,N
S-16 p(i) = p(i) + v1(i) * v2(i)
S-17 end do
S-18 !$omp end target
S-19 !$omp end target data
S-20 call output(p, N)
S-21 end subroutine

```

Fortran

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

C / C++

*Example target\_data.3c*

```

S-1 #include <math.h>
S-2 #define COLS 100
S-3 void gramSchmidt(float Q[][COLS], const int rows)
S-4 {
S-5 int cols = COLS;
S-6 #pragma omp target data map(Q[0:rows] [0:cols])
S-7 for(int k=0; k < cols; k++)
S-8 {
S-9 double tmp = 0.0;
S-10 #pragma omp target
S-11 #pragma omp parallel for reduction(+:tmp)
S-12 for(int i=0; i < rows; i++)
S-13 tmp += (Q[i][k] * Q[i][k]);
S-14 tmp = 1/sqrt(tmp);
S-15 #pragma omp target
S-16 #pragma omp parallel for
S-17 for(int i=0; i < rows; i++)
S-18 Q[i][k] *= tmp;
S-19 }
S-20}

```

C / C++

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

---

Fortran

---

1    Example target\_data.f

```
S-1 subroutine gramSchmidt(Q,rows,cols)
S-2 integer :: rows,cols, i,k
S-3 double precision :: Q(rows,cols), tmp
S-4 !$omp target data map(Q)
S-5 do k=1,cols
S-6 tmp = 0.0d0
S-7 !$omp target
S-8 !$omp parallel do reduction(+:tmp)
S-9 do i=1,rows
S-10 tmp = tmp + (Q(i,k) * Q(i,k))
S-11 end do
S-12 !$omp end target
S-13 tmp = 1.0d0/sqrt(tmp)
S-14 !$omp target
S-15 !$omp parallel do
S-16 do i=1,rows
S-17 Q(i,k) = Q(i,k)*tmp
S-18 enddo
S-19 !$omp end target
S-20 end do
S-21 !$omp end target data
S-22 end subroutine
```

---

Fortran

---

2    53.3 target data Construct with Orphaned  
3    Call

4    The following two examples show how the **target data** construct maps variables to a device  
5    data environment. The **target data** construct's device data environment encloses the **target**  
6    construct's device data environment in the function **vec\_mult()**.

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

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

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

---

C / C++

---

9      Example *target\_data.4c*

```
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 }
```

---

C / C++

---

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

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

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

## Fortran

1 Example target\_data.4f

```
S-1 module multis
S-2 contains
S-3 subroutine foo(p0,v1,v2,N)
S-4 real,pointer,dimension(:) :: p0, v1, v2
S-5 integer :: N,i
S-6 call init(v1, v2, N)
S-7 !$omp target data map(to: v1, v2) map(from: p0)
S-8 call vec_mult(p0,v1,v2,N)
S-9 !$omp end target data
S-10 call output(p0, N)
S-11 end subroutine
S-12 subroutine vec_mult(p1,v3,v4,N)
S-13 real,pointer,dimension(:) :: p1, v3, v4
S-14 integer :: N,i
S-15 !$omp target map(to: v3, v4) map(from: p1)
S-16 !$omp parallel do
S-17 do i=1,N
S-18 p1(i) = v3(i) * v4(i)
S-19 end do
S-20 !$omp end target
S-21 end subroutine
S-22 end module
```

## Fortran

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

## C / C++

6 Example target\_data.5c

```
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 }
```

C / C++

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

Fortran

*Example target\_data.5f*

```

S-1 module my_mult
S-2 contains
S-3 subroutine foo(p0,v1,v2,N)
S-4 real,dimension(:) :: p0, v1, v2
S-5 integer :: N,i
S-6 call init(v1, v2, N)
S-7 !$omp target data map(to: v1, v2) map(from: p0)
S-8 call vec_mult(p0,v1,v2,N)
S-9 !$omp end target data
S-10 call output(p0, N)
S-11 end subroutine
S-12 subroutine vec_mult(p1,v3,v4,N)
S-13 real,dimension(:) :: p1, v3, v4
S-14 integer :: N,i
S-15 !$omp target map(to: v3, v4) map(from: p1)
S-16 !$omp parallel do
S-17 do i=1,N
S-18 p1(i) = v3(i) * v4(i)
S-19 end do
S-20 !$omp end target
S-21 end subroutine
S-22 end module
S-23 program main
S-24 use my_mult
S-25 integer, parameter :: N=1024
S-26 real,allocatable, dimension(:) :: p, v1, v2
S-27 allocate(p(N), v1(N), v2(N))
```

```
S-28 call foo(p,v1,v2,N)
S-29 end program
```

Fortran

## 53.4 target data Construct with if Clause

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

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

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

C / C++

*Example target\_data.6c*

