OpenMP Application Programming Interface

Examples

Version 4.5.0 – November 2016

Source codes for OpenMP 4.5.0 Examples can be downloaded from github.

Copyright © 1997-2016 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.
Contents

Introduction 1

Examples 2

1. Parallel Execution 3
   1.1. A Simple Parallel Loop ................................. 5
   1.2. The parallel Construct ................................. 6
   1.3. Controlling the Number of Threads on Multiple Nesting Levels ......... 8
   1.4. Interaction Between the num_threads Clause and omp_set_dynamic .... 11
   1.5. Fortran Restrictions on the do Construct ..................... 13
   1.6. The nowait Clause ....................................... 15
   1.7. The collapse Clause ..................................... 18
   1.8. linear Clause in Loop Constructs .......................... 22
   1.9. The parallel sections Construct .......................... 24
   1.10. The firstprivate Clause and the sections Construct .............. 25
   1.11. The single Construct .................................... 27
   1.12. The workshare Construct ................................ 29
   1.13. The master Construct .................................... 33
   1.14. Parallel Random Access Iterator Loop ........................ 35
   1.15. The omp_set_dynamic and omp_set_num_threads Routines ......... 36
   1.16. The omp_get_num_threads Routine ........................ 38

2. OpenMP Affinity 40
   2.1. The proc_bind Clause .................................... 42
   2.1.1. Spread Affinity Policy .................................. 42
   2.1.2. Close Affinity Policy ................................... 44
   2.1.3. Master Affinity Policy .................................. 47
2.2. Affinity Query Functions ................................................. 48

3. Tasking 51
3.1. The task and taskwait Constructs .................................. 52
3.2. Task Priority ................................................................. 71
3.3. Task Dependences .......................................................... 73
   3.3.1. Flow Dependence ..................................................... 73
   3.3.2. Anti-dependence .................................................... 74
   3.3.3. Output Dependence ................................................ 75
   3.3.4. Concurrent Execution with Dependences ....................... 76
   3.3.5. Matrix multiplication .............................................. 78
3.4. The taskgroup Construct ................................................. 80
3.5. The taskyield Construct .................................................. 83
3.6. The taskloop Construct ................................................... 85

4. Devices 87
4.1. target Construct ........................................................... 88
   4.1.1. target Construct on parallel Construct ......................... 88
   4.1.2. target Construct with map Clause ................................ 89
   4.1.3. map Clause with to/from map-types .............................. 90
   4.1.4. map Clause with Array Sections .................................. 91
   4.1.5. target Construct with if Clause ................................ 93
4.2. target data Construct ..................................................... 96
   4.2.1. Simple target data Construct ..................................... 96
   4.2.2. target data Region Enclosing Multiple target Regions ....... 97
   4.2.3. target data Construct with Orphaned Call ...................... 101
   4.2.4. target data Construct with if Clause ............................ 104
4.3. target enter data and target exit data Constructs .................. 108
4.4. target update Construct ................................................ 111
   4.4.1. Simple target data and target update Constructs .............. 111
   4.4.2. target update Construct with if Clause ......................... 113
4.5. declare target Construct ................................................ 115
   4.5.1. declare target and end declare target for a Function .......... 115
   4.5.2. declare target Construct for Class Type ....................... 117
4.5.3. \texttt{declare target} and \texttt{end declare target} for Variables \hspace{1cm} 117
4.5.4. \texttt{declare target} and \texttt{end declare target} with \texttt{declare simd} \hspace{1cm} 120
4.5.5. \texttt{declare target} Directive with \texttt{link} Clause \hspace{1cm} 123

4.6. \texttt{teams} Constructs \hspace{1cm} 126
4.6.1. \texttt{target} and \texttt{teams} Constructs with \texttt{omp_get_num_teams} and \texttt{omp_get_team_num} Routines \hspace{1cm} 126
4.6.2. \texttt{target}, \texttt{teams}, and \texttt{distribute} Constructs \hspace{1cm} 128
4.6.3. \texttt{target teams}, and Distribute Parallel Loop Constructs \hspace{1cm} 129
4.6.4. \texttt{target teams} and Distribute Parallel Loop Constructs with Scheduling Clauses \hspace{1cm} 131
4.6.5. \texttt{target teams} and \texttt{distribute simd} Constructs \hspace{1cm} 132
4.6.6. \texttt{target teams} and Distribute Parallel Loop SIMD Constructs \hspace{1cm} 134

4.7. Asynchronous \texttt{target} Execution and Dependences \hspace{1cm} 135
4.7.1. Asynchronous \texttt{target} with Tasks \hspace{1cm} 135
4.7.2. \texttt{nowait} Clause on \texttt{target} Construct \hspace{1cm} 139
4.7.3. Asynchronous \texttt{target} with \texttt{nowait} and \texttt{depend} Clauses \hspace{1cm} 141

4.8. Array Sections in Device Constructs \hspace{1cm} 144

4.9. Device Routines \hspace{1cm} 148
4.9.1. \texttt{omp_is_initial_device} Routine \hspace{1cm} 148
4.9.2. \texttt{omp_get_num_devices} Routine \hspace{1cm} 150
4.9.3. \texttt{omp_set_default_device} and \texttt{omp_get_default_device} Routines \hspace{1cm} 151
4.9.4. Target Memory and Device Pointers Routines \hspace{1cm} 152

5. SIMD \hspace{1cm} 154
5.1. \texttt{simd} and \texttt{declare simd} Constructs \hspace{1cm} 155
5.2. \texttt{inbranch} and \texttt{notinbranch} Clauses \hspace{1cm} 161
5.3. Loop-Carried Lexical Forward Dependence \hspace{1cm} 165

6. Synchronization \hspace{1cm} 169
6.1. The \texttt{critical} Construct \hspace{1cm} 171
6.2. Worksharing Constructs Inside a \texttt{critical} Construct \hspace{1cm} 174
6.3. Binding of \texttt{barrier} Regions \hspace{1cm} 176
6.4. The \texttt{atomic} Construct \hspace{1cm} 178
6.5. Restrictions on the atomic Construct ....................................... 184
6.6. The flush Construct without a List ........................................... 187
6.7. The ordered Clause and the ordered Construct .............................. 190
6.8. Doacross Loop Nest ............................................................... 194
6.9. Lock Routines ........................................................................ 200
  6.9.1. The omp_init_lock Routine .................................................. 200
  6.9.2. The omp_init_lock_with_hint Routine .................................... 201
  6.9.3. Ownership of Locks ............................................................ 202
  6.9.4. Simple Lock Routines .......................................................... 203
  6.9.5. Nestable Lock Routines ......................................................... 206

7. Data Environment ........................................................................ 209
  7.1. The threadprivate Directive ....................................................... 211
  7.2. The default (none) Clause ......................................................... 217
  7.3. The private Clause ................................................................. 219
  7.4. Fortran Private Loop Iteration Variables ..................................... 223
  7.5. Fortran Restrictions on shared and private Clauses with Common Blocks 225
  7.6. Fortran Restrictions on Storage Association with the private Clause 227
  7.7. C/C++ Arrays in a firstprivate Clause ........................................ 230
  7.8. The lastprivate Clause ............................................................ 232
  7.9. The reduction Clause ............................................................... 233
  7.10. The copyin Clause ................................................................. 240
  7.11. The copyprivate Clause ......................................................... 242
  7.12. C++ Reference in Data-Sharing Clauses .................................... 246
  7.13. Fortran ASSOCIATE Construct ............................................... 247

8. Memory Model ........................................................................... 249
  8.1. The OpenMP Memory Model ..................................................... 250
  8.2. Race Conditions Caused by Implied Copies of Shared Variables in Fortran 256

9. Program Control ......................................................................... 257
  9.1. Conditional Compilation .......................................................... 259
  9.2. Internal Control Variables (ICVs) .............................................. 260
  9.3. Placement of flush, barrier, taskwait and taskyield Directives ....... 263
  9.4. Cancellation Constructs ........................................................... 267
9.5. Nested Loop Constructs ......................................................... 272
9.6. Restrictions on Nesting of Regions ........................................... 275

A. Document Revision History ...................................................... 281
   A.1. Changes from 4.0.2 to 4.5.0 ........................................... 281
   A.2. Changes from 4.0.1 to 4.0.2 ........................................... 282
   A.3. Changes from 4.0 to 4.0.1 ........................................... 282
   A.4. Changes from 3.1 to 4.0 ........................................... 282
Introduction

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

Note – This first release of the OpenMP Examples reflects the OpenMP Version 4.5 specifications. Additional examples are being developed and will be published in future releases of this document.

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

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

The latest source codes for OpenMP Examples can be downloaded from the sources directory at https://github.com/OpenMP/Examples. The codes for this OpenMP 4.5.0 Examples document have the tag v4.5.0.

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

http://www.openmp.org
The following are examples of the OpenMP API directives, constructs, and routines.

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

Each example is labeled as *ename.segno.ext*, where *ename* is the example name, *segno* is the sequence number in a section, and *ext* is the source file extension to indicate the code type and source form. *ext* is one of the following:

- *c* – C code,
- *cpp* – C++ code,
- *f* – Fortran code in fixed form, and
- *f90* – Fortran code in free form.
Parallel Execution

A single thread, the *initial thread*, begins sequential execution of an OpenMP enabled program, as if the whole program is in an implicit parallel region consisting of an implicit task executed by the *initial thread*.

A **parallel** construct encloses code, forming a parallel region. An *initial thread* encountering a **parallel** region forks (creates) a team of threads at the beginning of the **parallel** region, and joins them (removes from execution) at the end of the region. The initial thread becomes the master thread of the team in a **parallel** region with a *thread* number equal to zero, the other threads are numbered from 1 to number of threads minus 1. A team may be comprised of just a single thread.

Each thread of a team is assigned an implicit task consisting of code within the parallel region. The task that creates a parallel region is suspended while the tasks of the team are executed. A thread is tied to its task; that is, only the thread assigned to the task can execute that task. After completion of the **parallel** region, the master thread resumes execution of the generating task.

Any task within a **parallel** region is allowed to encounter another **parallel** region to form a nested **parallel** region. The parallelism of a nested **parallel** region (whether it forks additional threads, or is executed serially by the encountering task) can be controlled by the **OMP_NESTED** environment variable or the **omp_set_nested()** API routine with arguments indicating true or false.

The number of threads of a **parallel** region can be set by the **OMP_NUM_THREADS** environment variable, the **omp_set_num_threads()** routine, or on the **parallel** directive with the **num_threads** clause. The routine overrides the environment variable, and the clause overrides all. Use the **OMP_DYNAMIC** or the **omp_set_dynamic()** function to specify that the OpenMP implementation dynamically adjust the number of threads for **parallel** regions. The default setting for dynamic adjustment is implementation defined. When dynamic adjustment is on and the number of threads is specified, the number of threads becomes an upper limit for the number of threads to be provided by the OpenMP runtime.
WORKSHARING CONSTRUCTS

A worksharing construct distributes the execution of the associated region among the members of the team that encounter it. There is an implied barrier at the end of the worksharing region (there is no barrier at the beginning). The worksharing constructs are:

- loop constructs: *for* and *do*
- *sections*
- *single*
- *workshare*

The *for* and *do* constructs (loop constructs) create a region consisting of a loop. A loop controlled by a loop construct is called an *associated* loop. Nested loops can form a single region when the *collapse* clause (with an integer argument) designates the number of *associated* loops to be executed in parallel, by forming a "single iteration space" for the specified number of nested loops. The *ordered* clause can also control multiple associated loops.

An associated loop must adhere to a "canonical form" (specified in the *Canonical Loop Form* of the OpenMP Specifications document) which allows the iteration count (of all associated loops) to be computed before the (outermost) loop is executed. Most common loops comply with the canonical form, including C++ iterators.

A *single* construct forms a region in which only one thread (any one of the team) executes the region. The other threads wait at the implied barrier at the end, unless the *nowait* clause is specified.

The *sections* construct forms a region that contains one or more structured blocks. Each block of a *sections* directive is constructed with a *section* construct, and executed once by one of the threads (any one) in the team. (If only one block is formed in the region, the *section* construct, which is used to separate blocks, is not required.) The other threads wait at the implied barrier at the end, unless the *nowait* clause is specified.

The *workshare* construct is a Fortran feature that consists of a region with a single structure block (section of code). Statements in the *workshare* region are divided into units of work, and executed (once) by threads of the team.

MASTER CONSTRUCT

The *master* construct is not a worksharing construct. The master region is executed only by the master thread. There is no implicit barrier (and flush) at the end of the *master* region; hence the other threads of the team continue execution beyond code statements beyond the *master* region.
1.1 A Simple Parallel Loop

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

```
C / C++

Example ploop.1.c

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

```
Fortran

Example ploop.1.f

SUBROUTINE SIMPLE(N, A, B)

INTEGER I, N
REAL B(N), A(N)

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

END SUBROUTINE SIMPLE
```

1.2 The parallel Construct

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

```c
#include <omp.h>

void subdomain(float *x, int istart, int ipoints)
{
    int i;
    for (i = 0; i < ipoints; i++)
        x[istart+i] = 123.456;
}

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

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

Example parallel.1.c

OpenMP Examples Version 4.5.0 - November 2016
Example parallel.1.f

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

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

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

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

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

int main (void)
{
    omp_set_nested(1);
    omp_set_dynamic(0);
    #pragma omp parallel
    {
        #pragma omp parallel
        {
            #pragma omp single
            {
                /* If OMP_NUM_THREADS=2,3 was set, the following should print:
                    * Inner: num_thds=3
                    * Inner: num_thds=3
                */
                printf ("Inner: num_thds=%d\n", omp_get_num_threads());
            }
        }
        #pragma omp barrier
        omp_set_nested(0);
        #pragma omp parallel
        {
            #pragma omp single
            {
                /* Even if OMP_NUM_THREADS=2,3 was set, the following should print, because nesting is disabled:
                    * Inner: num_thds=1
                    * Inner: num_thds=1
                */
                printf ("Inner: num_thds=%d\n", omp_get_num_threads());
            }
        }
    }
```

C / C++
C / C++

Fortran

Example nthrs_nesting.1.f

```
program icv
  use omp_lib
  call omp_set_nested(.true.)
  call omp_set_dynamic(.false.)

  !$omp parallel
  !$omp parallel
  !$omp single
  ! If OMP_NUM_THREADS=2,3 was set, the following should print:
  ! Inner: num_thds= 3
  ! Inner: num_thds= 3
  ! If nesting is not supported, the following should print:
  ! Inner: num_thds= 1
  ! Inner: num_thds= 1
  print *, "Inner: num_thds=", omp_get_num_threads()

  !$omp end single
  !$omp end parallel
  !$omp barrier
  call omp_set_nested(.false.)
  !$omp parallel
  !$omp single
  ! Even if OMP_NUM_THREADS=2,3 was set, the following should print,
  ! because nesting is disabled:
  ! Inner: num_thds= 1
  ! Inner: num_thds= 1
  print *, "Inner: num_thds=", omp_get_num_threads()

  !$omp end single
  !$omp end parallel
  !$omp barrier
```
If OMP_NUM_THREADS=2,3 was set, the following should print:

Outer: num_thds= 2

print *, "Outer: num_thds=", omp_get_num_threads()
1.4 Interaction Between the \texttt{num\_threads} Clause and \texttt{omp\_set\_dynamic}

The following example demonstrates the \texttt{num\_threads} clause and the effect of the \texttt{omp\_set\_dynamic} routine on it.

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

---

\textbf{Example nthrs\_dynamic.1.c}

```c
#include <omp.h>

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

---

\textbf{Example nthrs\_dynamic.1.f}

```fortran
PROGRAM EXAMPLE
    INCLUDE "omp\_lib.h"    ! or USE OMP\_LIB
    CALL OMP\_SET\_DYNAMIC(.FALSE.)
    !$OMP PARALLEL NUM\_THREADS(10)
    ! do work here
    !$OMP END PARALLEL
END PROGRAM EXAMPLE
```

The call to the \texttt{omp\_set\_dynamic} routine with a non-zero argument in C/C++, or \texttt{.TRUE.} in Fortran, allows the OpenMP implementation to choose any number of threads between 1 and 10.
Example nthrs_dynamic.2.c

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

Example nthrs_dynamic.2.f

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

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

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

```
Example fort_do.1.f
```

```
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 !$OMP DO
S-10 DO 100 J = 1,10
S-11 CALL WORK(I,J)
S-12 100 CONTINUE !$OMP ENDDO implied here
S-13
S-14 !$OMP DO
S-15 DO 200 J = 1,10
S-16 200 A(I) = I + 1
S-17 !$OMP ENDDO
S-18
S-19 !$OMP DO
S-20 DO 300 I = 1,10
S-21 DO 300 J = 1,10
S-22 CALL WORK(I,J)
S-23 300 CONTINUE
S-24 !$OMP ENDDO
S-25 END SUBROUTINE DO_GOOD
```

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

```
Example fort_do.2.f
```

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

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

```c
#include <math.h>

void nowait_example(int n, int m, float *a, float *b, float *y, float *z)
{
  int i;
  #pragma omp parallel
  {
    #pragma omp for nowait
    for (i=1; i<n; i++)
      b[i] = (a[i] + a[i-1]) / 2.0;
    #pragma omp for nowait
    for (i=0; i<m; i++)
      y[i] = sqrt(z[i]);
  }
}
```

Example nowait.1.f

```fortran
SUBROUTINE NOWAIT_EXAMPLE(N, M, A, B, Y, Z)
INTEGER N, M
REAL A(*), B(*), Y(*), Z(*)
INTEGER I

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

CHAPTER 1. PARALLEL EXECUTION 15```
In the following example, static scheduling distributes the same logical iteration numbers to the threads that execute the three loop regions. This allows the nowait clause to be used, even though there is a data dependence between the loops. The dependence is satisfied as long the same thread executes the same logical iteration numbers in each loop.

Note that the iteration count of the loops must be the same. The example satisfies this requirement, since the iteration space of the first two loops is from 0 to n-1 (from 1 to N in the Fortran version), while the iteration space of the last loop is from 1 to n (2 to N+1 in the Fortran version).
Example nowait.2.f90

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
1.7 The collapse Clause

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

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

---

Example collapse.1.c

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

---

Example collapse.1.f

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

---

OpenMP Examples Version 4.5.0 - November 2016
In the next example, the \( k \) and \( j \) loops are associated with the loop construct. So the iterations of the \( k \) and \( j \) loops are collapsed into one loop with a larger iteration space, and that loop is then divided among the threads in the current team.

The sequential execution of the iterations in the \( k \) and \( j \) loops determines the order of the iterations in the collapsed iteration space. This implies that in the sequentially last iteration of the collapsed iteration space, \( k \) will have the value 2 and \( j \) will have the value 3. Since \texttt{klast} and \texttt{jlast} are \texttt{lastprivate}, their values are assigned by the sequentially last iteration of the collapsed \( k \) and \( j \) loop. This example prints: 2 3.