```
S-1 #define THRESHOLD 1000000
S-2 extern void init(float*, float*, int);
S-3 extern void init_again(float*, float*, int);
S-4 extern void output(float*, int);
S-5 void vec_mult(float *p, float *v1, float *v2, int N)
S-6 {
S-7 int i;
S-8 init(v1, v2, N);
S-9 #pragma omp target data if(N>THRESHOLD) map(from: p[0:N])
S-10 {
S-11 #pragma omp target if (N>THRESHOLD) map(to: v1[:N], v2[:N])
S-12 #pragma omp parallel for
S-13 for (i=0; i<N; i++)
S-14 p[i] = v1[i] * v2[i];
S-15 init_again(v1, v2, N);
S-16 #pragma omp target if (N>THRESHOLD) map(to: v1[:N], v2[:N])
S-17 #pragma omp parallel for
S-18 for (i=0; i<N; i++)
S-19 p[i] = p[i] + (v1[i] * v2[i]);
S-20 }
S-21 output(p, N);
S-22 }
```

---

**C / C++**

---

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

---

**Fortran**

---

5    Example target\_data.f90

```
S-1 module params
S-2 integer,parameter :: THRESHOLD=1000000
S-3 end module
S-4 subroutine vec_mult(p, v1, v2, N)
S-5 use params
S-6 real :: p(N), v1(N), v2(N)
S-7 integer :: i
S-8 call init(v1, v2, N)
S-9 !$omp target data if(N>THRESHOLD) map(from: p)
S-10 !$omp target if(N>THRESHOLD) map(to: v1, v2)
S-11 !$omp parallel do
S-12 do i=1,N
S-13 p(i) = v1(i) * v2(i)
S-14 end do
S-15 !$omp end target
S-16 call init_again(v1, v2, N)
S-17 !$omp target if(N>THRESHOLD) map(to: v1, v2)
S-18 !$omp parallel do
S-19 do i=1,N
S-20 p(i) = p(i) + v1(i) * v2(i)
S-21 end do
S-22 !$omp end target
S-23 !$omp end target data
S-24 call output(p, N)
S-25 end subroutine
```

---

**Fortran**

---

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

## C / C++

1 Example target\_data.7c

```
S-1 #define THRESHOLD 1000000
S-2 extern void init(float*, float*, int);
S-3 extern void output(float*, int);
S-4 void vec_mult(float *p, float *v1, float *v2, int N)
S-5 {
S-6 int i;
S-7 init(v1, v2, N);
S-8 #pragma omp target data map(from: p[0:N])
S-9 {
S-10 #pragma omp target if (N>THRESHOLD) map(to: v1[:N], v2[:N])
S-11 #pragma omp parallel for
S-12 for (i=0; i<N; i++)
S-13 p[i] = v1[i] * v2[i];
S-14 /* UNDEFINED behavior if N<=THRESHOLD */
S-15 output(p, N);
S-16 }
```

## C / C++

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

## Fortran

9 Example target\_data.7f

```
S-1 module params
S-2 integer, parameter :: THRESHOLD=1000000
S-3 end module
S-4 subroutine vec_mult(p, v1, v2, N)
S-5 use params
S-6 real :: p(N), v1(N), v2(N)
S-7 integer :: i
S-8 call init(v1, v2, N)
S-9 !$omp target data map(from: p)
S-10 !$omp target if(N>THRESHOLD) map(to: v1, v2)
S-11 !$omp parallel do
S-12 do i=1,N
S-13 p(i) = v1(i) * v2(i)
S-14 end do
```

```
S-15 !$omp end target
S-16 !$omp end target data
S-17 call output(p, N) !*** UNDEFINED behavior if N<=THRESHOLD
S-18 end subroutine
```

Fortran

1    **CHAPTER 54**

2    **target update Construct**

---

3    **54.1 Simple target data and target update  
4    Constructs**

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

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

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

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

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

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

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

1

*Example target\_update.1c*

```

S-1 extern void init(float *, float *, int);
S-2 extern void init_again(float *, float *, int);
S-3 extern void output(float *, int);
S-4 void vec_mult(float *p, float *v1, float *v2, int N)
S-5 {
S-6 int i;
S-7 init(v1, v2, N);
S-8 #pragma omp target data map(to: v1[:N], v2[:N]) map(from: p[0:N])
S-9 {
S-10 #pragma omp target
S-11 #pragma omp parallel for
S-12 for (i=0; i<N; i++)
S-13 p[i] = v1[i] * v2[i];
S-14 init_again(v1, v2, N);
S-15 #pragma omp target update to(v1[:N], v2[:N])
S-16 #pragma omp target
S-17 #pragma omp parallel for
S-18 for (i=0; i<N; i++)
S-19 p[i] = p[i] + (v1[i] * v2[i]);
S-20 }
S-21 output(p, N);
S-22 }
```

2

*Example target\_update.1f*

```

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 !$omp target data map(to: v1, v2) map(from: p)
S-6 !$omp target
S-7 !$omp parallel do
S-8 do i=1,N
S-9 p(i) = v1(i) * v2(i)
S-10 end do
S-11 !$omp end target
S-12 call init_again(v1, v2, N)
S-13 !$omp target update to(v1, v2)
S-14 !$omp target
S-15 !$omp parallel do
S-16 do i=1,N
S-17 p(i) = p(i) + v1(i) * v2(i)
```

```
S-18 end do
S-19 !$omp end target
S-20 !$omp end target data
S-21 call output(p, N)
S-22 end subroutine
```

Fortran

## 54.2 target update Construct with if Clause

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

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

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

C / C++

Example *target\_update.2c*