```c
#include <stdio.h>

void test()
{
    int j, k, jlast, klast;
    #pragma omp parallel
    {
        #pragma omp for collapse(2) lastprivate(jlast, klast)
        for (k=1; k<=2; k++)
            for (j=1; j<=3; j++)
            {
                jlast=j;
                klast=k;
            }
    #pragma omp single
    printf("\%d \%d\n", klast, jlast);
    }
}
```

Example collapse.2.c
Example collapse.2.f

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

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

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

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

The code prints

```
0 1 1
0 1 2
0 2 1
1 2 2
1 3 1
1 3 2
```
Example collapse.3.c

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

void work(int a, int j, int k);

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

Example collapse.3.f

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

CHAPTER 1. PARALLEL EXECUTION
The following example shows the use of the `linear` clause in a loop construct to allow the proper parallelization of a loop that contains an induction variable (`j`). At the end of the execution of the loop construct, the original variable `j` is updated with the value `N/2` from the last iteration of the loop.

```c
#include <stdio.h>

#define N 100

int main(void) {
    float a[N], b[N/2];
    int i, j;
    for (i = 0; i < N; i++)
        a[i] = i + 1;
    j = 0;
    #pragma omp parallel
    #pragma omp for linear(j:1)
    for (i = 0; i < N; i += 2) {
        b[j] = a[i] * 2.0f;
        j++;
    }
    printf("%d %f %f\n", j, b[0], b[j-1]);
    /* print out: 50 2.0 198.0 */
    return 0;
}
```

Example `linear_in_loop.1.c`
Fortran Example linear_in_loop.1.f90

program linear_loop
implicit none
integer, parameter :: N = 100
real :: a(N), b(N/2)
integer :: i, j

do i = 1, N
  a(i) = i
end do
j = 0
!$omp parallel
!$omp do linear(j:1)
doi = 1, N, 2
  j = j + 1
  b(j) = a(i) * 2.0
end do
!$omp end parallel
print *, j, b(1), b(j)
! print out: 50 2.0 198.0
end program
1.9 The parallel sections Construct

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

```
Example psections.1.c
S-1 void XAXIS();
S-2 void YAXIS();
S-3 void ZAXIS();
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 }
```

```
Example psections.1.f
S-1 SUBROUTINE SECT_EXAMPLE()
S-2 !$OMP PARALLEL SECTIONS
S-3 !$OMP SECTION
S-4 CALL XAXIS()
S-5 !$OMP SECTION
S-6 CALL YAXIS()
S-7
S-8 !$OMP SECTION
S-9 CALL ZAXIS()
S-10
S-11 !$OMP END PARALLEL SECTIONS
S-12 END SUBROUTINE SECT_EXAMPLE
```
1.10 The firstprivate Clause and the sections Construct

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

```
#include <omp.h>
#include <stdio.h>
#define NT 4
int main( ) {
    int section_count = 0;
    omp_set_dynamic(0);
    omp_set_num_threads(NT);
    #pragma omp parallel
    #pragma omp sections firstprivate( section_count )
    {
        #pragma omp section
        {
            section_count++;
            /* may print the number one or two */
            printf( "section_count %d\n", section_count );
        }
        #pragma omp section
        {
            section_count++;
            /* may print the number one or two */
            printf( "section_count %d\n", section_count );
        }
    }
    return 0;
}
```
Example fpriv_sections.1.f90

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
1.11 The single Construct

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

Example single.1.c

```c
#include <stdio.h>

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

void single_example()
{
    #pragma omp parallel
    {
        #pragma omp single
        printf("Beginning work1.\n");
        work1();
        #pragma omp single
        printf("Finishing work1.\n");
        work1();
        #pragma omp single nowait
        printf("Finished work1 and beginning work2.\n");
        work2();
    }
}
```

C / C++
Example single_1.f

SUBROUTINE WORK1()
END SUBROUTINE WORK1

SUBROUTINE WORK2()
END SUBROUTINE WORK2

PROGRAM SINGLE_EXAMPLE
!$OMP PARALLEL

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

CALL WORK1()

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

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

CALL WORK2()

!$OMP END PARALLEL

END PROGRAM SINGLE_EXAMPLE

END SUBROUTINE WORK2
1.12 The *workshare* Construct

The following are examples of the *workshare* construct.

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

*Example workshare.1.f*

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

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

*Example workshare.2.f*

```fortran
SUBROUTINE WSHARE2(AA, BB, CC, DD, EE, FF, N)
  INTEGER N
  REAL AA(N, N), BB(N, N), CC(N, N)
  REAL DD(N, N), EE(N, N), FF(N, N)
  !$OMP PARALLEL
  !$OMP WORKSHARE
  AA = BB
  CC = DD
  !$OMP END WORKSHARE NOWAIT
  !$OMP WORKSHARE
  EE = FF
  !$OMP END WORKSHARE
  !$OMP END PARALLEL
END SUBROUTINE WSHARE2
```
The following example shows the use of an atomic directive inside a workshare construct. The computation of \texttt{SUM(AA)} is workshared, but the update to \texttt{R} is atomic.

\textit{Example workshare.3.f}

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

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

Each task gets worked on in order by the threads:

```
AA = BB then
CC = DD then
EE .ne. 0 then
FF = 1 / EE then
GG = HH
```

\textit{Example workshare.4.f}

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

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

*Example workshare.5.f*

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

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

*Example workshare.6.f*

```fortran
SUBROUTINE WSHARE6_WRONG(AA, BB, CC, DD, N)
    INTEGER N
    REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
    INTEGER PRI
    !$OMP PARALLEL PRIVATE(PRI)
    !$OMP WORKSHARE
    AA = BB
    PRI = 1
    CC = DD * PRI
    !$OMP END WORKSHARE
    !$OMP END PARALLEL
END SUBROUTINE WSHARE6_WRONG
```
Fortran execution rules must be enforced inside a workshare construct. In the following example, the same result is produced in the following program fragment regardless of whether the code is executed sequentially or inside an OpenMP program with multiple threads:

Example workshare.7.f

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

$OMP END PARALLEL
END SUBROUTINE WSHARE7
```

Fortran
1.13 **The master Construct**

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

---

_C / C++_

---

### Example master.1.c

```c
#include <stdio.h>

extern float average(float,float,float);

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

---

_C / C++_

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

Example pra_iterator.1.cpp
```
#include <vector>
void iterator_example()
{
    std::vector<int> vec(23);
    std::vector<int>::iterator it;
    #pragma omp parallel for default(none) shared(vec)
    for (it = vec.begin(); it < vec.end(); it++)
    {
        // do work with *it //
    }
}
```
Some programs rely on a fixed, prespecified number of threads to execute correctly. Because the default setting for the dynamic adjustment of the number of threads is implementation defined, such programs can choose to turn off the dynamic threads capability and set the number of threads explicitly to ensure portability. The following example shows how to do this using `omp_set_dynamic`, and `omp_set_num_threads`.

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

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

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

void dynthreads(float *x, int npoints)
{
    int iam, ipoints;
    omp_set_dynamic(0);
    omp_set_num_threads(16);
    #pragma omp parallel shared(x, npoints) private(iam, ipoints)
    {
        if (omp_get_num_threads() != 16)
            abort();
        iam = omp_get_thread_num();
        ipoints = npoints/16;
        do_by_16(x, iam, ipoints);
    }
}
```
Example set_dynamic_nthrs.1.f

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

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

```c
#include <omp.h>

void work(int i);

void incorrect()
{
  int np, i;
  np = omp_get_num_threads(); /* misplaced */

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

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

SUBROUTINE INCORRECT()
  INCLUDE "omp_lib.h" ! or USE OMP_LIB
  INTEGER I, NP
  NP = OMP_GET_NUM_THREADS() ! misplaced: will return 1
  !$OMP PARALLEL DO SCHEDULE(STATIC)
  DO I = 0, NP-1
    CALL WORK(I)
  ENDDO
  !$OMP END PARALLEL DO
END SUBROUTINE INCORRECT
```
The following example shows how to rewrite this program without including a query for the number of threads:

```
Example get_nthrs.2.c

#include <omp.h>

void work(int i);

void correct()
{
    int i;

    #pragma omp parallel private(i)
    {
        i = omp_get_thread_num();
        work(i);
    }
}
```

```
Example get_nthrs.2.f

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

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

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

END SUBROUTINE CORRECT
```
OpenMP Affinity consists of a proc_bind policy (thread affinity policy) and a specification of places ("location units" or processors that may be cores, hardware threads, sockets, etc.). OpenMP Affinity enables users to bind computations on specific places. The placement will hold for the duration of the parallel region. However, the runtime is free to migrate the OpenMP threads to different cores (hardware threads, sockets, etc.) prescribed within a given place, if two or more cores (hardware threads, sockets, etc.) have been assigned to a given place.

Often the binding can be managed without resorting to explicitly setting places. Without the specification of places in the OMP_PLACES variable, the OpenMP runtime will distribute and bind threads using the entire range of processors for the OpenMP program, according to the OMP_PROC_BIND environment variable or the proc_bind clause. When places are specified, the OMP runtime binds threads to the places according to a default distribution policy, or those specified in the OMP_PROC_BIND environment variable or the proc_bind clause.

In the OpenMP Specifications document a processor refers to an execution unit that is enabled for an OpenMP thread to use. A processor is a core when there is no SMT (Simultaneous Multi-Threaded) support or SMT is disabled. When SMT is enabled, a processor is a hardware thread (HW-thread). (This is the usual case; but actually, the execution unit is implementation defined.) Processor numbers are numbered sequentially from 0 to the number of cores less one (without SMT), or 0 to the number HW-threads less one (with SMT). OpenMP places use the processor number to designate binding locations (unless an "abstract name" is used.)

The processors available to a process may be a subset of the system’s processors. This restriction may be the result of a wrapper process controlling the execution (such as numactl on Linux systems), compiler options, library-specific environment variables, or default kernel settings. For instance, the execution of multiple MPI processes, launched on a single compute node, will each have a subset of processors as determined by the MPI launcher or set by MPI affinity environment variables for the MPI library.

Threads of a team are positioned onto places in a compact manner, a scattered distribution, or onto the master’s place, by setting the OMP_PROC_BIND environment variable or the proc_bind
clause to close, spread, or master, respectively. When `OMP_PROC_BIND` is set to FALSE no
binding is enforced; and when the value is TRUE, the binding is implementation defined to a set of
places in the `OMP_PLACES` variable or to places defined by the implementation if the
`OMP_PLACES` variable is not set.

The `OMP_PLACES` variable can also be set to an abstract name (threads, cores, sockets) to specify
that a place is either a single hardware thread, a core, or a socket, respectively. This description of
the `OMP_PLACES` is most useful when the number of threads is equal to the number of hardware
thread, cores or sockets. It can also be used with a close or spread distribution policy when the
equality doesn’t hold.
### 2.1 The proc_bind Clause

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

![Machine Architecture Diagram]

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

- The example places are written in OpenMP's `OMP_PLACES` format:
  ```
  OMP_PLACES="{0,1},{2,3},{4,5},{6,7},{8,9},{10,11},{12,13},{14,15}" 
  ```

- Or, equivalently with a more compact place list:
  ```
  OMP_PLACES="{0:2}:8:2" 
  ```

### 2.1.1 Spread Affinity Policy

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

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

---

OpenMP Examples Version 4.5.0 - November 2016
It is unspecified on which place the master thread is initially started. If the master thread is initially started on p0, the following placement of threads will be applied in the parallel region:

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

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

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

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

Let \( T \) be the number of threads in the team, and \( P \) be the number of places in the parent’s place partition. The first \( T/P \) threads of the team (including the master thread) execute on the parent’s place. The next \( T/P \) threads execute on the next place in the place partition, and so on, with wrap around.
It is unspecified on which place the master thread is initially started. If the master thread is initially started on p0, the following placement of threads will be applied in the parallel region:

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

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

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

### 2.1.2 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.
It is unspecified on which place the master thread is initially started. If the master thread is initially started on p0, the following placement of threads will be applied in the `parallel` region:

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

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

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

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

Let $T$ be the number of threads in the team, and $P$ be the number of places in the parent’s place partition. The first $T/P$ threads of the team (including the master thread) execute on the parent’s place. The next $T/P$ threads execute on the next place in the place partition, and so on, with wrap around. The place partition is not changed by the `close` policy.
void work();
void foo()
{
#pragma omp parallel num_threads(16) proc_bind(close)
{
    work();
}
}

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

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

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

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

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

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

```c
void work();

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

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

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

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

- threads 0-3 execute on p2 with the place partition p0-p7
In the example below a team of threads is generated on each socket of the system, using nested parallelism. Several query functions are used to gather information to support the creation of the teams and to obtain socket and thread numbers.

For proper execution of the code, the user must create a place partition, such that each place is a listing of the core numbers for a socket. For example, in a 2 socket system with 8 cores in each socket, and sequential numbering in the socket for the core numbers, the OMP_PLACES variable would be set to "{0:8},{8:8}", using the place syntax {lower_bound:length:stride}, and the default stride of 1.

The code determines the number of sockets (n_sockets) using the omp_get_num_places() query function. In this example each place is constructed with a list of each socket’s core numbers, hence the number of places is equal to the number of sockets.

The outer parallel region forms a team of threads, and each thread executes on a socket (place) because the proc_bind clause uses spread in the outer parallel construct. Next, in the socket_init function, an inner parallel region creates a team of threads equal to the number of elements (core numbers) from the place of the parent thread. Because the outer parallel construct uses a spread affinity policy, each of its threads inherits a subpartition of the original partition. Hence, the omp_get_place_num_procs query function returns the number of elements (here procs = cores) in the subpartition of the thread. After each parent thread creates its nested parallel region on the section, the socket number and thread number are reported.

Note: Portable tools like hwloc (Portable HardWare LOCality package), which support many common operating systems, can be used to determine the configuration of a system. On some systems there are utilities, files or user guides that provide configuration information. For instance, the socket number and proc_id’s for a socket can be found in the /proc/cpuinfo text file on Linux systems.

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

  void socket_init(int socket_num)
  {
    int n_procs;
    n_procs = omp_get_place_num_procs(socket_num);
    #pragma omp parallel num_threads(n_procs) proc_bind(close)
    {
      printf("Reporting in from socket num, thread num: %d %d\n", socket_num, n_procs);
    }
  
```
int main()
{
    int n_sockets, socket_num;
    omp_set_nested(1); // or export OMP_NESTED=true
    omp_set_max_active_levels(2); // or export OMP_MAX_ACTIVE_LEVELS=2
    n_sockets = omp_get_num_places();
    #pragma omp parallel num_threads(n_sockets) private(socket_num) \
    proc_bind(spread)
    {
        socket_num = omp_get_place_num();
        socket_init(socket_num);
    }
}

Example affinity.6.f90

subroutine socket_init(socket_num)
    use omp_lib
    integer :: socket_num, n_procs
    n_procs = omp_get_place_num_procs(socket_num)
    !$omp parallel num_threads(n_procs) proc_bind(close)
    print*,"Reporting in from socket num, thread num: ", &
    socket_num,omp_get_thread_num()
end subroutine

program numa_teams
    use omp_lib
    integer :: n_sockets, socket_num
    call omp_set_nested(.true.) ! or export OMP_NESTED=true
    call omp_set_max_active_levels(2) ! or export OMP_MAX_ACTIVE_LEVELS=2
    n_sockets = omp_get_num_places()
    !$omp parallel num_threads(n_sockets) private(socket_num) & 
    !$omp& proc_bind(spread)
  
  

CHAPTER 2. OPENMP AFFINITY
S-24
S-25  socket_num = omp_get_place_num()
S-26  call socket_init(socket_num)
S-27
S-28  !$omp end parallel
S-29  end program

----------------------------------  Fortran  ----------------------------------
Tasking

Tasking constructs provide units of work to a thread for execution. Worksharing constructs do this, too (e.g. for, do, sections, and singles constructs); but the work units are tightly controlled by an iteration limit and limited scheduling, or a limited number of sections or single regions. Worksharing was designed with "data parallel" computing in mind. Tasking was designed for "task parallel" computing and often involves non-locality or irregularity in memory access.

The task construct can be used to execute work chunks: in a while loop; while traversing nodes in a list; at nodes in a tree graph; or in a normal loop (with a taskloop construct). Unlike the statically scheduled loop iterations of worksharing, a task is often enqueued, and then dequeued for execution by any of the threads of the team within a parallel region. The generation of tasks can be from a single generating thread (creating sibling tasks), or from multiple generators in a recursive graph tree traversals. A taskloop construct bundles iterations of an associated loop into tasks, and provides similar controls found in the task construct.

Sibling tasks are synchronized by the taskwait construct, and tasks and their descendent tasks can be synchronized by containing them in a taskgroup region. Ordered execution is accomplished by specifying dependences with a depend clause. Also, priorities can be specified as hints to the scheduler through a priority clause.

Various clauses can be used to manage and optimize task generation, as well as reduce the overhead of execution and to relinquish control of threads for work balance and forward progress.

Once a thread starts executing a task, it is the designated thread for executing the task to completion, even though it may leave the execution at a scheduling point and return later. The thread is tied to the task. Scheduling points can be introduced with the taskyield construct. With an untied clause any other thread is allowed to continue the task. An if clause with a true expression allows the generating thread to immediately execute the task as an undeferred task. By including the data environment of the generating task into the generated task with the mergeable and final clauses, task generation overhead can be reduced.

A complete list of the tasking constructs and details of their clauses can be found in the Tasking Constructs chapter of the OpenMP Specifications, in the OpenMP Application Programming Interface section.
3.1 The task and taskwait Constructs

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

```c
struct node {
    struct node *left;
    struct node *right;
};

extern void process(struct node *);

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

```fortran
RECURSIVE SUBROUTINE traverse ( P )
    TYPE Node
        TYPE(Node), POINTER :: left, right
    END TYPE Node
    TYPE(Node) :: P
    IF (associated(P%left)) THEN
        !$OMP TASK ! P is firstprivate by default
        CALL traverse(P%left)
        !$OMP END TASK
    ENDIF
    IF (associated(P%right)) THEN
        !$OMP TASK ! P is firstprivate by default
        CALL traverse(P%right)
        !$OMP END TASK
    ENDIF
    CALL process ( P )
END SUBROUTINE
```
In the next example, we force a postorder traversal of the tree by adding a `taskwait` directive.

Now, we can safely assume that the left and right sons have been executed before we process the current node.

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

```fortran
RECURSIVE SUBROUTINE traverse ( P )
  TYPE Node
  TYPE(Node), POINTER :: left, right
END TYPE Node
TYPE(Node) :: P
IF (associated(P%left)) THEN
  !$OMP TASK ! P is firstprivate by default
  CALL traverse(P%left)
  !$OMP END TASK
ENDIF
IF (associated(P%right)) THEN
  !$OMP TASK ! P is firstprivate by default
  CALL traverse(P%right)
  !$OMP END TASK
ENDIF
END SUBROUTINE
```
The following example demonstrates how to use the task construct to process elements of a linked list in parallel. The thread executing the single region generates all of the explicit tasks, which are then executed by the threads in the current team. The pointer p is firstprivate by default on the task construct so it is not necessary to specify it in a firstprivate clause.

Example tasking.3.c

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

void process(node * p)
{
    /* do work here */
}

void increment_list_items(node * head)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            node * p = head;
            while (p) {
                #pragma omp task
                // p is firstprivate by default
                process(p);
                p = p->next;
            }
        }
    }
}
The *fib()* function should be called from within a `parallel` region for the different specified tasks to be executed in parallel. Also, only one thread of the `parallel` region should call `fib()` unless multiple concurrent Fibonacci computations are desired.
Example tasking.4.c

```c
int fib(int n) {
    int i, j;
    if (n<2)
        return n;
    else {
        #pragma omp task shared(i)
        i=fib(n-1);
        #pragma omp task shared(j)
        j=fib(n-2);
        #pragma omp taskwait
        return i+j;
    }
}
```

Example tasking.4.f

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

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

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

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

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

Example tasking.5.f

```fortran
real*8 item(10000000)
integer i

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

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

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

Example tasking.6.c

```c
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];
extern void process(double);
int main() {
#pragma omp parallel
{
#pragma omp single
{
  int i;
#pragma omp task untied
  // i is firstprivate, item is shared
  {
  for (i=0; i<LARGE_NUMBER; i++)
  #pragma omp task
```
Example tasking.6.f

1

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

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

A task switch may occur at a task scheduling point. A single thread may execute both of the task regions that modify tp. The parts of these task regions in which tp is modified may be executed in any order so the resulting value of var can be either 1 or 2.
Example tasking.7.c

```c
#include <stdio.h>

int main() {
    int tp;
    #pragma omp threadprivate(tp)
    int var;
    void work() {
        #pragma omp task
        {
            /* do work here */
            #pragma omp task
            {
                tp = 1;
                /* do work here */
                #pragma omp task
                {
                    /* no modification of tp */
                }
                var = tp;  // value of tp can be 1 or 2
            }
            tp = 2;
        }
    }
    return 0;
}
```

Example tasking.7.f

```fortran
module example
    integer tp
    !$omp threadprivate(tp)
    integer var
    contains
    subroutine work
        !$omp task
        ! do work here
        !$omp task
        tp = 1
        ! do work here
        !$omp task
        ! no modification of tp
        !$omp end task
        var = tp  ! value of var can be 1 or 2
        !$omp end task
        tp = 2
end subroutine work
end module example
```
In this example, scheduling constraints prohibit a thread in the team from executing a new task that modifies `tp` while another such task region tied to the same thread is suspended. Therefore, the value written will persist across the task scheduling point.

```c/c++
int tp;
#pragma omp threadprivate(tp)
int var;
void work()
{
    #pragma omp parallel
    {
        /* do work here */
        #pragma omp task
        {
            tp++;
            /* do work here */
            #pragma omp task
            {
                /* do work here but don’t modify tp */
            }
        }
    }
    var = tp; //Value does not change after write above
}
```
The following two examples demonstrate how the scheduling rules illustrated in Section 2.11.3 of the OpenMP 4.0 specification affect the usage of locks and critical sections in tasks. If a lock is held across a task scheduling point, no attempt should be made to acquire the same lock in any code that may be interleaved. Otherwise, a deadlock is possible.

In the example below, suppose the thread executing task 1 defers task 2. When it encounters the task scheduling point at task 3, it could suspend task 1 and begin task 2 which will result in a deadlock when it tries to enter critical region 1.
//Capture data for the following task
#pragma omp task
{ /* do work here */ } //Task 3
}
}
}

Example tasking.9.f

module example
contains
subroutine work

 !$omp task
! Task 1

 !$omp task
! Task 2

 !$omp critical
! Critical region 1

 !$omp end critical
! do work here

 !$omp end task

 !$omp critical
! Critical region 2

 !$omp end critical
! Capture data for the following task

 !$omp task
!Task 3

 !$omp end task

 !$omp end critical

 !$omp end task

 !$omp end task

 end subroutine

 end module

In the following example, **lock** is held across a task scheduling point. However, according to the
scheduling restrictions, the executing thread can’t begin executing one of the non-descendant tasks
that also acquires **lock** before the task region is complete. Therefore, no deadlock is possible.
Example tasking.10.c

```c
#include <omp.h>

void work() {

  omp_lock_t lock;
  omp_init_lock(&lock);

  #pragma omp parallel
  {
    int i;
    #pragma omp for
    for (i = 0; i < 100; i++) {
      #pragma omp task
      {
        // lock is shared by default in the task
        omp_set_lock(&lock);
        // Capture data for the following task
        #pragma omp task
        // Task Scheduling Point 1
        { /* do work here */ }
        omp_unset_lock(&lock);
      }
    }
  }

  omp_destroy_lock(&lock);

}
```

Example tasking.10.f90

```fortran
module example
  include 'omp_lib.h'
  integer (kind=omp_lock_kind) lock
  integer i
contains
  subroutine work
    call omp_init_lock(lock)
    !$omp parallel
    !$omp do
    do i=1,100
    !$omp task
    ! Outer task
      call omp_set_lock(lock) ! lock is shared by
    ! default in the task
```
Fortran

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

C / C++

Example tasking.11.c

```
#include <stdio.h>
void foo ()
{
    int x = 2;
    #pragma omp task shared(x) mergeable
    {
        x++;
    }
    #pragma omp taskwait
    printf("%d\n", x);  // prints 3
}
```
Example tasking.11.f90

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

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

Example tasking.12.c

```c
#include <stdio.h>
void foo ( )
{
    int x = 2;
    #pragma omp task mergeable
    {
        x++;
    }
    #pragma omp taskwait
    printf("%d\n",x); // prints 2 or 3
}
```
The following example shows the use of the `final` clause and the `omp_in_final` API call in a recursive binary search program. To reduce overhead, once a certain depth of recursion is reached the program uses the `final` clause to create only included tasks, which allow additional optimizations. The use of the `omp_in_final` API call allows programmers to optimize their code by specifying which parts of the program are not necessary when a task can create only included tasks (that is, the code is inside a `final` task). In this example, the use of a different state variable is not necessary so once the program reaches the part of the computation that is finalized and copying from the parent state to the new state is eliminated. The allocation of `new_state` in the stack could also be avoided but it would make this example less clear. The `final` clause is most effective when used in conjunction with the `mergeable` clause since all tasks created in a `final` task region are included tasks that can be merged if the `mergeable` clause is present.
recursive subroutine bin_search(pos, n, state)
   use omp_lib
   integer :: pos, n
   character, pointer :: state(:)
   character, target, dimension(n) :: new_state1, new_state2
   integer, parameter :: LIMIT = 3
   if (pos .eq. n) then
      call check_solution(state)
      return
   endif
   !$omp task final(pos > LIMIT) mergeable
   if (.not. omp_in_final()) then
      new_state1(1:pos) = state(1:pos)
      state => new_state1
   endif
   state(pos+1) = 'z'
   call bin_search(pos+1, n, state)
   !$omp end task
   !$omp task final(pos > LIMIT) mergeable
   if (.not. omp_in_final()) then
      new_state2(1:pos) = state(1:pos)
      state => new_state2
   endif
   state(pos+1) = 'y'
   call bin_search(pos+1, n, state)
The following example illustrates the difference between the if and the final clauses. The if clause has a local effect. In the first nest of tasks, the one that has the if clause will be undeferred but the task nested inside that task will not be affected by the if clause and will be created as usual. Alternatively, the final clause affects all task constructs in the final task region but not the final task itself. In the second nest of tasks, the nested tasks will be created as included tasks. Note also that the conditions for the if and final clauses are usually the opposite.

```
Example tasking.14.c

void bar(void);

void foo ( )
{
    int i;
    #pragma omp task if(0) // This task is undeferred
    {
        #pragma omp task // This task is a regular task
        for (i = 0; i < 3; i++) {
            #pragma omp task // This task is a regular task
            bar();
        }
    }
    #pragma omp task final(1) // This task is a regular task
    {
        #pragma omp task // This task is included
        for (i = 0; i < 3; i++) {
            #pragma omp task // This task is also included
            bar();
        }
    }
}
```
subroutine foo()
  integer i
  !$omp task if(.FALSE.) ! This task is undeferred
  !$omp task ! This task is a regular task
  do i = 1, 3
    !$omp task ! This task is a regular task
    call bar()
    !$omp end task
  enddo
  !$omp end task
  !$omp end task
  !$omp task final(.TRUE.) ! This task is a regular task
  !$omp task ! This task is included
  do i = 1, 3
    !$omp task ! This task is also included
    call bar()
    !$omp end task
  enddo
  !$omp end task
  !$omp end task
end subroutine
3.2 Task Priority

In this example we compute arrays in a matrix through a compute_array routine. Each task has a priority value equal to the value of the loop variable \( i \) at the moment of its creation. A higher priority on a task means that a task is a candidate to run sooner.

The creation of tasks occurs in ascending order (according to the iteration space of the loop) but a hint, by means of the `priority` clause, is provided to reverse the execution order.

```
C / C++
void compute_array (float *node, int M);

void compute_matrix (float *array, int N, int M)
{
    int i;
    #pragma omp parallel private(i)
    #pragma omp single
    {
        for (i=0; i<N; i++) {
            #pragma omp task priority(i)
            compute_array(&array[i*M], M);
        }
    }
}
```

```
Fortran
subroutine compute_matrix(matrix, M, N)
    implicit none
    integer :: M, N
    real :: matrix(M, N)
    integer :: i
    interface
    subroutine compute_array(node, M)
       implicit none
       integer :: M
       real :: node(M)
    end subroutine
    end interface
    !$omp parallel private(i)
    !$omp single
    do i=1,N
        !$omp task priority(i)
```
call compute_array(matrix(:, i), M)
enddo
$omp end single
$omp end parallel
subroutine compute_matrix
3.3 Task Dependences

3.3.1 Flow Dependence

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

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

```fortran
program example
    integer :: x
    x = 1
    !$omp parallel
    !$omp single
    !$omp task shared(x) depend(out: x)
    x = 2
    !$omp end task
    !$omp task shared(x) depend(in: x)
    print*, "x = ", x
    !$omp end task
    !$omp end single
    !$omp end parallel
end program
```
The program will always print "x = 2", because the depend clauses enforce the ordering of the tasks. If the depend clauses had been omitted, then the tasks could execute in any order and the program and the program would have a race condition.

### 3.3.2 Anti-dependence

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

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

C / C++

OpenMP Examples Version 4.5.0 - November 2016
The program will always print "x = 1", because the `depend` clauses enforce the ordering of the tasks. If the `depend` clauses had been omitted, then the tasks could execute in any order and the program would have a race condition.

### 3.3.3 Output Dependence

In this example we show an output dependence expressed using the `depend` clause on the `task` construct.
Example task_dep.3.f90

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

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

### 3.3.4 Concurrent Execution with Dependences

In this example we show potentially concurrent execution of tasks using multiple flow dependences expressed using the `depend` clause on the `task` construct.
Example task_dep.4.c

```c
#include <stdio.h>

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

Example task_dep.4.f90

```fortran
program example
    integer :: x
    x = 1
    !$omp parallel
    !$omp single
    !$omp task shared(x) depend(out: x)
    x = 2
    !$omp end task
    !$omp task shared(x) depend(in: x)
    print*, "x + 1 = ", x+1, "."
    !$omp end task
    !$omp task shared(x) depend(in: x)
    print*, "x + 2 = ", x+2, "."
    !$omp end task
    !$omp end single
    !$omp end parallel
end program
```
The last two tasks are dependent on the first task. However there is no dependence between the last two tasks, which may execute in any order (or concurrently if more than one thread is available).

Thus, the possible outputs are "x + 1 = 3. x + 2 = 4." and "x + 2 = 4. x + 1 = 3." If the depend clauses had been omitted, then all of the tasks could execute in any order and the program would have a race condition.

### 3.3.5 Matrix multiplication

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

```c
// Assume BS divides N perfectly
void matmul_depend(int N, int BS, float A[N][N], float B[N][N], float C[N][N])
{
    int i, j, k, ii, jj, kk;
    for (i = 0; i < N; i+=BS) {
        for (j = 0; j < N; j+=BS) {
            for (k = 0; k < N; k+=BS) {
                // Note 1: i, j, k, A, B, C are firstprivate by default
                // Note 2: A, B and C are just pointers
                #pragma omp task private(ii, jj, kk) \
                    depend ( in: A[i:BS][k:BS], B[k:BS][j:BS] ) \ 
                    depend ( inout: C[i:BS][j:BS] )
                for (ii = i; ii < i+BS; ii++)
                    for (jj = j; jj < j+BS; jj++)
                        for (kk = k; kk < k+BS; kk++)
            }
        }
    }
}
```
Example task_dep.5.f90

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
### 3.4 The taskgroup Construct

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

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

---

**Example taskgroup.1.c**

```c
extern void start_background_work(void);
extern void check_step(void);
extern void print_results(void);
struct tree_node
{
    struct tree_node *left;
    struct tree_node *right;
};
typedef struct tree_node* tree_type;
extern void init_tree(tree_type);
define max_steps 100
void compute_something(tree_type tree)
{
    // some computation
}
void compute_tree(tree_type tree)
{
    if (tree->left)
    {
        #pragma omp task
        compute_tree(tree->left);
    }
    if (tree->right)
    {
        #pragma omp task
        compute_tree(tree->right);
    }
    #pragma omp task
    compute_something(tree);
}
int main()
{

```
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  }

Example taskgroup.1.f90

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
endif
!
omp task
    call compute_something(tree)
!
omp end task
end subroutine
end module
program main
    use tree_type_mod
    type(tree_type), pointer :: tree
    call init_tree(tree);
!
omp parallel
!
omp single
!
omp task
    call start_background_work()
!
omp end task
!
omp taskgroup
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
omp end taskgroup ! wait on tree traversal in this step
!
omp task
    call compute_tree(tree)
!
omp end task
!
3.5 The `taskyield` Construct

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

```c
#include <omp.h>

void something_useful ( void );
void something_critical ( void );

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

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

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

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

---

CHAPTER 3. TASKING  83
call something_critical()
call omp_unset_lock(lock)
end do
end subroutine
3.6 The taskloop Construct

The following example illustrates how to execute a long running task concurrently with tasks created with a `taskloop` directive for a loop having unbalanced amounts of work for its iterations.

The `grainsize` clause specifies that each task is to execute at least 500 iterations of the loop.

The `nogroup` clause removes the implicit taskgroup of the `taskloop` construct; the explicit `taskgroup` construct in the example ensures that the function is not exited before the long-running task and the loops have finished execution.

```c
S-1 void long_running_task(void);
S-2 void loop_body(int i, int j);
S-3
S-4 void parallel_work(void) {
S-5     int i, j;
S-6     #pragma omp taskgroup
S-7     {
S-8         #pragma omp task
S-9             long_running_task(); // can execute concurrently
S-10        #pragma omp taskloop private(j) grainsize(500) nogroup
S-11             for (i = 0; i < 10000; i++) { // can execute concurrently
S-12                 for (j = 0; j < i; j++) {
S-13                     loop_body(i, j);
S-14                 }
S-15             }
S-16         }
S-17     }
S-18 }
```

Example taskloop.1.c
Example taskloop.1.f90

subroutine parallel_work
    integer i
    integer j
    !$omp taskgroup
    !$omp task
call long_running_task()
    !$omp end task
    !$omp taskloop private(j) grainsize(500) nogroup
do i=1,10000
do j=1,i
call loop_body(i, j)
end do
end do
    !$omp end taskloop
    !$omp end taskgroup
end subroutine
CHAPTER 4

Devices

The target construct consists of a target directive and an execution region. The target region is executed on the default device or the device specified in the device clause.

In OpenMP version 4.0, by default, all variables within the lexical scope of the construct are copied to and from the device, unless the device is the host, or the data exists on the device from a previously executed data-type construct that has created space on the device and possibly copied host data to the device storage.

The constructs that explicitly create storage, transfer data, and free storage on the device are categorized as structured and unstructured. The target data construct is structured. It creates a data region around target constructs, and is convenient for providing persistent data throughout multiple target regions. The target enter data and target exit data constructs are unstructured, because they can occur anywhere and do not support a "structure" (a region) for enclosing target constructs, as does the target data construct.

The map clause is used on target constructs and the data-type constructs to map host data. It specifies the device storage and data movement to and from the device, and controls on the storage duration.

There is an important change in the OpenMP 4.5 specification that alters the data model for scalar variables and C/C++ pointer variables. The default behavior for scalar variables and C/C++ pointer variables in an 4.5 compliant code is firstprivate. Example codes that have been updated to reflect this new behavior are annotated with a description that describes changes required for correct execution. Often it is a simple matter of mapping the variable as tofrom to obtain the intended 4.0 behavior.

In OpenMP version 4.5 the mechanism for target execution is specified as occurring through a target task. When the target construct is encountered a new target task is generated. The target task completes after the target region has executed and all data transfers have finished.