```
S-1 extern void init(float *, float *, int);
S-2 extern int maybe_init_again(float *, int);
S-3 extern void output(float *, int);
S-4 void vec_mult(float *p, float *v1, float *v2, int N)
S-5 {
S-6 int i;
S-7 init(v1, v2, N);
S-8 #pragma omp target data map(to: v1[:N], v2[:N]) map(from: p[0:N])
S-9 {
S-10 int changed;
S-11 #pragma omp target
S-12 #pragma omp parallel for
S-13 for (i=0; i<N; i++)
S-14 p[i] = v1[i] * v2[i];
S-15 changed = maybe_init_again(v1, N);
S-16 #pragma omp target update if (changed) to(v1[:N])
S-17 changed = maybe_init_again(v2, N);
```

```

S-18 #pragma omp target update if (changed) to(v2[:N])
S-19 #pragma omp target
S-20 #pragma omp parallel for
S-21 for (i=0; i<N; i++)
S-22 p[i] = p[i] + (v1[i] * v2[i]);
S-23 }
S-24 output(p, N);
S-25 }
```

C / C++

Fortran

1 Example target\_update.2f

```

S-1 subroutine vec_mult(p, v1, v2, N)
S-2 interface
S-3 logical function maybe_init_again (v1, N)
S-4 real :: v1(N)
S-5 integer :: N
S-6 end function
S-7 end interface
S-8 real :: p(N), v1(N), v2(N)
S-9 integer :: i
S-10 logical :: changed
S-11 call init(v1, v2, N)
S-12 !$omp target data map(to: v1, v2) map(from: p)
S-13 !$omp target
S-14 !$omp parallel do
S-15 do i=1, N
S-16 p(i) = v1(i) * v2(i)
S-17 end do
S-18 !$omp end target
S-19 changed = maybe_init_again(v1, N)
S-20 !$omp target if(changed) update to(v1(:N))
S-21 changed = maybe_init_again(v2, N)
S-22 !$omp target if(changed) update to(v2(:N))
S-23 !$omp target
S-24 !$omp parallel do
S-25 do i=1, N
S-26 p(i) = p(i) + v1(i) * v2(i)
S-27 end do
S-28 !$omp end target
S-29 !$omp end target data
S-30 call output(p, N)
S-31 end subroutine
```

Fortran

1 CHAPTER 55

2 **declare target Construct**

---

3 **55.1 declare target and end declare target  
4 for a Function**

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

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

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

C / C++

13 Example *declare\_target.1c*

```
S-1 #pragma omp declare target
S-2 extern void fib(int N);
S-3 #pragma omp end declare target
S-4 #define THRESHOLD 1000000
S-5 void fib_wrapper(int n)
S-6 {
S-7 #pragma omp target if(n > THRESHOLD)
S-8 {
S-9 fib(n);
S-10 }
S-11 }
```

---

**C / C++**

---

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

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

---

**Fortran**

---

6      *Example declare\_target.If*

```
S-1 module module_fib
S-2 contains
S-3 subroutine fib(N)
S-4 integer :: N
S-5 !$omp declare target
S-6 !...
S-7 end subroutine
S-8 end module
S-9 module params
S-10 integer :: THRESHOLD=1000000
S-11 end module
S-12 program my_fib
S-13 use params
S-14 use module_fib
S-15 !$omp target if(N > THRESHOLD)
S-16 call fib(N)
S-17 !$omp end target
S-18 end program
```

---

**Fortran**

---

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

## Fortran

1 Example declare\_target.2f

```
S-1 program my_fib
S-2 integer :: N = 8
S-3 !$omp declare target(fib)
S-4 !$omp target
S-5 call fib(N)
S-6 !$omp end target
S-7 end program
S-8 subroutine fib(N)
S-9 integer :: N
S-10 !$omp declare target
S-11 print*, "hello from fib"
S-12 ...
S-13 end subroutine
```

## Fortran

## 2 55.2 declare target Construct for Class Type

### C++

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

7 Example declare\_target.2c

```
S-1 struct typeX
S-2 {
S-3 int a;
S-4 };
S-5 class typeY
S-6 {
S-7 int a;
S-8 public:
S-9 int foo() { return a^0x01; }
S-10 };
S-11 #pragma omp declare target
S-12 struct typeX varX; // ok
S-13 class typeY varY; // ok if varY.foo() not called on target device
```

```
S-14 #pragma omp end declare target
S-15 void foo()
S-16 {
S-17 #pragma omp target
S-18 {
S-19 varX.a = 100; // ok
S-20 varY.foo(); // error foo() is not available on a target device
S-21 }
S-22 }
```

C++

## 55.3 declare target and end declare target for Variables

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

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

C / C++

*Example declare\_target.3c*

```
S-1 #define N 1000
S-2 #pragma omp declare target
S-3 float p[N], v1[N], v2[N];
S-4 #pragma omp end declare target
S-5 extern void init(float *, float *, int);
S-6 extern void output(float *, int);
S-7 void vec_mult()
S-8 {
S-9 int i;
S-10 init(v1, v2, N);
S-11 #pragma omp target update to(v1, v2)
S-12 #pragma omp target
S-13 #pragma omp parallel for
```

```

S-14 for (i=0; i<N; i++)
S-15 p[i] = v1[i] * v2[i];
S-16 #pragma omp target update from(p)
S-17 output(p, N);
S-18 }
```

C / C++

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

Fortran

3 Example *declare\_target.f90*

```

S-1 module my_arrays
S-2 !$omp declare target (N, p, v1, v2)
S-3 integer, parameter :: N=1000
S-4 real :: p(N), v1(N), v2(N)
S-5 end module
S-6 subroutine vec_mult()
S-7 use my_arrays
S-8 integer :: i
S-9 call init(v1, v2, N);
S-10 !$omp target update to(v1, v2)
S-11 !$omp target
S-12 !$omp parallel do
S-13 do i = 1,N
S-14 p(i) = v1(i) * v2(i)
S-15 end do
S-16 !$omp end target
S-17 !$omp target update from (p)
S-18 call output(p, N)
S-19 end subroutine
```

Fortran

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

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

1      Example declare\_target.4c

```

S-1 #define N 10000
S-2 #pragma omp declare target
S-3 float Q[N][N];
S-4 float Pfun(const int i, const int k)
S-5 { return Q[i][k] * Q[k][i]; }
S-6 #pragma omp end declare target
S-7 float accum(int k)
S-8 {
S-9 float tmp = 0.0;
S-10 #pragma omp target update to(Q)
S-11 #pragma omp target
S-12 #pragma omp parallel for reduction(+:tmp)
S-13 for(int i=0; i < N; i++)
S-14 tmp += Pfun(i,k);
S-15 return tmp;
S-16 }
```

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

7      Example declare\_target.4f

```

S-1 module my_global_array
S-2 !$omp declare target (N,Q)
S-3 integer, parameter :: N=10
S-4 real :: Q(N,N)
S-5 contains
S-6 function Pfun(i,k)
S-7 !$omp declare target
S-8 real :: Pfun
S-9 integer,intent(in) :: i,k
S-10 Pfun=(Q(i,k) * Q(k,i))
S-11 end function
S-12 end module
S-13 function accum(k) result(tmp)
S-14 use my_global_array
S-15 real :: tmp
S-16 integer :: i, k
S-17 tmp = 0.0e0
```

```
S-18 !$omp target
S-19 !$omp parallel do reduction(+:tmp)
S-20 do i=1,N
S-21 tmp = tmp + Pfun(k,i)
S-22 end do
S-23 !$omp end target
S-24 end function
```

Fortran

## 1 55.4 declare target and end declare target 2 with declare simd

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

C / C++

7 Example *declare\_target.5c*

```
S-1 #define N 10000
S-2 #define M 1024
S-3 #pragma omp declare target
S-4 float Q[N][N];
S-5 #pragma omp declare simd uniform(i) linear(k) notinbranch
S-6 float P(const int i, const int k)
S-7 {
S-8 return Q[i][k] * Q[k][i];
S-9 }
S-10 #pragma omp end declare target
S-11 float accum(void)
S-12 {
S-13 float tmp = 0.0;
S-14 int i, k;
S-15 #pragma omp target
S-16 #pragma omp parallel for reduction(+:tmp)
S-17 for (i=0; i < N; i++) {
S-18 float tmp1 = 0.0;
S-19 #pragma omp simd reduction(+:tmp1)
S-20 for (k=0; k < M; k++) {
S-21 tmp1 += P(i,k);
```

```

S-22 }
S-23 tmp += tmp1;
S-24 }
S-25 return tmp;
S-26 }
```

C / C++

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

Fortran

*Example declare\_target.f90*

```

S-1 module my_global_array
S-2 !$omp declare target (N,Q)
S-3 integer, parameter :: N=10000, M=1024
S-4 real :: Q(N,N)
S-5 contains
S-6 function P(k,i)
S-7 !$omp declare simd uniform(i) linear(k) notinbranch
S-8 !$omp declare target
S-9 real :: P
S-10 integer,intent(in) :: k,i
S-11 P=(Q(k,i) * Q(i,k))
S-12 end function
S-13 end module
S-14 function accum() result(tmp)
S-15 use my_global_array
S-16 real :: tmp, tmp1
S-17 integer :: i
S-18 tmp = 0.0e0
S-19 !$omp target
S-20 !$omp parallel do private(tmp1) reduction(+:tmp)
S-21 do i=1,N
S-22 tmp1 = 0.0e0
S-23 !$omp simd reduction(+:tmp1)
S-24 do k = 1,M
S-25 tmp1 = tmp1 + P(k,i)
S-26 end do
S-27 tmp = tmp + tmp1
S-28 end do
S-29 !$omp end target
S-30 end function
```

Fortran

1 CHAPTER 56

2 teams Constructs

---

3 56.1 target and teams Constructs with `omp_get_num_teams`  
4 and `omp_get_team_num` Routines

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

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

C / C++

12 Example `teams.1c`

```
S-1 #include <stdlib.h>
S-2 #include <omp.h>
S-3 float dotprod(float B[], float C[], int N)
S-4 {
S-5 float sum0 = 0.0;
S-6 float sum1 = 0.0;
S-7 #pragma omp target map(to: B[:N], C[:N])
S-8 #pragma omp teams num_teams(2)
S-9 {
S-10 int i;
S-11 if (omp_get_num_teams() != 2)
S-12 abort();
S-13 if (omp_get_team_num() == 0)
S-14 {
S-15 #pragma omp parallel for reduction(+:sum0)
```