This new specification does not affect the execution of pre-4.5 code; it is a necessary element for asynchronous execution of the target region when using the new nowait clause introduced in OpenMP 4.5.
4.1 target Construct

4.1.1 target Construct on parallel Construct

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

```
# Example target.1.c
extern void init(float*, float*, int);
extern void output(float*, int);
void vec_mult(int N)
{
    int i;
    float p[N], v1[N], v2[N];
    init(v1, v2, N);
    #pragma omp target
    #pragma omp parallel for private(i)
    for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];
    output(p, N);
}
```

```
# Example target.1.f90
subroutine vec_mult(N)
integer :: i,N
real :: p(N), v1(N), v2(N)
call init(v1, v2, N)
!$omp target
!$omp parallel do
do i=1,N
    p(i) = v1(i) * v2(i)
end do
!$omp end target
call output(p, N)
end subroutine
```
4.1.2 target Construct with map Clause

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

C / C++

Example target.2.c

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

C / C++

Fortran

Example target.2.f90

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

Fortran

CHAPTER 4. DEVICES 89
4.1.3 map Clause with to/from map-types

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

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

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

--- C / C++ ---

Example target.3.c

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++ ---

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.
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`.

```
extern void init(float*, float*, int);
extern void output(float*, int);
void vec_mult(float *p, float *v1, float *v2, int N) {
  int i;
  init(v1, v2, N);
  #pragma omp target map(to: v1[0:N], v2[:N]) map(from: p[0:N])
  #pragma omp parallel for
  for (i=0; i<N; i++)
    p[i] = v1[i] * v2[i];
  output(p, N);
}
```
In C, the length of the pointed-to array must be specified. In Fortran the extent of the array is known and the length need not be specified. A section of the array can be specified with the usual Fortran syntax, as shown in the following example. The value 1 is assumed for the lower bound for array section \( v2(1:N) \).

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

A more realistic situation in which an assumed-size array is passed to `vec_mult` requires that the length of the arrays be specified, because the compiler does not know the size of the storage. A section of the array must be specified with the usual Fortran syntax, as shown in the following example. The value 1 is assumed for the lower bound for array section \( v2(:N) \).
Example target.4b.f90

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

4.1.5 target Construct with if Clause

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

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

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

Example target.5.c

```c
#define THRESHOLD1 1000000
#define THRESHOLD2 1000
extern void init(float*, float*, int);
extern void output(float*, int);
void vec_mult(float *p, float *v1, float *v2, int N)
{
  int i;
  init(v1, v2, N);
  #pragma omp target if(N>THRESHOLD1) map(to: v1[0:N], v2[:N])
  #pragma omp parallel for if(N>THRESHOLD2)
  #pragma omp target map(to: v1(1:N), v2(:N)) map(from: p(1:N))
```
The following example is a modification of the above target.5 code to show the combined `target` and parallel loop directives. It uses the `directive-name` modifier in multiple `if` clauses to specify the component directive to which it applies.

The `if` clause with the `target` modifier applies to the `target` component of the combined directive, and the `if` clause with the `parallel` modifier applies to the `parallel` component of the combined directive.
Example target.6.c

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

Example target.6.f90

```fortran
module params
integer,parameter :: THRESHOLD1=1000000, THRESHOLD2=1000
end module

subroutine vec_mult(p, v1, v2, N)
    use params
    real :: p(N), v1(N), v2(N)
    integer :: i
    call init(v1, v2, N)
    !$omp target parallel do &
    !$omp& if(target: N>THRESHOLD1) if(parallel: N>THRESHOLD2) &
    !$omp& map(to: v1, v2 ) map(from: p)
    do i=1,N
        p(i) = v1(i) * v2(i)
    end do
    !$omp end target parallel do
    call output(p, N)
end subroutine
```
4.2 target data Construct

4.2.1 Simple target data Construct

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

```
Example target_data.1.c
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 }
```

The Fortran code passes a reference and specifies the extent of the arrays in the declaration. No length information is necessary in the map clause, as is required with C/C++ pointers.
Example target_data.1.f90

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

4.2.2 target data Region Enclosing Multiple target Regions

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

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

---

**Example target_data.3.c**

```c
#include <math.h>
#define COLS 100
void gramSchmidt(float Q[][COLS], const int rows)
{
    int cols = COLS;
    #pragma omp target data map(Q[0:rows][0:cols])
    for(int k=0; k < cols; k++)
    {
        double tmp = 0.0;
        #pragma omp target map(tofrom: tmp)
        #pragma omp parallel for reduction(+:tmp)
        for(int i=0; i < rows; i++)
        {
            tmp += (Q[i][k] * Q[i][k]);
        }
        tmp = 1/sqrt(tmp);
    }
    #pragma omp target
    #pragma omp parallel for
    for(int i=0; i < rows; i++)
    {
        Q[i][k] *= tmp;
    }
    /* Note: The variable tmp is now mapped with tofrom, for correct execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro. */
}
```
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.

Example `target_data.3.f90`

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

! Note: The variable \( tmp \) is now mapped with `tofrom`, for correct ! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
4.2.3 target data Construct with Orphaned Call

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

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

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

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

---

C / C++

Example target_data.4.c

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

---

CHAPTER 4. DEVICES 101
The Fortran code maps the pointers and storage in an identical manner (same extent, but uses indices from 1 to N).

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

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

### Fortran

**Example target_data.4.f90**

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

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.
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.
subroutine vec_mult(p1,v3,v4,N)
real,dimension(:) :: p1, v3, v4
integer :: N,i
!$omp target map(to: v3, v4) map(from: p1)
!$omp parallel do
    do i=1,N
        p1(i) = v3(i) * v4(i)
    end do
!$omp end target
end subroutine

end module

program main
use my_mult
integer, parameter :: N=1024
real,allocatable, dimension(:) :: p, v1, v2
allocate( p(N), v1(N), v2(N) )
call foo(p,v1,v2,N)
end program

4.2.4 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.
The \texttt{if} clauses work the same way for the following Fortran code. The \texttt{target} constructs enclosed in the \texttt{target data} region should also use an \texttt{if} clause with the same condition, so that the \texttt{target data} region and the \texttt{target} region are either both created for the device, or are both ignored.
In the following example, when the `if` clause conditional expression on the `target` construct evaluates to `false`, the target region will execute on the host device. However, the `target data` construct created an enclosing device data environment that mapped `p[0:N]` to a device data environment on the default device. At the end of the `target data` region the array section `p[0:N]` will be assigned from the device data environment to the corresponding variable in the data environment of the task that encountered the `target data` construct, resulting in undefined values in `p[0:N].`
The if clauses work the same way for the following Fortran code. When the if clause conditional expression on the target construct evaluates to false, the target region will execute on the host device. However, the target data construct created an enclosing device data environment that mapped the p array (and v1 and v2) to a device data environment on the default target device. At the end of the target data region the p array will be assigned from the device data environment to the corresponding variable in the data environment of the task that encountered the target data construct, resulting in undefined values in p.

---

Fortran

Example target_data.7.f90

```fortran
module params
integer, parameter :: THRESHOLD=1000000
end module

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

The structured data construct (target data) provides persistent data on a device for subsequent target constructs as shown in the target data examples above. This is accomplished by creating a single target data region containing target constructs.

The unstructured data constructs allow the creation and deletion of data on the device at any appropriate point within the host code, as shown below with the target enter data and target exit data constructs.

The following C++ code creates/deletes a vector in a constructor/destructor of a class. The constructor creates a vector with target enter data and uses an alloc modifier in the map clause to avoid copying values to the device. The destructor deletes the data (target exit data) and uses the delete modifier in the map clause to avoid copying data back to the host. Note, the stand-alone target enter data occurs after the host vector is created, and the target exit data construct occurs before the host data is deleted.

```cpp
class Matrix {
  Matrix(int n) {
    len = n;
    v = new double[len];
    #pragma omp target enter data map(alloc:v[0:len])
  }
  ~Matrix() {
    // NOTE: delete map type should be used, since the corresponding host data will cease to exist after the deconstructor is called.
    #pragma omp target exit data map(delete:v[0:len])
    delete[] v;
  }
  private:
  double* v;
  int len;
};
```

The following C code allocates and frees the data member of a Matrix structure. The
The \texttt{init_matrix} function allocates the memory used in the structure and uses the \texttt{target enter data} directive to map it to the target device. The \texttt{free_matrix} function removes the mapped array from the target device and then frees the memory on the host. Note, the stand-alone \texttt{target enter data} occurs after the host memory is allocated, and the \texttt{target exit data} construct occurs before the host data is freed.

```
#include <stdlib.h>

typedef struct {
  double *A;
  int N;
} Matrix;

void init_matrix(Matrix *mat, int n) {
  mat->A = (double *)malloc(n*sizeof(double));
  mat->N = n;
  #pragma omp target enter data map(alloc:mat->A[:n])
}

void free_matrix(Matrix *mat) {
  #pragma omp target exit data map(delete:mat->A[:mat->N])
  mat->N = 0;
  free(mat->A);
  mat->A = NULL;
}
```

The following Fortran code allocates and deallocates a module array. The \texttt{initialize} subroutine allocates the module array and uses the \texttt{target enter data} directive to map it to the target device. The \texttt{finalize} subroutine removes the mapped array from the target device and then deallocates the array on the host. Note, the stand-alone \texttt{target enter data} occurs after the host memory is allocated, and the \texttt{target exit data} construct occurs before the host data is deallocated.
module example
  real(8), allocatable :: A(:)
contains
  subroutine initialize(N)
    integer :: N
    allocate(A(N))
    !$omp target enter data map(alloc:A)
  end subroutine initialize
  subroutine finalize()
    !$omp target exit data map(delete:A)
    deallocate(A)
  end subroutine finalize
end module example
4.4 target update Construct

4.4.1 Simple target data and target update Constructs

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

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

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

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

The target update construct assigns the new values of \( v1 \) and \( v2 \) from the task’s data environment to the corresponding mapped array sections in the device data environment of the target data construct.

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

The second target region uses the updated values of \( v1[:N] \) and \( v2[:N] \).

---

\textbf{C / C++}

\textit{Example target\_update.1.c}

```c
extern void init(float *, float *, int);
extern void init\_again(float *, float *, int);
extern void output(float *, int);
void vec\_mult(float *p, float *v1, float *v2, int N)
{
    int i;
    init(v1, v2, N);
    #pragma omp target data map(to: v1[:N], v2[:N]) map(from: p[0:N])
    {
        #pragma omp target
        #pragma omp parallel for
        for (i=0; i<N; i++)
            p[i] = v1[i] * v2[i];
        init\_again(v1, v2, N);
        #pragma omp target update to(v1[:N], v2[:N])
        #pragma omp target
        #pragma omp parallel for
        for (i=0; i<N; i++)
```
Example target_update.1.f90

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

OpenMP Examples Version 4.5.0 - November 2016
4.4.2 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.2.c

```c
extern void init(float *, float *, int);
extern int maybe_init_again(float *, int);
extern void output(float *, int);
void vec_mult(float *p, float *v1, float *v2, int N)
{
    int i;
    init(v1, v2, N);
    #pragma omp target data map(to: v1[::N], v2[::N]) map(from: p[0:N])
    {
        int changed;
        #pragma omp target
        #pragma omp parallel for
        for (i=0; i<N; i++)
            p[i] = v1[i] * v2[i];
        changed = maybe_init_again(v1, N);
        #pragma omp target update if (changed) to(v1[::N])
        changed = maybe_init_again(v2, N);
        #pragma omp target update if (changed) to(v2[::N])
        #pragma omp target
        #pragma omp parallel for
        for (i=0; i<N; i++)
            p[i] = p[i] + (v1[i] * v2[i]);
    }
    output(p, N);
}
```
Example target_update.2.f90

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

4.5 declare target Construct

4.5.1 declare target and end declare target for a Function

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

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

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

```c
#include <omp.h>

#define THRESHOLD 1000000

void fib_wrapper(int n)
{
    #pragma omp target if(n >= THRESHOLD)
    {
        fib(n);
    }
}
```

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

The program uses the `module_fib` module, which presents an explicit interface to the compiler with the `declare target` declarations for processing the `fib` call.
Example declare_target.1.f90

```fortran
module module_fib
  contains
    subroutine fib(N)
      integer :: N
      !$omp declare target
      !...
    end subroutine
  end module

module params
  integer :: THRESHOLD=1000000
end module

program my_fib
  use params
  use module_fib
  !$omp target if( N > THRESHOLD )
  call fib(N)
  !$omp end target
end program
```

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

Example declare_target.2.f90

```fortran
program my_fib
  integer :: N = 8
  !$omp declare target(fib)
  !$omp target
  call fib(N)
  !$omp end target
end program

subroutine fib(N)
  integer :: N
  !$omp declare target
  print*, "hello from fib"
  !...
end subroutine
```
4.5.2 declare target Construct for Class Type

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

Example declare_target.2.cpp

```cpp
struct typeX
{
    int a;
};

class typeY
{
    int a;
    public:
    int foo() { return a^0x01;}
};

#pragma omp declare target
struct typeX varX; // ok
class typeY varY; // ok if varY.foo() not called on target device
#pragma omp end declare target

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

4.5.3 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.

---

**Example declare_target.3.c**

```
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  }
```

---

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

---

**Example declare_target.3.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)
```

---

OpenMP Examples Version 4.5.0 - November 2016
The following example also indicates that the function `Pfun()` is available on the target device, as well as the variable `Q`, which is mapped to the implicit device data environment of each target device. The `target update` directive is then used to manage the consistency of the variable `Q` between the data environment of the encountering host device task and the implicit device data environment of the default target device.

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

C / C++
Example declare_target.4.c

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

/* Note: The variable tmp is now mapped with tofrom, for correct execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro. */
```

The Fortran version of the above C code uses a different syntax. In Fortran modules a list syntax on the `declare target` directive is used to declare mapped variables and procedures. The `N` and `Q` variables are declared as a comma separated list. When the `declare target` directive is used to declare just the procedure, the procedure name need not be listed – it is implicitly assumed, as illustrated in the `Pfun()` function.
Example declare_target.f90

```fortran
module my_global_array

!$omp declare target (N,Q)
integer, parameter :: N=10
real :: Q(N,N)
contains
function Pfun(i,k)
  !$omp declare target
  real :: Pfun
  integer,intent(in) :: i,k
  Pfun=(Q(i,k) * Q(k,i))
end function
end module

function accum(k) result(tmp)
  use my_global_array
  real :: tmp
  integer :: i, k
  tmp = 0.0e0
  !$omp target map(tofrom: tmp)
  !$omp parallel do reduction(+:tmp)
  do i=1,N
    tmp = tmp + Pfun(k,i)
  end do
  !$omp end target
end function

! Note: The variable tmp is now mapped with tofrom, for correct
! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
```

4.5.4 declare target and end declare target

with declare simd

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

OpenMP Examples Version 4.5.0 - November 2016
Example declare_target.5.c

```c
#define N 10000
#define M 1024
#pragma omp declare target
float Q[N][N];
#pragma omp declare simd uniform(i) linear(k) notinbranch
float P(const int i, const int k)
{
    return Q[i][k] * Q[k][i];
}
#pragma omp end declare target

float accum(void)
{
    float tmp = 0.0;
    int i, k;
    #pragma omp target map(tofrom: tmp)
    #pragma omp parallel for reduction(+:tmp)
    for (i=0; i < N; i++) {
        float tmp1 = 0.0;
        #pragma omp simd reduction(+:tmp1)
        for (k=0; k < M; k++) {
            tmp1 += P(i,k);
        }
        tmp += tmp1;
    }
    return tmp;
}

/* Note: The variable tmp is now mapped with tofrom, for correct
   execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro. */
```

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.
Example declare_target.5.f90

module my_global_array

!$omp declare target (N,Q)
integer, parameter :: N=10000, M=1024
real :: Q(N,N)

contains

function P(k,i)
!$omp declare simd uniform(i) linear(k) notinbranch
!$omp declare target
real :: P
integer,intent(in) :: k,i
P=(Q(k,i) * Q(i,k))
end function

end module

function accum() result(tmp)
use my_global_array
real :: tmp, tmp1
integer :: i

tmp = 0.0e0
!$omp target map(tofrom: tmp)
!$omp parallel do private(tmp1) reduction(+:tmp)
do i=1,N
  tmp1 = 0.0e0
!$omp simd reduction(+:tmp1)
do k = 1,M
  tmp1 = tmp1 + P(k,i)
  end do
  tmp = tmp + tmp1
end do
!$omp end target
end function

! Note: The variable tmp is now mapped with tofrom, for correct
! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
4.5.5 declare target Directive with link Clause

In the OpenMP 4.5 standard the declare target directive was extended to allow static data to be mapped, when needed, through a link clause.

Data storage for items listed in the link clause becomes available on the device when it is mapped implicitly or explicitly in a map clause, and it persists for the scope of the mapping (as specified by a target construct, a target data construct, or target enter/exit data constructs).

Tip: When all the global data items will not fit on a device and are not needed simultaneously, use the link clause and map the data only when it is needed.

The following C and Fortran examples show two sets of data (single precision and double precision) that are global on the host for the entire execution on the host; but are only used globally on the device for part of the program execution. The single precision data are allocated and persist only for the first target region. Similarly, the double precision data are in scope on the device only for the second target region.