```

S-16 for (i=0; i<N/2; i++)
S-17 sum0 += B[i] * C[i];
S-18 }
S-19 else if (omp_get_team_num() == 1)
S-20 {
S-21 #pragma omp parallel for reduction(+:sum1)
S-22 for (i=N/2; i<N; i++)
S-23 sum1 += B[i] * C[i];
S-24 }
S-25 }
S-26 return sum0 + sum1;
S-27 }
```

C / C++

Fortran

1

*Example teams.If*

```

S-1 function dotprod(B,C,N) result(sum)
S-2 use omp_lib, ONLY : omp_get_num_teams, omp_get_team_num
S-3 real :: B(N), C(N), sum,sum0, sum1
S-4 integer :: N, i
S-5 sum0 = 0.0e0
S-6 sum1 = 0.0e0
S-7 !$omp target map(to: B, C)
S-8 !$omp teams num_teams(2)
S-9 if (omp_get_num_teams() /= 2) stop "2 teams required"
S-10 if (omp_get_team_num() == 0) then
S-11 !$omp parallel do reduction(+:sum0)
S-12 do i=1,N/2
S-13 sum0 = sum0 + B(i) * C(i)
S-14 end do
S-15 else if (omp_get_team_num() == 1) then
S-16 !$omp parallel do reduction(+:sum1)
S-17 do i=N/2+1,N
S-18 sum1 = sum1 + B(i) * C(i)
S-19 end do
S-20 end if
S-21 !$omp end teams
S-22 !$omp end target
S-23 sum = sum0 + sum1
S-24 end function
```

Fortran

## 1 56.2 target, teams, and distribute Constructs

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

6 The number of teams in the league is less than or equal to the variable *num\_blocks*. Each team in  
7 the league has a number of threads less than or equal to the variable *block\_threads*. The iterations  
8 in the outer loop are distributed among the master threads of each team.

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

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

C / C++

18 Example teams.2c

```
S-1 float dotprod(float B[], float C[], int N, int block_size,
S-2 int num_teams, int block_threads)
S-3 {
S-4 float sum = 0;
S-5 int i, i0;
S-6 #pragma omp target map(to: B[0:N], C[0:N])
S-7 #pragma omp teams num_teams(num_teams) thread_limit(block_threads) \
S-8 reduction(+:sum)
S-9 #pragma omp distribute
S-10 for (i0=0; i0<N; i0 += block_size)
S-11 #pragma omp parallel for reduction(+:sum)
S-12 for (i=i0; i < min(i0+block_size,N); i++)
S-13 sum += B[i] * C[i];
S-14 return sum;
S-15 }
```

C / C++

1      Example teams.2f

```

S-1 function dotprod(B,C,N, block_size, num_teams, block_threads) result(sum)
S-2 implicit none
S-3 real :: B(N), C(N), sum
S-4 integer :: N, block_size, num_teams, block_threads, i, i0
S-5 sum = 0.0e0
S-6 !$omp target map(to: B, C)
S-7 !$omp teams num_teams(num_teams) thread_limit(block_threads) &
S-8 !$omp& reduction(+:sum)
S-9 !$omp distribute
S-10 do i0=1,N, block_size
S-11 !$omp parallel do reduction(+:sum)
S-12 do i = i0, min(i0+block_size,N)
S-13 sum = sum + B(i) * C(i)
S-14 end do
S-15 end do
S-16 !$omp end teams
S-17 !$omp end target
S-18 end function

```

## 2 56.3 target teams, and Distribute Parallel 3 Loop Constructs

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

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

C / C++

1 Example teams.3c

```
S-1 float dotprod(float B[], float C[], int N)
S-2 {
S-3 float sum = 0;
S-4 int i;
S-5 #pragma omp target teams map(to: B[0:N], C[0:N])
S-6 #pragma omp distribute parallel for reduction(+:sum)
S-7 for (i=0; i<N; i++)
S-8 sum += B[i] * C[i];
S-9 return sum;
S-10 }
```

C / C++

Fortran

2 Example teams.3f

```
S-1 function dotprod(B,C,N) result(sum)
S-2 real :: B(N), C(N), sum
S-3 integer :: N, i
S-4 sum = 0.0e0
S-5 !$omp target teams map(to: B, C)
S-6 !$omp distribute parallel do reduction(+:sum)
S-7 do i = 1,N
S-8 sum = sum + B(i) * C(i)
S-9 end do
S-10 !$omp end teams
S-11 !$omp end target
S-12 end function
```

Fortran

## 1 56.4 target teams and Distribute Parallel Loop 2 Constructs with Scheduling Clauses

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

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

9 The **dist\_schedule** clause on the distribute parallel loop construct indicates that loop iterations  
10 are distributed to the master thread of each team in chunks of 1024 iterations.

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

C / C++

13 Example teams.4c

```
S-1 #define N 1024*1024
S-2 float dotprod(float B[], float C[])
S-3 {
S-4 float sum = 0;
S-5 int i;
S-6 #pragma omp target map(to: B[0:N], C[0:N])
S-7 #pragma omp teams num_teams(8) thread_limit(16)
S-8 #pragma omp distribute parallel for reduction(+:sum) \
S-9 dist_schedule(static, 1024) schedule(static, 64)
S-10 for (i=0; i<N; i++)
S-11 sum += B[i] * C[i];
S-12 return sum;
S-13 }
```

C / C++

1      Example teams.4f

```

S-1 module arrays
S-2 integer,parameter :: N=1024*1024
S-3 real :: B(N), C(N)
S-4 end module
S-5 function dotprod() result(sum)
S-6 use arrays
S-7 real :: sum
S-8 integer :: i
S-9 sum = 0.0e0
S-10 !$omp target map(to: B, C)
S-11 !$omp teams num_teams(8) thread_limit(16)
S-12 !$omp distribute parallel do reduction(:sum) &
S-13 !$omp& dist_schedule(static, 1024) schedule(static, 64)
S-14 do i = 1,N
S-15 sum = sum + B(i) * C(i)
S-16 end do
S-17 !$omp end teams
S-18 !$omp end target
S-19 end function

```

## 2 56.5 target teams and distribute simd Constructs

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

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

---

C / C++

1      *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 }
```

---

C / C++

---

Fortran

2      *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
```

---

Fortran

## 1 56.6 target teams and Distribute Parallel Loop 2 SIMD Constructs

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

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

C / C++

8 Example teams.6c

```
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 parallel for simd
S-9 for (i=0; i<N; i++)
S-10 p[i] = v1[i] * v2[i];
S-11 output(p, N);
S-12 }
```

C / C++

Fortran

9 Example teams.6f

```
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 parallel do 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
```

Fortran

2  
3 

## Asynchronous Execution of a target Region Using Tasks

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

---

C / C++

10 *Example async\_target.1c*

```
S-1 #pragma omp declare target
S-2 float F(float);
S-3 #pragma omp end declare target
S-4 #define N 1000000000
S-5 #define CHUNKSZ 1000000
S-6 void init(float *, int);
S-7 float Z[N];
S-8 void pipedF()
S-9 {
S-10 int C, i;
S-11 init(Z, N);
S-12 for (C=0; C<N; C+=CHUNKSZ)
S-13 {
S-14 #pragma omp task
S-15 #pragma omp target map(Z[C:CHUNKSZ])
S-16 #pragma omp parallel for
S-17 for (i=0; i<CHUNKSZ; i++)
S-18 Z[i] = F(Z[i]);
S-19 }
S-20 #pragma omp taskwait
S-21 }
```

---

**C / C++**

---

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

---

**Fortran**

---

3      Example *async\_target.F*

```
S-1 module parameters
S-2 integer, parameter :: N=1000000000, CHUNKSZ=1000000
S-3 end module
S-4 subroutine pipedF()
S-5 use parameters, ONLY: N, CHUNKSZ
S-6 integer :: C, i
S-7 real :: z(N)
S-8
S-9 interface
S-10 function F(z)
S-11 !$omp declare target
S-12 real, intent(IN) ::z
S-13 real ::F
S-14 end function F
S-15 end interface
S-16
S-17 call init(z,N)
S-18
S-19 do C=1,N,CHUNKSZ
S-20
S-21 !$omp task
S-22 !$omp target map(z(C:C+CHUNKSZ-1))
S-23 !$omp parallel do
S-24 do i=C,C+CHUNKSZ-1
S-25 z(i) = F(z(i))
S-26 end do
S-27 !$omp end target
S-28 !$omp end task
S-29
S-30 end do
S-31 !$omp taskwait
S-32 print*, z
S-33
S-34 end subroutine pipedF
```

---

**Fortran**

---

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

```

1 Example async_target.2c

S-1 #include <stdlib.h>
S-2 #include <omp.h>
S-3 #pragma omp declare target
S-4 extern void init(float *, float *, int);
S-5 #pragma omp end declare target
S-6 extern void foo();
S-7 extern void output(float *, int);
S-8 void vec_mult(float *p, int N, int dev)
S-9 {
S-10 float *v1, *v2;
S-11 int i;
S-12 #pragma omp task shared(v1, v2) depend(out: v1, v2)
S-13 #pragma omp target device(dev) map(v1, v2)
S-14 {
S-15 // check whether on device dev
S-16 if (omp_is_initial_device())
S-17 abort();
S-18 v1 = malloc(N*sizeof(float));
S-19 v2 = malloc(N*sizeof(float));
S-20 init(v1, v2, N);
S-21 }
S-22 foo(); // execute other work asynchronously
S-23 #pragma omp task shared(v1, v2, p) depend(in: v1, v2)
S-24 #pragma omp target device(dev) map(to: v1, v2) map(from: p[0:N])
S-25 {
S-26 // check whether on device dev
S-27 if (omp_is_initial_device())
S-28 abort();
S-29 #pragma omp parallel for
S-30 for (i=0; i<N; i++)
S-31 p[i] = v1[i] * v2[i];
S-32 free(v1);
S-33 free(v2);
S-34 }
S-35 #pragma omp taskwait
S-36 output(p, N);
S-37 }
```

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

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

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

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



### Fortran

18 Example *async\_target.2f*