--- C / C++ ---

Example declare_target.6.c

```c
#define N 100000000

#pragma omp declare target link(sp,sv1,sv2) 
    link(dp,dv1,dv2)

float sp[N], sv1[N], sv2[N];
double dp[N], dv1[N], dv2[N];

void s_init(float *, float *, int);
void d_init(double *, double *, int);
void s_output(float *, int);
void d_output(double *, int);

#pragma omp declare target
void s_vec_mult_accum()
{
    int i;

    #pragma omp parallel for
    for (i=0; i<N; i++)
        sp[i] = sv1[i] * sv2[i];
}

void d_vec_mult_accum()
{
    int i;
```
```c
#pragma omp parallel for
for (i=0; i<N; i++)
    dp[i] = dv1[i] * dv2[i];
}
#pragma omp end declare target

int main()
{
    s_init(sv1, sv2, N);
    #pragma omp target map(to:sv1,sv2) map(from:sp)
    s_vec_mult_accum();
    s_output(sp, N);
```

```c
    d_init(dv1, dv2, N);
    #pragma omp target map(to:dv1,dv2) map(from:dp)
    d_vec_mult_accum();
    d_output(dp, N);
```

    return 0;
}
```

---

**Example declare_target.6.f90**

```fortran
module m_dat
    integer, parameter :: N=100000000
    !$omp declare target link(sp,sv1,sv2)
    real :: sp(N), sv1(N), sv2(N)
```

```fortran
    !$omp declare target link(dp,dv1,dv2)
    double precision :: dp(N), dv1(N), dv2(N)
```

contains
```fortran
    subroutine s_vec_mult_accum()
    !$omp declare target
    integer :: i
```

```fortran
    !$omp parallel do
    do i = 1,N
        sp(i) = sv1(i) * sv2(i)
    end do
```

```fortran
    end subroutine s_vec_mult_accum
```

```fortran
    subroutine d_vec_mult_accum()
    !$omp declare target
```

```fortran
    end subroutine d_vec_mult_accum
```

```fortran
end module m_dat
```
integer :: i

!$omp parallel do
do i = 1,N
  dp(i) = dv1(i) * dv2(i)
end do

end subroutine

end module m_dat

program prec_vec_mult
  use m_dat
  call s_init(sv1, sv2, N)
  !$omp target map(to:sv1,sv2) map(from:sp)
  call s_vec_mult_accum()
  !$omp end target
  call s_output(sp, N)

  call d_init(dv1, dv2, N)
  !$omp target map(to:dv1,dv2) map(from:dp)
  call d_vec_mult_accum()
  !$omp end target
  call d_output(dp, N)

end program
4.6 teams Constructs

4.6.1 target and teams Constructs with omp_get_num_teams and omp_get_team_num Routines

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

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

---

**Example teams.1.c**

```c
#include <stdlib.h>
#include <omp.h>
float dotprod(float B[], float C[], int N)
{
    float sum0 = 0.0;
    float sum1 = 0.0;
    #pragma omp target map(to: B[:N], C[:N]) map(tofrom: sum0, sum1)
    #pragma omp teams num_teams(2)
    {
        int i;
        if (omp_get_num_teams() != 2)
            abort();
        if (omp_get_team_num() == 0)
            {
                #pragma omp parallel for reduction(+:sum0)
                for (i=0; i<N/2; i++)
                    sum0 += B[i] * C[i];
            }
        else if (omp_get_team_num() == 1)
            {
                #pragma omp parallel for reduction(+:sum1)
                for (i=N/2; i<N; i++)
                    sum1 += B[i] * C[i];
            }
    }
    return sum0 + sum1;
}

/* Note: The variables sum0,sum1 are now mapped with tofrom, for correct
```
Example teams.1.f90

```fortran
function dotprod(B,C,N) result(sum)

use omp_lib, ONLY : omp_get_num_teams, omp_get_team_num

real :: B(N), C(N), sum,sum0, sum1
integer :: N, i

sum0 = 0.0e0
sum1 = 0.0e0

 !$omp target map(to: B, C) map(tofrom: sum0, sum1)
 !$omp teams num_teams(2)

 if (omp_get_num_teams() /= 2) stop "2 teams required"
 if (omp_get_team_num() == 0) then
   !$omp parallel do reduction(+:sum0)
   do i=1,N/2
   sum0 = sum0 + B(i) * C(i)
   end do
 else if (omp_get_team_num() == 1) then
   !$omp parallel do reduction(+:sum1)
   do i=N/2+1,N
   sum1 = sum1 + B(i) * C(i)
   end do
 end if
 !$omp end teams
 !$omp end target
 sum = sum0 + sum1
end function

! Note: The variables sum0,sum1 are now mapped with tofrom, for correct
! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
```

CHAPTER 4. DEVICES  127
4.6.2 target, teams, and distribute Constructs

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

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

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

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

```c
#define min(x, y) (((x) < (y)) ? (x) : (y))

float dotprod(float B[], float C[], int N, int block_size,
              int num_teams, int block_threads)
{
    float sum = 0.0;
    int i, i0;
    #pragma omp target map(to: B[0:N], C[0:N]) map(tofrom: sum)
    #pragma omp teams num_teams(num_teams) thread_limit(block_threads) \
         reduction(+:sum)
    #pragma omp distribute
    for (i0=0; i0<N; i0 += block_size)
        #pragma omp parallel for reduction(+:sum)
        for (i=i0; i< min(i0+block_size,N); i++)
            sum += B[i] * C[i];
    return sum;
}
/* Note: The variable sum is now mapped with tofrom, for correct execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro. */
```

---
Example teams.2.f90

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

! Note: The variable sum is now mapped with tofrom, for correct
! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
```

4.6.3 target teams, and Distribute Parallel Loop Constructs

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

The distribute parallel loop construct schedules the loop iterations across the master threads of each team and then across the threads of each team.
Example teams.3.c

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

/* Note: The variable sum is now mapped with tofrom from the defaultmap
 clause on the combined target teams construct, for correct
 execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro. */

Example teams.3.f90

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

! Note: The variable sum is now mapped with tofrom from the defaultmap
! clause on the combined target teams construct, for correct
! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
4.6.4 target teams and Distribute Parallel Loop Constructs with Scheduling Clauses

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

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

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

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

---

Example teams.4.c

```c
#define N 1024*1024
float dotprod(float B[], float C[])
{
    float sum = 0.0;
    int i;
    #pragma omp target map(to: B[0:N], C[0:N]) map(tofrom: sum)
    #pragma omp teams num_teams(8) thread_limit(16) reduction(+:sum)
    #pragma omp distribute parallel for reduction(+:sum) \
        dist_schedule(static, 1024) schedule(static, 64)
    for (i=0; i<N; i++)
        sum += B[i] * C[i];
    return sum;
}
/* Note: The variable sum is now mapped with tofrom, for correct
   execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro. */
```
Example teams.4.f90

```fortran
module arrays
integer,parameter :: N=1024*1024
real :: B(N), C(N)
end module

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

! Note: The variable sum is now mapped with tofrom, for correct
! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
```

4.6.5 target teams and distribute simd Constructs

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

The `distribute simd` construct schedules the loop iterations across the master thread of each team and then uses SIMD parallelism to execute the iterations.
Example teams.5.c

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

Example teams.5.f90

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

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

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

--- C / C++ ---

target teams and Distribute Parallel Loop SIMD Constructs

--- Fortran ---

Example teams.6.c

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

Example teams.6.f90

```fortran
subroutine vec_mult(p, v1, v2, N)
    real :: p(N), v1(N), v2(N)
    integer :: i
    call init(v1, v2, N)
    !$omp target teams map(to: v1, v2) map(from: p)
    !$omp distribute parallel do simd
    do i=1,N
        p(i) = v1(i) * v2(i)
    end do
    !$omp end target teams
    call output(p, N)
end subroutine
```
Asynchronous execution of a `target` region can be accomplished by creating an explicit task around the `target` region. Examples with explicit tasks are shown at the beginning of this section.

As of OpenMP 4.5 and beyond, the `nowait` clause can be used on the `target` directive for asynchronous execution. Examples with `nowait` clauses follow the explicit `task` examples.

This section also shows the use of `depend` clauses to order executions through dependences.

### 4.7.1 Asynchronous `target` with Tasks

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

```c
#include <omp.h>

int N = 1000000000;
#define CHUNKSZ 1000000

void init(float *, int);

int main() {
    float Z[N];
    void pipedF() {
        int C, i;
        init(Z, N);
        for (C=0; C<N; C+=CHUNKSZ) {
            #pragma omp task shared(Z)
            #pragma omp target map(Z[C:CHUNKSZ])
            #pragma omp parallel for
            for (i=0; i<CHUNKSZ; i++)
                Z[i] = F(Z[i]);
        }
        #pragma omp taskwait
    }
    return 0;
}
```

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

```fortran
module parameters
  integer, parameter :: N=1000000000, CHUNKSZ=1000000
end module
subroutine pipedF()
  use parameters, ONLY: N, CHUNKSZ
  integer :: C, i
  real :: z(N)
  interface
    function F(z)
      !$omp declare target
      real, intent(IN) :: z
      real :: F
      end function F
  end interface
  call init(z,N)
  do C=1,N,CHUNKSZ
    !$omp task shared(z)
    !$omp target map(z(C:C+CHUNKSZ-1))
    !$omp parallel do
      do i=C,C+CHUNKSZ-1
        z(i) = F(z(i))
      end do
    !$omp end target
    !$omp end task
  end do
  !$omp taskwait
  print*, z
end subroutine pipedF
```

The following example shows how the `task` and `target` constructs are used to execute multiple `target` regions asynchronously. The task dependence ensures that the storage is allocated and initialized on the device before it is accessed.
The Fortran example below is similar to the C version above. Instead of pointers, though, it uses the convenience of Fortran allocatable arrays on the device. In order to preserve the arrays allocated on the device across multiple target regions, a target data region is used in this case.
If there is no shape specified for an allocatable array in a `map` clause, only the array descriptor (also called a dope vector) is mapped. That is, device space is created for the descriptor, and it is initially populated with host values. In this case, the `v1` and `v2` arrays will be in a non-associated state on the device. When space for `v1` and `v2` is allocated on the device in the first `target` region the addresses to the space will be included in their descriptors.

At the end of the first `target` region, the arrays `v1` and `v2` are preserved on the device for access in the second `target` region. At the end of the second `target` region, the data in array `p` is copied back, the arrays `v1` and `v2` are not.

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

**Note** – This example is not strictly compliant with the OpenMP 4.5 specification since the allocation status of allocatable arrays `v1` and `v2` is changed inside the `target` region, which is not allowed. (See the restrictions for the `map` clause in the *Data-mapping Attribute Rules and Clauses* section of the specification.) However, the intention is to relax the restrictions on mapping of allocatable variables in the next release of the specification so that the example will be compliant.

---

**Fortran**

```
subroutine mult(p, N, idev)
  use omp_lib, ONLY: omp_is_initial_device
  real :: p(N)
  real, allocatable :: v1(:), v2(:)
  integer :: i, idev
  !$omp declare target (init)
  !$omp target data map(v1,v2)
  !$omp task shared(v1,v2) depend(out: N)
  !$omp target device(idev)
  if( omp_is_initial_device() ) &
     stop "not executing on target device"
  allocate(v1(N), v2(N))
  call init(v1,v2,N)
  !$omp end target
  !$omp end task
  call foo() ! execute other work asychronously
  !$omp task shared(v1,v2,p) depend(in: N)
  !$omp target device(idev) map(from: p)
```
```
The following example shows how to execute code asynchronously on a device without an explicit task. The `nowait` clause on a `target` construct allows the thread of the `target task` to perform other work while waiting for the `target` region execution to complete. Hence, the `target` region can execute asynchronously on the device (without requiring a host thread to idle while waiting for the `target task` execution to complete).

In this example the product of two vectors (arrays), \( v1 \) and \( v2 \), is formed. One half of the operations is performed on the device, and the last half on the host, concurrently.

After a team of threads is formed the master thread generates the `target task` while the other threads can continue on, without a barrier, to the execution of the host portion of the vector product. The completion of the `target task` (asynchronous target execution) is guaranteed by the synchronization in the implicit barrier at the end of the host vector-product worksharing loop region. See the `barrier` glossary entry in the OpenMP specification for details.

The host loop scheduling is `dynamic`, to balance the host thread executions, since one thread is being used for offload generation. In the situation where little time is spent by the `target task` in setting up and tearing down the the target execution, `static` scheduling may be desired.
Example async_target.3.c

```c
#include <stdio.h>
#define N 1000000  // N must be even
void init(int n, float *v1, float *v2);

int main(){
    int i, n=N;
    int chunk=1000;
    float v1[N],v2[N],vxv[N];
    init(n, v1,v2);

    #pragma omp for schedule(dynamic,chunk)
    for(i=n/2; i<n; i++){ vxv[i] = v1[i]*v2[i]; }    
    printf(" vxv[0] vxv[n-1] %f %f\n", vxv[0], vxv[n-1]);
    return 0;
}
```

Example async_target.3.f90

```fortran
program concurrent_async
use omp_lib
integer,parameter :: n=1000000  ! n must be even
integer :: i, chunk=1000
real :: v1(n),v2(n),vxv(n)

call init(n, v1,v2)
```
4.7.3 Asynchronous target with nowait and depend Clauses

More details on dependences can be found in Section 3.3 on page 73, Task Dependences. In this example, there are three flow dependences. In the first two dependences the target task does not execute until the preceding explicit tasks have finished. These dependences are produced by arrays v1 and v2 with the out dependence type in the first two tasks, and the in dependence type in the target task.

The last dependence is produced by array p with the out dependence type in the target task, and the in dependence type in the last task. The last task does not execute until the target task finishes.

The nowait clause on the target construct creates a deferrable target task, allowing the encountering task to continue execution without waiting for the completion of the target task.
Example async_target.4.c

```c
void vec_mult(int N)
{
    int i;
    float p[N], v1[N], v2[N];
    #pragma omp parallel num_threads(2)
    {
        #pragma omp single
        {
            #pragma omp task depend(out:v1)
            init(v1, N);
            #pragma omp task depend(out:v2)
            init(v2, N);
            #pragma omp target nowait depend(in:v1,v2) depend(out:p) \ 
                map(to:v1,v2) map( from: p)
            for (i=0; i<N; i++)
                p[i] = v1[i] * v2[i];
            #pragma omp task depend(in:p)
            output(p, N);
        }
    }
}
```

Example async_target.4.f90

```fortran
subroutine vec_mult(N)
    implicit none
    integer :: i, N
    real, allocatable :: p(:), v1(:), v2(:)
    allocate( p(N), v1(N), v2(N) )
    !$omp parallel num_threads(2)
```
!$omp single
  !$omp task depend(out:v1)
call init(v1, N)
  !$omp end task
S-16
  !$omp task depend(out:v2)
call init(v2, N)
  !$omp end task
S-20
  !$omp target nowait depend(in:v1,v2) depend(out:p) &
  !$omp&
    map(to:v1,v2) map(from: p)
  !$omp parallel do
  do i=1,N
    p(i) = v1(i) * v2(i)
  end do
  !$omp end target
S-29
  !$omp task depend(in:p)
call output(p, N)
  !$omp end task
S-33
  !$omp end single
S-34
  !$omp end parallel
S-36
dallocate( p, v1, v2 )
S-38
end subroutine
4.8 Array Sections in Device Constructs

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

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

```
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-10    }
S-11  }
S-12 }
```

```
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   !$omp end target
S-9   !$omp end target data
S-10 end subroutine
```

This example shows the invalid usage of two separate sections of the same array inside of a `target` construct.
Example array_sections.2.c

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

Example array_sections.2.f90

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

This example shows the valid usage of two separate sections of the same array inside of a `target` construct.
This example shows the valid usage of a wholly contained array section of an already mapped array section inside of a `target` construct.
Example array_sections.4.c

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

Example array_sections.4.f90

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

4.9.1 `omp_is_initial_device` Routine

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

---

C / C++

---

Example device.1.c

```c
#include <stdio.h>
#include <omp.h>
#pragma omp declare target
void vec_mult(float *p, float *v1, float *v2, int N);
extern float *p, *v1, *v2;
extern int N;
#pragma omp end declare target
extern void init_vars(float *, float *, int);
extern void output(float *, int);
void foo()
{
    init_vars(v1, v2, N);
    #pragma omp target device(42) map(p[:N], v1[:N], v2[:N])
    {
        vec_mult(p, v1, v2, N);
    }
    output(p, N);
}
void vec_mult(float *p, float *v1, float *v2, int N)
{
    int i;
    int nthreads;
    if (!omp_is_initial_device())
    {
        printf("1024 threads on target device\n");
        nthreads = 1024;
    }
    else
    {
        printf("8 threads on initial device\n");
        nthreads = 8;
    }
    #pragma omp parallel for private(i) num_threads(nthreads)
    for (i=0; i<N; i++)
```
Example device.1.f90

module params
    integer, parameter :: N=1024
end module params

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

program prog_vec_mult
    use params
    use vmult
    real :: p(N), v1(N), v2(N)
    call init(v1,v2,N)
    !$omp target device(42) map(p, v1, v2)
    call vec_mult(p, v1, v2, N)
    !$omp end target
    call output(p, N)
end program
4.9.2 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.

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

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

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

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

void foo(void)
{
    int default_device = omp_get_default_device();
    printf("Default device = %d\n", default_device);
    omp_set_default_device(default_device+1);
    if (omp_get_default_device() != default_device+1)
        printf("Default device is still = %d\n", default_device);
}
```

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

The following example shows how to create space on a device, transfer data to and from that space, and free the space, using API calls. The API calls directly execute allocation, copy and free operations on the device, without invoking any mapping through a target directive. The omp_target_alloc routine allocates space and returns a device pointer for referencing the space in the omp_target_memcpy API routine on the host. The omp_target_free routine frees the space on the device.

The example also illustrates how to access that space in a target region by exposing the device pointer in an is_device_ptr clause.

The example creates an array of cosine values on the default device, to be used on the host device. The function fails if a default device is not available.

```
void get_dev_cos(double *mem, size_t s)
{
    int h, t, i;
    double * mem_dev_cpy;
    h = omp_get_initial_device();
    t = omp_get_default_device();

    if (omp_get_num_devices() < 1 || t < 0){
        printf(" ERROR: No device found.\n");
        exit(1);
    }

    mem_dev_cpy = omp_target_alloc( sizeof(double) * s, t);
    if(mem_dev_cpy == NULL){
        printf(" ERROR: No space left on device.\n");
        exit(1);
    }

    /* dst src */
    omp_target_memcpy(mem_dev_cpy, mem, sizeof(double)*s, 0, 0, t, h);

    #pragma omp target is_device_ptr(mem_dev_cpy) device(t)
}
```
#pragma omp teams distribute parallel for
for(i=0;i<s;i++) { mem_dev_cpy[i] = cos((double)i); } /* init data */

/* dst src */
omp_target_memcpy(mem, mem_dev_cpy, sizeof(double)*s, 0, 0, h, t);
omp_target_free(mem_dev_cpy, t);
SIMD

Single instruction, multiple data (SIMD) is a form of parallel execution in which the same operation is performed on multiple data elements independently in hardware vector processing units (VPU), also called SIMD units. The addition of two vectors to form a third vector is a SIMD operation. Many processors have SIMD (vector) units that can perform simultaneously 2, 4, 8 or more executions of the same operation (by a single SIMD unit).