```
S-1 subroutine mult(p, N, idev)
S-2 use omp_lib, ONLY: omp_is_initial_device
S-3 real :: p(N)
S-4 real,allocatable :: v1(:), v2(:)
S-5 integer :: i, idev
S-6 !$omp declare target (init)
S-7
S-8 !$omp task shared(v1,v2) depend(out: N)
S-9 !$omp target device(idev) map(v1,v2)
S-10 if(omp_is_initial_device()) &
S-11 stop "not executing on target device"
S-12 allocate(v1(N), v2(N))
S-13 call init(v1,v2,N)
S-14 !$omp end target
S-15 !$omp end task
S-16
S-17 call foo() ! execute other work asynchronously
S-18
S-19 !$omp task shared(v1,v2,p) depend(in: N)
S-20 !$omp target device(idev) map(to: v1,v2) map(from: p)
S-21 if(omp_is_initial_device()) &
S-22 stop "not executing on target device"
S-23 !$omp parallel do
S-24 do i = 1,N
```

```
S-25 p(i) = v1(i) * v2(i)
S-26 end do
S-27 deallocate(v1,v2)
S-28
S-29 !$omp end target
S-30 !$omp end task
S-31
S-32 !$omp taskwait
S-33 call output(p, N)
S-34
S-35 end subroutine
```

Fortran

1 CHAPTER 58

2 **Array Sections in Device Constructs**

---

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

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

C / C++

7 Example `array_sections.1c`

```
S-1 void foo ()
S-2 {
S-3 int A[30];
S-4 #pragma omp target data map(A[0:4])
S-5 {
S-6 /* Cannot map distinct parts of the same array */
S-7 #pragma omp target map(A[7:20])
S-8 {
S-9 A[2] = 0;
S-10 }
S-11 }
S-12 }
```

C / C++

Forran

1      Example array\_sections.If

```
S-1 subroutine foo()
S-2 integer :: A(30)
S-3 A = 1
S-4 !$omp target data map(A(1:4))
S-5 ! Cannot map distinct parts of the same array
S-6 !$omp target map(A(8:27))
S-7 A(3) = 0
S-8 !$omp end target map
S-9 !$omp end target data
S-10 end subroutine
```

Forran

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

C / C++

4      Example array\_sections.2c

```
S-1 void foo ()
S-2 {
S-3 int A[30], *p;
S-4 #pragma omp target data map(A[0:4])
S-5 {
S-6 p = &A[0];
S-7 /* invalid because p[3] and A[3] are the same
S-8 * location on the host but the array section
S-9 * specified via p[...] is not a subset of A[0:4] */
S-10 #pragma omp target map(p[3:20])
S-11 {
S-12 A[2] = 0;
S-13 p[8] = 0;
S-14 }
S-15 }
S-16 }
```

C / C++

Forran

1 Example array\_sections.2f

```
S-1 subroutine foo()
S-2 integer,target :: A(30)
S-3 integer,pointer :: p(:)
S-4 A=1
S-5 !$omp target data map(A(1:4))
S-6 p=>A
S-7 ! invalid because p(4) and A(4) are the same
S-8 ! location on the host but the array section
S-9 ! specified via p(...) is not a subset of A(1:4)
S-10 !$omp target map(p(4:23))
S-11 A(3) = 0
S-12 p(9) = 0
S-13 !$omp end target
S-14 !$omp end target data
S-15 end subroutine
```

Forran

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

C / C++

4 Example array\_sections.3c

```
S-1 void foo ()
S-2 {
S-3 int A[30], *p;
S-4 #pragma omp target data map(A[0:4])
S-5 {
S-6 p = &A[0];
S-7 #pragma omp target map(p[7:20])
S-8 {
S-9 A[2] = 0;
S-10 p[8] = 0;
S-11 }
S-12 }
S-13 }
```

C / C++



## Fortran

1       *Example array\_sections.3f*

```
S-1 subroutine foo()
S-2 integer,target :: A(30)
S-3 integer,pointer :: p(:)
S-4 !$omp target data map(A(1:4))
S-5 p=>A
S-6 !$omp target map(p(8:27))
S-7 A(3) = 0
S-8 p(9) = 0
S-9 !$omp end target map
S-10 !$omp end target data
S-11 end subroutine
```



## Fortran

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



## C / C++

4       *Example array\_sections.4c*

```
S-1 void foo ()
S-2 {
S-3 int A[30], *p;
S-4 #pragma omp target data map(A[0:10])
S-5 {
S-6 p = &A[0];
S-7 #pragma omp target map(p[3:7])
S-8 {
S-9 A[2] = 0;
S-10 p[8] = 0;
S-11 A[8] = 1;
S-12 }
S-13 }
S-14 }
```



## C / C++

Forran

1      *Example array\_sections.4f*

```
S-1 subroutine foo()
S-2 integer,target :: A(30)
S-3 integer,pointer :: p(:)
S-4 !$omp target data map(A(1:10))
S-5 p=>A
S-6 !$omp target map(p(4:10))
S-7 A(3) = 0
S-8 p(9) = 0
S-9 A(9) = 1
S-10 !$omp end target
S-11 !$omp end target data
S-12 end subroutine
```

Forran

## 1 CHAPTER 59

# 2 Device Routines

---

### 3 59.1 `omp_is_initial_device` Routine

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

C / C++

7 Example `device.1c`