Loops without loop-carried backward dependency (or with dependency preserved using ordered simd) are candidates for vectorization by the compiler for execution with SIMD units. In addition, with state-of-the-art vectorization technology and declare simd construct extensions for function vectorization in the OpenMP 4.5 specification, loops with function calls can be vectorized as well. The basic idea is that a scalar function call in a loop can be replaced by a vector version of the function, and the loop can be vectorized simultaneously by combining a loop vectorization (simd directive on the loop) and a function vectorization (declare simd directive on the function).

A simd construct states that SIMD operations be performed on the data within the loop. A number of clauses are available to provide data-sharing attributes (private, linear, reduction and lastprivate). Other clauses provide vector length preference/restrictions (simdlen / safelen), loop fusion (collapse), and data alignment (aligned).

The declare simd directive designates that a vector version of the function should also be constructed for execution within loops that contain the function and have a simd directive. Clauses provide argument specifications (linear, uniform, and aligned), a requested vector length (simdlen), and designate whether the function is always/never called conditionally in a loop (branch/inbranch). The latter is for optimizing performance.

Also, the simd construct has been combined with the worksharing loop constructs (for simd and do simd) to enable simultaneous thread execution in different SIMD units.
5.1 `simd` and declare `simd` Constructs

The following example illustrates the basic use of the `simd` construct to assure the compiler that the loop can be vectorized.

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

```fortran
subroutine star(a,b,c,n,ioff_ptr)
  implicit none
  double precision :: a(*),b(*),c(*)
  integer :: n, i
  integer, pointer :: ioff_ptr
  !$omp simd
  do i = 1,n
    a(i) = a(i) * b(i) * c(i+ioff_ptr)
  end do
end subroutine
```
When a function can be inlined within a loop the compiler has an opportunity to vectorize the loop. By guaranteeing SIMD behavior of a function’s operations, characterizing the arguments of the function and privatizing temporary variables of the loop, the compiler can often create faster, vector code for the loop. In the examples below the declare simd construct is used on the add1 and add2 functions to enable creation of their corresponding SIMD function versions for execution within the associated SIMD loop. The functions characterize two different approaches of accessing data within the function: by a single variable and as an element in a data array, respectively. The add3 C function uses dereferencing.

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

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

---

C / C++

Example SIMD.2.c

```c
#include <stdio.h>

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

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

#pragma omp declare simd uniform(fact) linear(a,b:1)
double add3(double *a, double *b, double fact)
{
```

---

156 OpenMP Examples Version 4.5.0 - November 2016
double c;
c = *a + *b + fact;
return c;
}

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

int main(){
    int i;
    const int N=32;
    double a[N], b[N];
    
    for ( i=0; i<N; i++ ) {
        a[i] = i; b[i] = N-i;
    }
    
    work(a, b, N );
    
    for ( i=0; i<N; i++ ) {
        printf("%d %f\n", i, a[i]);
    }
    
    return 0;
}

---

Example SIMD.2.f90

program main
    implicit none
    integer, parameter :: N=32
    integer :: i
    double precision :: a(N), b(N)
    do i = 1,N
        a(i) = i-1
        b(i) = N-(i-1)
    end do
A thread that encounters a SIMD construct executes a vectorized code of the iterations. Similar to the concerns of a worksharing loop a loop vectorized with a SIMD construct must assure that temporary and reduction variables are privatized and declared as reductions with clauses. The example below illustrates the use of `private` and `reduction` clauses in a SIMD construct.
A `safelen(N)` clause in a `simd` construct assures the compiler that there are no loop-carried dependencies for vectors of size $N$ or below. If the `safelen` clause is not specified, then the default safelen value is the number of loop iterations.

The `safelen(16)` clause in the example below guarantees that the vector code is safe for vectors up to and including size 16. In the loop, $m$ can be 16 or greater, for correct code execution. If the value of $m$ is less than 16, the behavior is undefined.
The following SIMD construct instructs the compiler to collapse the \( i \) and \( j \) loops into a single SIMD loop in which SIMD chunks are executed by threads of the team. Within the workshared loop chunks of a thread, the SIMD chunks are executed in the lanes of the vector units.
Example SIMD.5.f90

```fortran
subroutine work( a, b, c, n )
  implicit none
  integer :: i,j,n
  double precision :: a(n,n), b(n,n), c(n,n), tmp
  !$omp do simd collapse(2) private(tmp)
  do j = 1,n
    do i = 1,n
      tmp = a(i,j) + b(i,j)
      c(i,j) = tmp
    end do
  end do
end subroutine work
```

## 5.2 inbranch and notinbranch Clauses

The following examples illustrate the use of the `declare simd` construct with the `inbranch` and `notinbranch` clauses. The `notinbranch` clause informs the compiler that the function `foo` is never called conditionally in the SIMD loop of the function `myaddint`. On the other hand, the `inbranch` clause for the function `goo` indicates that the function is always called conditionally in the SIMD loop inside the function `myaddfloat`. 
Example SIMD.6.c

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

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

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

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

Example SIMD.6.f90

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

function myaddint(a, b, n) result(r)
```
In the code below, the function \texttt{fib()} is called in the main program and also recursively called in the function \texttt{fib()} within an \texttt{if} condition. The compiler creates a masked vector version and a non-masked vector version for the function \texttt{fib()} while retaining the original scalar version of the \texttt{fib()} function.
Example SIMD.7.c

```c
#include <stdio.h>
#include <stdlib.h>
#define N 45
int a[N], b[N], c[N];
#pragma omp declare simd inbranch
int fib( int n )
{
    if (n <= 1)
        return n;
    else {
        return fib(n-1) + fib(n-2);
    }
}
```

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

Example SIMD.7.f90

```fortran
program fibonacci
implicit none
integer,parameter :: N=45
integer :: a(0:N-1), b(0:N-1)
integer :: i
integer, external :: fib
!$omp simd
do i = 0,N-1
```

OpenMP Examples Version 4.5.0 - November 2016
The following example tests the restriction on an SIMD loop with the loop-carried lexical forward-dependence. This dependence must be preserved for the correct execution of SIMD loops.

A loop can be vectorized even though the iterations are not completely independent when it has loop-carried dependences that are forward lexical dependences, indicated in the code below by the read of $A[j+1]$ and the write to $A[j]$ in C/C++ code (or $A(j+1)$ and $A(j)$ in Fortran). That is, the read of $A[j+1]$ (or $A(j+1)$ in Fortran) before the write to $A[j]$ (or $A(j)$ in Fortran) ordering must be preserved for each iteration in $j$ for valid SIMD code generation.

This test assures that the compiler preserves the loop carried lexical forward-dependence for generating a correct SIMD code.
Example SIMD.8.c

#include <stdio.h>
#include <math.h>

int P[1000];
float A[1000];

float do_work(float *arr)
{
    float pri;
    int i;
    #pragma omp simd lastprivate(pri)
    for (i = 0; i < 999; ++i) {
        int j = P[i];
        pri = 0.5f;
        if (j % 2 == 0) {
            pri = A[j+1] + arr[i];
        }
        A[j] = pri * 1.5f;
        pri = pri + A[j];
    }
    return pri;
}

int main(void)
{
    float pri, arr[1000];
    int i;
    for (i = 0; i < 1000; ++i) {
        P[i] = i;
        A[i] = i * 1.5f;
        arr[i] = i * 1.8f;
    }
    pri = do_work(&arr[0]);
    if (pri == 8237.25) {
        printf("passed: result pri = %.7f (8237.25) \n", pri);
    }
    else {
        printf("failed: result pri = %.7f (8237.25) \n", pri);
    }
    return 0;
}
Example SIMD.8.f90

module work
integer :: P(1000)
real :: A(1000)
contains
function do_work(arr) result(pri)
implicit none
real, dimension(*) :: arr
real :: pri
integer :: i, j
!$omp simd private(j) lastprivate(pri)
do i = 1, 999
    j = P(i)
pri = 0.5
    if (mod(j-1, 2) == 0) then
        pri = A(j+1) + arr(i)
    endif
    A(j) = pri * 1.5
    pri = pri + A(j)
end do
end function do_work
end module work

program simd_8f
use work
implicit none
real :: pri, arr(1000)
integer :: i
do i = 1, 1000
    P(i) = i
    A(i) = (i-1) * 1.5
    arr(i) = (i-1) * 1.8
end do
pri = do_work(arr)
if (pri == 8237.25) then
S-43    print 2, "passed", pri
S-44    else
S-45    print 2, "failed", pri
S-46    endif
S-47    2 format(a, ": result pri = ", f7.2, " (8237.25)"")
S-48    end program
CHAPTER 6

Synchronization

The barrier construct is a stand-alone directive that requires all threads of a team (within a contention group) to execute the barrier and complete execution of all tasks within the region, before continuing past the barrier.

The critical construct is a directive that contains a structured block. The construct allows only a single thread at a time to execute the structured block (region). Multiple critical regions may exist in a parallel region, and may act cooperatively (only one thread at a time in all critical regions), or separately (only one thread at a time in each critical regions when a unique name is supplied on each critical construct). An optional (lock) hint clause may be specified on a named critical construct to provide the OpenMP runtime guidance in selection a locking mechanism.

On a finer scale the atomic construct allows only a single thread at a time to have atomic access to a storage location involving a single read, write, update or capture statement, and a limited number of combinations when specifying the capture atomic-clause clause. The atomic-clause clause is required for some expression statements, but are not required for update statements. Please see the details in the atomic Construct subsection of the Directives chapter in the OpenMP Specifications document.

The ordered construct either specifies a structured block in a loop, simd, or loop SIMD region that will be executed in the order of the loop iterations. The ordered construct sequentializes and orders the execution of ordered regions while allowing code outside the region to run in parallel.

Since OpenMP 4.5 the ordered construct can also be a stand-alone directive that specifies cross-iteration dependences in a doacross loop nest. The depend clause uses a sink dependence-type, along with a iteration vector argument (vec) to indicate the iteration that satisfies the dependence. The depend clause with a source dependence-type specifies dependence satisfaction.

The flush directive is a stand-alone construct that forces a thread’s temporal local storage (view) of a variable to memory where a consistent view of the variable storage can be accesses. When the construct is used without a variable list, all the locally thread-visible data as defined by the base language are flushed. A construct with a list applies the flush operation only to the items in the list.
The **flush** construct also effectively insures that no memory (load or store) operation for the variable set (list items, or default set) may be reordered across the **flush** directive.

General-purpose routines provide mutual exclusion semantics through locks, represented by lock variables. The semantics allows a task to *set*, and hence *own* a lock, until it is *unset* by the task that set it. A *nestable* lock can be set multiple times by a task, and is used when in code requires nested control of locks. A *simple lock* can only be set once by the owning task. There are specific calls for the two types of locks, and the variable of a specific lock type cannot be used by the other lock type.

Any explicit task will observe the synchronization prescribed in a **barrier** construct and an implied barrier. Also, additional synchronizations are available for tasks. All children of a task will wait at a **taskwait** (for their siblings to complete). A **taskgroup** construct creates a region in which the current task is suspended at the end of the region until all sibling tasks, and their descendants, have completed. Scheduling constraints on task execution can be prescribed by the **depend** clause to enforce dependence on previously generated tasks. More details on controlling task executions can be found in the *Tasking* Chapter in the OpenMP Specifications document.
The following example includes several critical constructs. The example illustrates a queuing model in which a task is dequeued and worked on. To guard against multiple threads dequeuing the same task, the dequeuing operation must be in a critical region. Because the two queues in this example are independent, they are protected by critical constructs with different names, xaxis and yaxis.

```
Example critical.1.c

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

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

Example critical.1.f

```
SUBROUTINE CRITICAL_EXAMPLE(X, Y)

REAL X(*), Y(*)
INTEGER IX_NEXT, IY_NEXT

!$OMP PARALLEL SHARED(X, Y) PRIVATE(IX_NEXT, IY_NEXT)

!$OMP CRITICAL(XAXIS)
CALL DEQUEUE(IX_NEXT, X)
!$OMP END CRITICAL(XAXIS)
!$OMP CRITICAL(YAXIS)
CALL WORK(IY_NEXT, Y)
!$OMP END CRITICAL(YAXIS)
```

CHAPTER 6. SYNCHRONIZATION
The following example extends the previous example by adding the `hint` clause to the `critical` constructs.

```
#include <omp.h>

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

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

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

        #pragma omp critical (yaxis) hint(omp_lock_hint_contended)
        iy_next = dequeue(y);
        work(iy_next, y);
    }
}
```
SUBROUTINE CRITICAL EXAMPLE(X, Y)
USE OMP_LIB ! or INCLUDE "omp_lib.h"

REAL X(*), Y(*)
INTEGER IX_NEXT, IY_NEXT

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

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

!$OMP END PARALLEL
END SUBROUTINE CRITICAL EXAMPLE
6.2 Worksharing Constructs Inside a critical Construct

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

--- C / C+++ ---

Example worksharing_critical.1.c

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

--- C / C+++ ---
SUBROUTINE CRITICAL_WORK()

INTEGER I

I = 1

!$OMP PARALLEL SECTIONS
!$OMP SECTION
!$OMP CRITICAL (NAME)
!$OMP PARALLEL
!$OMP SINGLE
I = I + 1
!$OMP END SINGLE
!$OMP END PARALLEL
!$OMP END CRITICAL (NAME)
!$OMP END PARALLEL SECTIONS
END SUBROUTINE CRITICAL_WORK
6.3 Binding of barrier Regions

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

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

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

```
Example barrier_regions.1.c

S-1 void work(int n) {}  
S-2 void sub3(int n)  
S-3 {  
S-4   work(n);  
S-5   #pragma omp barrier  
S-6   work(n);  
S-7 }  
S-8 void sub2(int k)  
S-9 {  
S-10   #pragma omp parallel shared(k)  
S-11     sub3(k);  
S-12 }  
S-13 void sub1(int n)  
S-14 {  
S-15   int i;  
S-16   #pragma omp parallel private(i) shared(n)  
S-17     {  
S-18       #pragma omp for  
S-19         for (i=0; i<n; i++)  
S-20           sub2(i);  
S-21     }  
S-22 }  
S-23 void main()  
S-24 {  
S-25   sub1(2);  
S-26   sub2(2);
```
Example barrier_regions.1.f

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

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

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

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

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

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

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

Note that the \texttt{atomic} directive applies only to the statement immediately following it. As a result, elements of \( y \) are not updated atomically in this example.

\begin{verbatim}
C / C++

Example atomic.1.c

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

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

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

int main()
{
    float x[1000];
    float y[10000];
    int index[10000];
    int i;
    for (i = 0; i < 10000; i++) {
        index[i] = i % 1000;
        y[i]=0.0;
    }
}
\end{verbatim}
for (i = 0; i < 1000; i++)
    x[i] = 0.0;
atomic_example(x, y, index, 10000);
return 0;

Example atomic.1.f

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

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

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

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

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

    DO I=1,10000
        INDEX(I) = MOD(I, 1000) + 1
        Y(I) = 0.0
    ENDDO
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
int atomic_read(const int *p)
{
    int value;
    /* Guarantee that the entire value of *p is read atomically. No part of *p can change during the read operation. */
    #pragma omp atomic read
    value = *p;
    return value;
}

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

C / C++
Example atomic.2.f

```fortran
function atomic_read(p)
    integer :: atomic_read
    integer, intent(in) :: p

    ! Guarantee that the entire value of p is read atomically. No part of p can change during the read operation.

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

subroutine atomic_write(p, value)
    integer, intent(out) :: p
    integer, intent(in) :: value

    ! Guarantee that value is stored atomically into p. No part of p can change until after the entire write operation is completed.

    !$omp atomic write
    p = value
end subroutine atomic_write
```

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

Example atomic.3.c

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

/*
/* Use fetch_and_add to implement a lock */

struct locktype {
  int ticketnumber;
  int turn;
};

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

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

  // Do some work. The flush is needed to ensure visibility of
  // variables not involved in atomic directives
  #pragma omp flush
  work();
  #pragma omp flush

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

Example atomic.3.f

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

module m

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

end module m
integer :: atomic_read
integer, intent(in) :: p
end function
end interface
type locktype
  integer ticketnumber
  integer turn
end type
contains
subroutine do_locked_work(lock)
type(locktype), intent(inout) :: lock
integer myturn
integer junk
! obtain the lock
  myturn = fetch_and_add(lock%ticketnumber)
  do while (atomic_read(lock%turn) .ne. myturn)
    continue
  enddo
! Do some work. The flush is needed to ensure visibility of variables not involved in atomic directives
!$omp flush
call work
!$omp flush
! Release the lock
  junk = fetch_and_add(lock%turn)
end subroutine
end module
6.5 Restrictions on the atomic Construct

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

Example atomic_restrict.1.c

```c
void atomic_wrong ()
{
    union {int n; float x;} u;
    #pragma omp parallel
    {
        #pragma omp atomic update
        u.n++;
        #pragma omp atomic update
        u.x += 1.0;
    /* Incorrect because the atomic constructs reference the same location 
    through incompatible types */
    }
}
```

Example atomic_restrict.1.f

```fortran
SUBROUTINE ATOMIC_WRONG()
    INTEGER:: I
    REAL:: R
    EQUIVALENCE(I,R)
    !$OMP PARALLEL
    !$OMP ATOMIC UPDATE
    I = I + 1
    !$OMP ATOMIC UPDATE
    R = R + 1.0
    ! incorrect because I and R reference the same location 
    ! but have different types
    !$OMP END PARALLEL
END SUBROUTINE ATOMIC_WRONG
```

OpenMP Examples Version 4.5.0 - November 2016
Example atomic_restrict.2.c

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

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

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

        #pragma omp atomic update
        *r += 1.0;

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

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

Example atomic_restrict.2.f

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

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

SUBROUTINE ATOMIC_WRONG2()
    COMMON /BLK/ I
    INTEGER I

!$OMP PARALLEL
```
Although the following example might work on some implementations, this is also non-conforming:

Example atomic_re restrict.3.f

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

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

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

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

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

```
Example flush_nolist.1.c

int x, *p = &x;