```
S-1 #include <stdio.h>
S-2 #include <omp.h>
S-3 #pragma omp declare target
S-4 void vec_mult(float *p, float *v1, float *v2, int N);
S-5 extern float *p, *v1, *v2;
S-6 extern int N;
S-7 #pragma omp end declare target
S-8 extern void init_vars(float *, float *, int);
S-9 extern void output(float *, int);
S-10 void foo()
S-11 {
S-12 init_vars(v1, v2, N);
S-13 #pragma omp target device(42) map(p[:N], v1[:N], v2[:N])
S-14 {
S-15 vec_mult(p, v1, v2, N);
S-16 }
S-17 output(p, N);
S-18 }
S-19 void vec_mult(float *p, float *v1, float *v2, int N)
S-20 {
S-21 int i;
S-22 int nthreads;
```

```

S-23 if (!omp_is_initial_device())
S-24 {
S-25 printf("1024 threads on target device\n");
S-26 nthreads = 1024;
S-27 }
S-28 else
S-29 {
S-30 printf("8 threads on initial device\n");
S-31 nthreads = 8;
S-32 }
S-33 #pragma omp parallel for private(i) num_threads(nthreads);
S-34 for (i=0; i<N; i++)
S-35 p[i] = v1[i] * v2[i];
S-36 }
```

C / C++

Fortran

1

*Example device.If*

```

S-1 module params
S-2 integer,parameter :: N=1024
S-3 end module params
S-4 module vmult
S-5 contains
S-6 subroutine vec_mult(p, v1, v2, N)
S-7 use omp_lib, ONLY : omp_is_initial_device
S-8 !$omp declare target
S-9 real :: p(N), v1(N), v2(N)
S-10 integer :: i, nthreads, N
S-11 if (.not. omp_is_initial_device()) then
S-12 print*, "1024 threads on target device"
S-13 nthreads = 1024
S-14 else
S-15 print*, "8 threads on initial device"
S-16 nthreads = 8
S-17 endif
S-18 !$omp parallel do private(i) num_threads(nthreads)
S-19 do i = 1,N
S-20 p(i) = v1(i) * v2(i)
S-21 end do
S-22 end subroutine vec_mult
S-23 end module vmult
S-24 program prog_vec_mult
S-25 use params
S-26 use vmult
S-27 real :: p(N), v1(N), v2(N)
S-28 call init(v1,v2,N)
```

```
S-29 !$omp target device(42) map(p, v1, v2)
S-30 call vec_mult(p, v1, v2, N)
S-31 !$omp end target
S-32 call output(p, N)
S-33 end program
```

Fortran

## 59.2 `omp_get_num_devices` Routine

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

C / C++

*Example device.2c*

```
S-1 #include <omp.h>
S-2 extern void init(float *, float *, int);
S-3 extern void output(float *, int);
S-4 void vec_mult(float *p, float *v1, float *v2, int N)
S-5 {
S-6 int i;
S-7 init(v1, v2, N);
S-8 int ndev = omp_get_num_devices();
S-9 int do_offload = (ndev>0 && N>1000000);
S-10 #pragma omp target if(do_offload) map(to: v1[0:N], v2[:N]) map(from: p[0:N])
S-11 #pragma omp parallel for if(N>1000) private(i)
S-12 for (i=0; i<N; i++)
S-13 p[i] = v1[i] * v2[i];
S-14 output(p, N);
S-15 }
```

C / C++

## Fortran

1 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
```

## Fortran

## 2 59.3 **omp\_set\_default\_device** and 3 **omp\_get\_default\_device** Routines

4 The following example shows how the **omp\_set\_default\_device** and  
5 **omp\_get\_default\_device** runtime library routines can be used to set the default device and  
6 determine the default device respectively.

## C / C++

7 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 }
```

1

*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

## 1 CHAPTER 60

# 2 Fortran ASSOCIATE Construct

---

### Fortran

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

9 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
```

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

12 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
```

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

7       *Example associate.3f*

```
S-1 program example
S-2 integer :: v
S-3 v = 15
S-4 associate(u => v)
S-5 !$omp parallel private(v)
S-6 v = -1
S-7 print *, v ! private v=-1
S-8 print *, u ! original v=15
S-9 !$omp end parallel
S-10 end associate
S-11 end program
```

Fortran

1 APPENDIX A

2

## Document Revision History

---

3

### A.1 Changes from 4.0.1 to 4.0.2

- 4
- Names of examples were changed from numbers to mnemonics

5

  - Added SIMD examples (Section 51 on page 182)

6

  - Applied miscellaneous fixes in several source codes

7

  - Added the revision history

8

### A.2 Changes from 4.0 to 4.0.1

9       Added the following new examples:

- 10
  - the `proc_bind` clause (Section 8 on page 22)
  - the `taskgroup` construct (Section 18 on page 73)

12

### A.3 Changes from 3.1 to 4.0

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

15       Version 4.0 added the following new examples:

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