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

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

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

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

Example flush_nolist.1.f

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

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

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

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

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

OpenMP Examples Version 4.5.0 - November 2016
INTEGER N
INTEGER I, J, SUM

I = 1
SUM = 0
P = 1

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

CALL F2(J)
! I, N, and SUM were not flushed
! because they were not accessible in f2
! J was flushed because it was accessible
SUM = SUM + I + J + P + N

!$OMP END PARALLEL

G = SUM
END FUNCTION G

PROGRAM FLUSH_NOLIST
COMMON /DATA/ X, P
INTEGER, TARGET :: X
INTEGER, POINTER :: P
INTEGER RESULT, G
P => X
RESULT = G(7)
PRINT *, RESULT
END PROGRAM FLUSH_NOLIST
6.7 The ordered Clause and the ordered Construct

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

--- C / C++ ---

```c
#include <stdio.h>

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

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

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

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

--- C / C++ ---

--- Fortran ---

```fortran
SUBROUTINE WORK(K)
    INTEGER k

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

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

    END SUBROUTINE SUB
```

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

```c
S-1
void work(int i) {}
S-2
void ordered_wrong(int n)
S-3
{
S-4
    int i;
S-5
    #pragma omp for ordered
S-6
    for (i=0; i<n; i++) {
S-7          /* incorrect because an iteration may not execute more than one
S-8                              ordered region */
S-9          #pragma omp ordered
S-10         work(i);
S-11         #pragma omp ordered
S-12         work(i+1);
S-13      }
S-14      }
S-15      }
```
Fortran

Example ordered.2.f

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

SUBROUTINE ORDERED_WRONG(N)
INTEGER N

INTEGER I

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

!$OMP ORDERED
CALL WORK(I+1)
!$OMP END ORDERED
END DO
END SUBROUTINE ORDERED_WRONG
```

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

C / C++

Example ordered.3.c

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

void ordered_good(int n)
{
  int i;
  #pragma omp for ordered
  for (i=0; i<n; i++) {
    if (i <= 10) {
      #pragma omp ordered
      work(i);
    }
    if (i > 10) {
      #pragma omp ordered
      work(i+1);
    }
  }
```
Example ordered.3.f

```fortran
SUBROUTINE ORDERED_GOOD(N

INTEGER N

!*OMP DO ORDERED
DO I = 1,N
  IF (I <= 10) THEN
    !$OMP ORDERED
    CALL WORK(I)
    !$OMP END ORDERED
  ENDIF
  IF (I > 10) THEN
    !$OMP ORDERED
    CALL WORK(I+1)
    !$OMP END ORDERED
  ENDIF
ENDDO
END SUBROUTINE ORDERED_GOOD
```
6.8 Doacross Loop Nest

An ordered clause can be used on a loop construct with an integer parameter argument to define the number of associated loops within a doacross loop nest where cross-iteration dependences exist. A depend clause on an ordered construct within an ordered loop describes the dependences of the doacross loops.

In the code below, the depend(sink: i-1) clause defines an i-1 to i cross-iteration dependence that specifies a wait point for the completion of computation from iteration i-1 before proceeding to the subsequent statements. The depend(source) clause indicates the completion of computation from the current iteration (i) to satisfy the cross-iteration dependence that arises from the iteration. For this example the same sequential ordering could have been achieved with an ordered clause without a parameter, on the loop directive, and a single ordered directive without the depend clause specified for the statement executing the bar function.

```
Example doacross.1.c
```

```c
float foo(int i);
float bar(float a, float b);
float baz(float b);

void work( int N, float *A, float *B, float *C )
{
    int i;

    #pragma omp for ordered(1)
    for (i=1; i<N; i++)
    {
        A[i] = foo(i);

        #pragma omp ordered depend(sink: i-1)
        B[i] = bar(A[i], B[i-1]);

        #pragma omp ordered depend(source)
        C[i] = baz(B[i]);
    }
}
```

---

C / C++
The following code is similar to the previous example but with *doacross loop nest* extended to two nested loops, \(i\) and \(j\), as specified by the *ordered(2)* clause on the loop directive. In the C/C++ code, the \(i\) and \(j\) loops are the first and second associated loops, respectively, whereas in the Fortran code, the \(j\) and \(i\) loops are the first and second associated loops, respectively. The *depend(sink:i-1,j)* and *depend(sink:i,j-1)* clauses in the C/C++ code define cross-iteration dependences in two dimensions from iterations \((i-1, j)\) and \((i, j-1)\) to iteration \((i, j)\). Likewise, the *depend(sink:j-1,i)* and *depend(sink:j,i-1)* clauses in the Fortran code define cross-iteration dependences from iterations \((j-1, i)\) and \((j, i-1)\) to iteration \((j, i)\).
The following example shows the incorrect use of the `ordered` directive with a `depend` clause. There are two issues with the code. The first issue is a missing `ordered depend(source)` directive, which could cause a deadlock. The second issue is the `depend(sink:i+1,j)` and `depend(sink:i,j+1)` clauses define dependences on lexicographically later source iterations $(i+1,j)$ and $(i,j+1)$, which could cause a deadlock as well since they may not start to execute until the current iteration completes.
Example doacross.3.c

```c
#define N 100

void work_wrong(double p[][N][N])
{
  int i, j, k;

  #pragma omp parallel for ordered(2) private(i,j,k)
  for (i=1; i<N-1; i++)
    {
      for (j=1; j<N-1; j++)
        {
          #pragma omp ordered depend(sink: i-1,j) depend(sink: i+1,j) \
            depend(sink: i,j-1) depend(sink: i,j+1)
          for (k=1; k<N-1; k++)
            {
              double tmp1 = p[i-1][j][k] + p[i+1][j][k];
              double tmp2 = p[i][j-1][k] + p[i][j+1][k];
              double tmp3 = p[i][j][k-1] + p[i][j][k+1];
              p[i][j][k] = (tmp1 + tmp2 + tmp3) / 6.0;
            }
        }
    }
}
```

Example doacross.3.f90

```fortran```
subroutine work_wrong(N, p)
  integer :: N
  real(8), dimension(N,N,N) :: p
  integer :: i, j, k
  real(8) :: tmp1, tmp2, tmp3

!$omp parallel do ordered(2) private(i,j,k,tmp1,tmp2,tmp3)
  do i=2, N-1
    do j=2, N-1
      !$omp ordered depend(sink: i-1,j) depend(sink: i+1,j) &
      !$omp& depend(sink: i,j-1) depend(sink: i,j+1)
      do k=2, N-1
        tmp1 = p(k-1,j,i) + p(k+1,j,i)
        tmp2 = p(k,j-1,i) + p(k,j+1,i)
        tmp3 = p(k,j,i-1) + p(k,j,i+1)
      end do
    end do
  end do
end subroutine work_wrong```

CHAPTER 6. SYNCHRONIZATION 197
The following example illustrates the use of the `collapse` clause for a *doacross loop nest*. The *i* and *j* loops are the associated loops for the collapsed loop as well as for the *doacross loop nest*. The example also shows a compliant usage of the dependence source directive placed before the corresponding sink directive. Checking the completion of computation from previous iterations at the sink point can occur after the source statement.
**Example doacross.4.f90**

```fortran
subroutine work( N, M, A, B, C )

integer :: N, M
real(8), dimension(M, N) :: A, B, C
real(8), external :: foo
integer :: i, j
real(8) :: alpha = 1.2

!$omp do collapse(2) ordered(2)
do j=2, N-1
  do i=2, M-1
    A(i,j) = foo(i, j)
    !$omp ordered depend(source)
    B(i,j) = alpha * A(i,j)
    !$omp ordered depend(sink: j,i-1) depend(sink: j-1,i)
    C(i,j) = 0.2 * (A(i-1,j) + A(i+1,j) + &
                  A(i,j-1) + A(i,j+1) + A(i,j))
  end do
end do
end subroutine
```
6.9 Lock Routines

This section is about the use of lock routines for synchronization.

6.9.1 The omp_init_lock Routine

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

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

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

```fortran
FUNCTION NEW_LOCKS()
    USE OMP_LIB ! or INCLUDE "omp_lib.h"
    INTEGER(OMP_LOCK_KIND), DIMENSION(1000) :: NEW_LOCKS
    INTEGER I
    !$OMP PARALLEL DO PRIVATE(I)
    DO I=1,1000
        CALL OMP_INIT_LOCK(NEW_LOCKS(I))
    END DO
    !$OMP END PARALLEL DO
END FUNCTION NEW_LOCKS
```
6.9.2 The omp_init_lock_with_hint Routine

The following example demonstrates how to initialize an array of locks in a parallel region by using omp_init_lock_with_hint. Note, hints are combined with an | or + operator in C/C++ and a + operator in Fortran.

Example init_lock_with_hint.1.cpp

```cpp
#include <omp.h>

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

    #pragma omp parallel for private(i)
    for (i=0; i<1000; i++)
    {
        omp_init_lock_with_hint(&lock[i],
                                omp_lock_hint_contended | omp_lock_hint_speculative);
    }
    return lock;
}
```

Example init_lock_with_hint.1.f

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

    INTEGER I

    !$OMP PARALLEL DO PRIVATE(I)
    DO I=1,1000
        CALL OMP_INIT_LOCK_WITH_HINT(NEW_LOCKS(I),
                                     OMP_LOCK_HINT_CONTENDED + OMP_LOCK_HINT_SPECULATIVE)
    END DO
    !$OMP END PARALLEL DO
END FUNCTION NEW_LOCKS
```
Ownership of locks has changed since OpenMP 2.5. In OpenMP 2.5, locks are owned by threads; so a lock released by the `omp_unset_lock` routine must be owned by the same thread executing the routine. Beginning with OpenMP 3.0, locks are owned by task regions; so a lock released by the `omp_unset_lock` routine in a task region must be owned by the same task region.

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

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

int main()
{
    int x;
    omp_lock_t lck;
    omp_init_lock (&lck);
    omp_set_lock (&lck);
    x = 0;
    #pragma omp parallel shared (x)
    {
        #pragma omp master
        {
            x = x + 1;
            omp_unset_lock (&lck);
        }
        /* Some more stuff. */
    }
    omp_destroy_lock (&lck);
    return 0;
}
```
Example lock_owner.1.f

```fortran
program lock
  use omp_lib
  integer :: x
  integer (kind=omp_lock_kind) :: lck
  call omp_init_lock (lck)
  call omp_set_lock(lck)
  x = 0
  !$omp parallel shared (x)
  !$omp master
    x = x + 1
    call omp_unset_lock(lck)
  !$omp end master
  !$omp end parallel
  call omp_destroy_lock(lck)
end
```

### 6.9.4 Simple Lock Routines

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

Note that the argument to the lock routines should have type `omp_lock_t`, and that there is no need to flush it.
Example simple_lock.1.c

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

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

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

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

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

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

        work(id); /* we now have the lock
                    and can do the work */
        omp_unset_lock(&lck);
    }

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

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

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

SUBROUTINE WORK(ID)
END SUBROUTINE WORK

PROGRAM SIMPLELOCK

INCLUDE "omp_lib.h" ! or USE OMP_LIB

INTEGER(OMP_LOCK_KIND) LCK
INTEGER ID

CALL OMP_INIT_LOCK(LCK)

!$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
    ID = OMP_GET_THREAD_NUM()
    CALL OMP_SET_LOCK(LCK)
    PRINT *, 'My thread id is ', ID
    CALL OMP_UNSET_LOCK(LCK)
END PARALLEL

DO WHILE (.NOT. OMP_TEST_LOCK(LCK))
    CALL SKIP(ID) ! We do not yet have the lock
    ! so we must do something else
END DO

CALL WORK(ID) ! We now have the lock
! and can do the work

CALL OMP_UNSET_LOCK( LCK )

!$OMP END PARALLEL

CALL OMP_DESTROY_LOCK( LCK )

END PROGRAM SIMPLELOCK
```
6.9.5 Nestable Lock Routines

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

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

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

Example nestable_lock.1.c
Example nestable_lock.1.f

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

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

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

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

CHAPTER 6. SYNCHRONIZATION 207
SUBROUTINE NESTLOCK(P)
USE OMP_LIB ! or INCLUDE "omp_lib.h"
USE DATA
TYPE(LOCKED_PAIR) :: P
INTEGER WORK1, WORK2, WORK3
EXTERNAL WORK1, WORK2, WORK3

!$OMP PARALLEL SECTIONS
!$OMP SECTION
CALL INCR_PAIR(P, WORK1(), WORK2())
!$OMP SECTION
CALL INCR_B(P, WORK3())
!$OMP END PARALLEL SECTIONS

END SUBROUTINE NESTLOCK
The OpenMP data environment contains data attributes of variables and objects. Many constructs (such as parallel, simd, task) accept clauses to control data-sharing attributes of referenced variables in the construct, where data-sharing applies to whether the attribute of the variable is shared, is private storage, or has special operational characteristics (as found in the firstprivate, lastprivate, linear, or reduction clause).

The data environment for a device (distinguished as a device data environment) is controlled on the host by data-mapping attributes, which determine the relationship of the data on the host, the original data, and the data on the device, the corresponding data.

DATA-SHARING ATTRIBUTES

Data-sharing attributes of variables can be classified as being predetermined, explicitly determined or implicitly determined.

Certain variables and objects have predetermined attributes. A commonly found case is the loop iteration variable in associated loops of a for or do construct. It has a private data-sharing attribute. Variables with predetermined data-sharing attributes can not be listed in a data-sharing attribute clause; but there are some exceptions (mainly concerning loop iteration variables).

Variables with explicitly determined data-sharing attributes are those that are referenced in a given construct and are listed in a data-sharing attribute clause on the construct. Some of the common data-sharing clauses are: shared, private, firstprivate, lastprivate, linear, and reduction.

Variables with implicitly determined data-sharing attributes are those that are referenced in a given construct, do not have predetermined data-sharing attributes, and are not listed in a data-sharing attribute clause of an enclosing construct. For a complete list of variables and objects with predetermined and implicitly determined attributes, please refer to the Data-sharing Attribute Rules for Variables Referenced in a Construct subsection of the OpenMP Specifications document.
DATA-MAPPING ATTRIBUTES

The map clause on a device construct explicitly specifies how the list items in the clause are mapped from the encountering task’s data environment (on the host) to the corresponding item in the device data environment (on the device). The common list items are arrays, array sections, scalars, pointers, and structure elements (members).

Procedures and global variables have predetermined data mapping if they appear within the list or block of a declare target directive. Also, a C/C++ pointer is mapped as a zero-length array section, as is a C++ variable that is a reference to a pointer.

Without explicit mapping, non-scalar and non-pointer variables within the scope of the target construct are implicitly mapped with a map-type of tofrom. Without explicit mapping, scalar variables within the scope of the target construct are not mapped, but have an implicit firstprivate data-sharing attribute. (That is, the value of the original variable is given to a private variable of the same name on the device.) This behavior can be changed with the defaultmap clause.

The map clause can appear on target, target data and target enter/exit data constructs. The operations of creation and removal of device storage as well as assignment of the original list item values to the corresponding list items may be complicated when the list item appears on multiple constructs or when the host and device storage is shared. In these cases the item’s reference count, the number of times it has been referenced (+1 on entry and -1 on exited) in nested (structured) map regions and/or accumulative (unstructured) mappings, determines the operation. Details of the map clause and reference count operation are specified in the map Clause subsection of the OpenMP Specifications document.
7.1 The threadprivate Directive

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

Example threadprivate.1.c

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

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

Example threadprivate.1.f

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

The following example uses threadprivate on a static variable:

Example threadprivate.2.c

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

\textit{Example threadprivate.3.cpp}

```
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     for (int i=0; i<n; i++) {
S-34         g(a, b); /* Value of a is unspecified. */
S-35     }
S-36 }
S-37 }
```
The following examples show non-conforming uses and correct uses of the \texttt{threadprivate} directive.

\begin{verbatim}
Example threadprivate.2.f
S-1 MODULE INC_MODULE
S-2   COMMON /T/ A
S-3 END MODULE INC_MODULE
S-4 SUBROUTINE INC_MODULE_WRONG()
S-5   USE INC_MODULE
S-6   !$OMP THREADPRIVATE(/T/)
S-7 !non-conforming because /T/ not declared in INC_MODULE_WRONG
S-8 END SUBROUTINE INC_MODULE_WRONG
\end{verbatim}

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

\begin{verbatim}
Example threadprivate.3.f
S-1 SUBROUTINE INC_WRONG()
S-2   COMMON /T/ A
S-3   !$OMP THREADPRIVATE(/T/)
S-4 CONTAINS
S-5   SUBROUTINE INC_WRONG_SUB()
S-6   !$OMP PARALLEL COPYIN(/T/)
S-7 !non-conforming because /T/ not declared in INC_WRONG_SUB
S-8 !$OMP END PARALLEL
S-9 END SUBROUTINE INC_WRONG_SUB
S-10 END SUBROUTINE INC_WRONG
\end{verbatim}

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

\begin{verbatim}
Example threadprivate.4.f
S-1 SUBROUTINE INC_GOOD()
S-2   COMMON /T/ A
S-3   !$OMP THREADPRIVATE(/T/)
S-4 CONTAINS
S-5   SUBROUTINE INC_GOOD_SUB()
S-6   COMMON /T/ A
S-7   !$OMP THREADPRIVATE(/T/)
S-8 !$OMP PARALLEL COPYIN(/T/)
S-9 END SUBROUTINE INC_GOOD_SUB
S-10 END SUBROUTINE INC_GOOD
\end{verbatim}
The following is an example of the use of `threadprivate` for local variables:

```
Example threadprivate.5.f

PROGRAM INC_GOOD2
    INTEGER, ALLOCATABLE, SAVE :: A(:)
    INTEGER, POINTER, SAVE :: PTR
    INTEGER, SAVE :: I
    INTEGER, TARGET :: TARG
    LOGICAL :: FIRSTIN = .TRUE.

    !$OMP THREADPRIVATE(A, I, PTR)

    ALLOCATE (A(3))
    A = (/1,2,3/)
    PTR => TARG
    I = 5

    !$OMP PARALLEL COPYIN(I, PTR)
    !$OMP CRITICAL
    IF (FIRSTIN) THEN
        TARG = 4  ! Update target of ptr
        I = I + 10
        IF (ALLOCATED(A)) A = A + 10
        FIRSTIN = .FALSE.
    END IF
    IF (ALLOCATED(A)) THEN
        PRINT *, 'a = ', A
    ELSE
        PRINT *, 'A is not allocated'
    END IF
    PRINT *, 'ptr = ', PTR
    PRINT *, 'i = ', I
    PRINT *
    !$OMP END CRITICAL
    !$OMP END PARALLEL
END PROGRAM INC_GOOD2
```

The above program, if executed by two threads, will print one of the following two sets of output:
The following is an example of the use of `threadprivate` for module variables:

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

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

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

Example threadprivate.6.f
The following example illustrates initialization of \texttt{threadprivate} variables for class-type \texttt{T}. \texttt{t1} is default constructed, \texttt{t2} is constructed taking a constructor accepting one argument of integer type, \texttt{t3} is copy constructed with argument \texttt{f()}:

\begin{verbatim}
Example threadprivate.4.cpp
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)
\end{verbatim}

The following example illustrates the use of \texttt{threadprivate} for static class members. The \texttt{threadprivate} directive for a static class member must be placed inside the class definition.

\begin{verbatim}
Example threadprivate.5.cpp
S-1  class T {
S-2    public:
S-3      static int i;
S-4    #pragma omp threadprivate(i)
S-5    }
\end{verbatim}
7.2 The default (none) Clause

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

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

Example default_none.1.c

```c
#include <omp.h>

int x, y, z[1000];
#pragma omp threadprivate(x)

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

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

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

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

CHAPTER 7. DATA ENVIRONMENT 217
SUBROUTINE DEFAULT_NONE(A)

  INCLUDE "omp_lib.h" ! or USE OMP_LIB

  INTEGER A

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

  !$OMP THREADPRIVATE(/BLOCKX/)

  INTEGER I, J
  i = 1

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

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

  END PARALLEL

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

  !$OMP END PARALLEL

END SUBROUTINE DEFAULT_NONE
7.3 The private Clause

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

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

int main()
{
    int i, j;
    int *ptr_i, *ptr_j;
    i = 1;
    j = 2;
    ptr_i = &i;
    ptr_j = &j;

    #pragma omp parallel private(i) firstprivate(j)
    {
        i = 3;
        j = j + 2;
        assert (*ptr_i == 1 && *ptr_j == 2);
    }
    assert(i == 1 && j == 2);
    return 0;
}
```
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.
The following example demonstrates that a list item that appears in a `private` clause in a `parallel` construct may also appear in a `private` clause in an enclosed worksharing construct, which results in an additional private copy.

---

### Example private.2.f

```fortran
MODULE PRIV_EXAMPLE2
    REAL A
    CONTAINS
    SUBROUTINE G(K)
        REAL K
        A = K ! Accessed in the region but outside of the construct; therefore unspecified whether original or private list item is modified.
    END SUBROUTINE G
    SUBROUTINE F(N)
        INTEGER N
        REAL A
        INTEGER I
        !$OMP PARALLEL DO PRIVATE(A)
        DO I = 1,N
            A = I
            CALL G(A*2)
        ENDDO
        !$OMP END PARALLEL DO
    END SUBROUTINE F
END MODULE PRIV_EXAMPLE2
```

---

### Example private.3.c

```c
#include <assert.h>
void priv_example3()
{
    int i, a;
    #pragma omp parallel private(a)
    {
        a = 1;
        #pragma omp parallel for private(a)
```
```c
for (i=0; i<10; i++)
{
    a = 2;
}
assert(a == 1);
```
7.4 Fortran Private Loop Iteration Variables

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

*Example* `fort_loopvar.1.f90`

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

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

*Example* `fort_loopvar.2.f90`

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

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

The following example is conforming:

*Example fort_sp_common.1.f*

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

The following example is also conforming:

*Example fort_sp_common.2.f*

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

Example `fort_sp_common.3.f`

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

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

Example `fort_sp_common.4.f`

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

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

Example `fort_sp_common.5.f`

```
SUBROUTINE COMMON_WRONG2()
  COMMON /C/ X,Y
  ! Incorrect: common block C cannot be declared both
  ! shared and private
  !$OMP PARALLEL PRIVATE (/C/), SHARED(/C/)
  ! do work here
  !$OMP END PARALLEL
END SUBROUTINE COMMON_WRONG2
```
7.6 Fortran Restrictions on Storage Association with the `private` Clause

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

*Example fort_sa_private.1.f*

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

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

*Example fort_sa_private.2.f*

```fortran
PROGRAM PRIV_RESTRICT2
    COMMON /BLOCK2/ X
    X = 1.0
    !$OMP PARALLEL PRIVATE (X)
    X = 2.0
    CALL SUB()
    !$OMP END PARALLEL

CONTAINS

SUBROUTINE SUB()
    COMMON /BLOCK2/ Y
    PRINT *,X ! X is undefined
    PRINT *,Y ! Y is undefined
END SUBROUTINE SUB

END PROGRAM PRIV_RESTRICT2
```

*Example fort_sa_private.3.f*
PROGRAM PRIV_RESTRICT3
EQUIVALENCE (X,Y)
X = 1.0
!
$OMP PARALLEL PRIVATE(X)
PRINT *,Y ! Y is undefined
Y = 10
PRINT *,X ! X is undefined
$OMP END PARALLEL
END PROGRAM PRIV_RESTRICT3

Example fort_sa_private.4.f

PROGRAM PRIV_RESTRICT4
INTEGER I, J
INTEGER A(100), B(100)
EQUIVALENCE (A(51), B(1))
!
$OMP PARALLEL DO DEFAULT(PRIVATE) PRIVATE(I,J) LASTPRIVATE(A)
DO I=1,100
  DO J=1,100
    B(J) = J - 1
  ENDDO
ENDDO
!
DO J=1,50
  B(J) = B(J) + 1 ! B is undefined
  A(J) = J ! B becomes undefined at this point
ENDDO
!
DO J=1,1,100
  A(J) = J ! A becomes undefined at this point
ENDDO
ENDDO

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

Example fort_sa_private.5.f

SUBROUTINE SUB1(X)
DIMENSION X(10)
!
! This use of X does not conform to the


! specification. It would be legal Fortran 90, but the OpenMP private directive allows the compiler to break the sequence association that A had with the rest of the common block.

FORALL (I = 1:10) X(I) = I

END SUBROUTINE SUB1

PROGRAM PRIV_RESTRICT5

COMMON /BLOCK5/ A

DIMENSION B(10)

EQUIVALENCE (A,B(1))

! the common block has to be at least 10 words

A = 0

!$OMP PARALLEL PRIVATE(/BLOCK5/)

! Without the private clause, we would be passing a member of a sequence that is at least ten elements long.

! With the private clause, A may no longer be sequence-associated.

CALL SUB1(A)

!$OMP MASTER

PRINT *, A

!$OMP END MASTER

!$OMP END PARALLEL

END PROGRAM PRIV_RESTRICT5
The following example illustrates the size and value of list items of array or pointer type in a `firstprivate` clause. The size of new list items is based on the type of the corresponding original list item, as determined by the base language.

In this example:

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

Note that B and E involve variable length array types.

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

Example carrays_fpriv.1.c

```c
#include <assert.h>

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

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

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

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

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

assert(n != 0);
```

int main() {
    f(2, A, A[0]);
    return 0;
}
7.8 The lastprivate Clause

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

---

Example lastprivate.1.c

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

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

---

Example lastprivate.1.f

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

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

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

\begin{verbatim}
SUBROUTINE REDUCTION1(A, B, C, D, X, Y, N)
    REAL :: X(*), A, D
    INTEGER :: Y(*), N, B, C
    INTEGER :: I
    A = 0
    B = 0
    C = Y(1)
    D = X(1)
!$OMP PARALLEL DO PRIVATE(I) SHARED(X, Y, N) REDUCTION(:+A) &
!$OMP& REDUCTION(IEOR:B) REDUCTION(+:C) REDUCTION(MAX:D)
    DO I=1,N
        A = A + X(I)
        B = IEOR(B, Y(I))
        C = MIN(C, Y(I))
        IF (D < X(I)) D = X(I)
\end{verbatim}
A common implementation of the preceding example is to treat it as if it had been written as follows:

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

---

Fortran

```fortran
END DO
END SUBROUTINE REDUCTION1
```

---

C / C++
Example reduction.2.f90

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

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

Example reduction.3.f90

PROGRAM REDUCTION_WRONG
MAX = HUGE(0)
M = 0
!$OMP PARALLEL DO REDUCTION(MAX: M)
! MAX is no longer the intrinsic so this is non-conforming
DO I = 1, 100
CALL SUB(M, I)
END DO
The following conforming program performs the reduction using the *intrinsic procedure name* MAX
even though the intrinsic MAX has been renamed to REN.

*Example reduction.4.f90*

```
MODULE M
  INTRINSIC MAX
END MODULE M

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

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

*Example reduction.5.f90*

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

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

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

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

\[\]

\textit{Example reduction.6.c}

\begin{verbatim}
#include <stdio.h>

int main (void)
{
    int a, i;

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

        // To avoid race conditions, add a barrier here.

        #pragma omp for reduction(+:a)
        for (i = 0; i < 10; i++) {
            a += i;
        }

        #pragma omp single
        printf ("Sum is %d\n", a);
    }
    return 0;

}\end{verbatim}

C / C++

CHAPTER 7. DATA ENVIRONMENT 237
The following example demonstrates the reduction of array $a$. In C/C++ this is illustrated by the explicit use of an array section $a[0:N]$ in the reduction clause. The corresponding Fortran example uses array syntax supported in the base language. As of the OpenMP 4.5 specification the explicit use of array section in the reduction clause in Fortran is not permitted. But this oversight will be fixed in the next release of the specification.
Example reduction.7.f90

```
program array_red

integer, parameter :: n=100
integer :: j
real :: a(n), b(n,n)

call init(n,b)
a(:) = 0.0e0
!
omp parallel do reduction(+:a)
do j = 1, n
   a(:) = a(:) + b(:,j)
end do
print*, " a(1) a(n): ", a(1), a(n)
end program
```

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

```c
#include <stdlib.h>

float* work;
int size;
float tol;

#pragma omp threadprivate(work,size,tol)

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

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

This example demonstrates the use of the **copyin** clause to initialize threadprivate variables upon entry to a parallel region. The `build()` function initializes the threadprivate data, and the `copyin_example()` function copies the threadprivate values from the master thread to each team member.
Example copyin.1.f

1

MODULE M

REAL, POINTER, SAVE :: WORK(:)

INTEGER :: SIZE

REAL :: TOL

!$OMP THREADPRIVATE(WORK,SIZE,TOL)

END MODULE M

SUBROUTINE COPYIN_EXAMPLE( T, N )

USE M

REAL :: T

INTEGER :: N

TOL = T

SIZE = N

!$OMP PARALLEL COPYIN(TOL,SIZE)

CALL BUILD

!$OMP END PARALLEL

END SUBROUTINE COPYIN_EXAMPLE

SUBROUTINE BUILD

USE M

ALLOCATE(WORK(SIZE))

WORK = TOL

END SUBROUTINE BUILD
7.11 The *copyprivate* Clause

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

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

### Example copyprivate.1.c

```c
#include <stdio.h>

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

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

### Example copyprivate.1.f

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

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

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

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

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

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

    #pragma omp barrier

    return_val = *tmp;

    #pragma omp barrier

    #pragma omp single nowait
    
    free(tmp);

    return return_val;
}
```

Example copyprivate.2.f

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

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

!$OMP MASTER
READ (11) TMP

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

Example copyprivate.3.c

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

omp_lock_t *new_lock()
{
    omp_lock_t *lock_ptr;

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

    return lock_ptr;
}
```

---

Fortran

```
S-11       !$OMP  BARRIER
S-12       READ_NEXT = TMP
S-13       !$OMP  BARRIER
S-14       !$OMP  SINGLE
S-15       DEALLOCATE (TMP)
S-16       !$OMP  END SINGLE NOWAIT
S-17       END FUNCTION READ_NEXT
```

---

C / C++
Example copyprivate.3.f

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

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

Example copyprivate.4.f

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 !$OMP DEALLOCATE (B)
S-19 !$OMP END SINGLE NOWAIT
S-20 END SUBROUTINE S
C++ Reference in Data-Sharing Clauses

C++ reference types are allowed in data-sharing attribute clauses as of OpenMP 4.5, except for the `threadprivate`, `copyin` and `copyprivate` clauses. (See the Data-Sharing Attribute Clauses Section of the 4.5 OpenMP specification.) When a variable with C++ reference type is privatized, the object the reference refers to is privatized in addition to the reference itself. The following example shows the use of reference types in data-sharing clauses in the usual way. Additionally it shows how the data-sharing of formal arguments with a C++ reference type on an orphaned task generating construct is determined implicitly. (See the Data-sharing Attribute Rules for Variables Referenced in a Construct Section of the 4.5 OpenMP specification.)

Example `cpp_reference.1.cpp`

```cpp
void task_body (int &);  
void gen_task (int &x) { // on orphaned task construct reference argument
    #pragma omp task // x is implicitly determined firstprivate(x)
    task_body (x);
}
void test (int &y, int &z) {
    #pragma omp parallel private(y)
    {
        y = z + 2;
        gen_task (y); // no matter if the argument is determined private
        gen_task (z); // or shared in the enclosing context.
        y++; // each thread has its own int object y refers to
        gen_task (y);
    }
}
```
7.13 Fortran ASSOCIATE Construct

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

Example associate.1.f

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

Example associate.2.f

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

In next example, within the parallel construct, the association name \( \text{thread}_\text{id} \) is associated with the private copy of \( i \). The print statement should output the unique thread number.

Example associate.2.f

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

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

Example associate.3.f90

```fortran
```
program example
integer :: v
v = 15
associate(u => v)
!$omp parallel private(v)
v = -1
print *, v  ! private v=-1
print *, u   ! original v=15
!$omp end parallel
end associate
end program
In this chapter, examples illustrate race conditions on access to variables with shared data-sharing attributes. A race condition can exist when two or more threads are involved in accessing a variable in which not all of the accesses are reads; that is, a WaR, RaW or WaW condition exists (R=read, a=after, W=write). A RaR does not produce a race condition. Ensuring thread execution order at the processor level is not enough to avoid race conditions, because the local storage at the processor level (registers, caches, etc.) must be synchronized so that a consistent view of the variable in the memory hierarchy can be seen by the threads accessing the variable.

OpenMP provides a shared-memory model which allows all threads access to memory (shared data). Each thread also has exclusive access to threadprivate memory (private data). A private variable referenced in an OpenMP directive’s structured block is a new version of the original variable (with the same name) for each task (or SIMD lane) within the code block. A private variable is initially undefined (except for variables in firstprivate and linear clauses), and the original variable value is unaltered by assignments to the private variable, (except for reduction, lastprivate and linear clauses).

Private variables in an outer parallel region can be shared by implicit tasks of an inner parallel region (with a share clause on the inner parallel directive). Likewise, a private variable may be shared in the region of an explicit task (through a shared clause).

The flush directive forces a consistent view of local variables of the thread executing the flush. When a list is supplied on the directive, only the items (variables) in the list are guaranteed to be flushed.

Implied flushes exist at prescribed locations of certain constructs. For the complete list of these locations and associated constructs, please refer to the flush Construct section of the OpenMP Specifications document.

Examples 1-3 show the difficulty of synchronizing threads through flush and atomic directives.
8.1 The OpenMP Memory Model

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

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

---

Example mem_model.1.c

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

int main(){
    int x;
    x = 2;
    #pragma omp parallel num_threads(2) shared(x)
    {
        if (omp_get_thread_num() == 0) {
            x = 5;
        } else {
            /* Print 1: the following read of x has a race */
            printf("1: Thread# %d: x = %d\n", omp_get_thread_num(),x );
        }
    }
    #pragma omp barrier
    if (omp_get_thread_num() == 0) {
        /* Print 2 */
        printf("2: Thread# %d: x = %d\n", omp_get_thread_num(),x );
    } else {
        /* Print 3 */
        printf("3: Thread# %d: x = %d\n", omp_get_thread_num(),x );
    }
    return 0;
}
```
The following example demonstrates why synchronization is difficult to perform correctly through variables. The value of flag is undefined in both prints on thread 1 and the value of data is only well-defined in the second print.
Example mem_model.2.c

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

int main()
{
    int data;
    int flag=0;
    #pragma omp parallel num_threads(2)
    {
        if (omp_get_thread_num()==0)
        {
            /* Write to the data buffer that will be
               read by thread */
            data = 42;
            /* Flush data to thread 1 and strictly order
               the write to data
               relative to the write to the flag */
            #pragma omp flush(flag, data)
            /* Set flag to release thread 1 */
            flag = 1;
            /* Flush flag to ensure that thread 1 sees
               the change */
            #pragma omp flush(flag)
        }
        else if(omp_get_thread_num()==1)
        {
            /* Loop until we see the update to the flag */
            #pragma omp flush(flag, data)
            while (flag < 1)
            {
                #pragma omp flush(flag, data)
            }
            /* Values of flag and data are undefined */
            printf("flag=%d data=%d\n", flag, data);
            #pragma omp flush(flag, data)
            /* Values data will be 42, value of flag
               still undefined */
            printf("flag=%d data=%d\n", flag, data);
        }
    }
    return 0;
}
```
Example mem_model_2.f

```fortran
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
```

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

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

int main()
{
    int flag=0;

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

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

    return 0;
}
```
**Example mem_model.3.f**

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

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

**CHAPTER 8. MEMORY MODEL**

255
8.2 Race Conditions Caused by Implied Copies of Shared Variables in Fortran

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

Example fort_race.1.f90

```fortran
SUBROUTINE SHARED_RACE
    INCLUDE "omp_lib.h" ! or USE OMP_LIB
    REAL A(20)
    INTEGER MYTHREAD
    !$OMP PARALLEL SHARED(A) PRIVATE(MYTHREAD)
    MYTHREAD = OMP_GET_THREAD_NUM()
    IF (MYTHREAD .EQ. 0) THEN
        CALL SUB(A(1:10)) ! compiler may introduce writes to A(6:10)
    ELSE
        A(6:10) = 12
    ENDIF
    !$OMP END PARALLEL
END SUBROUTINE SHARED_RACE

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

Fortran
CHAPTER 9

Program Control

Some specific and elementary concepts of controlling program execution are illustrated in the examples of this chapter. Control can be directly managed with conditional control code (ifdef’s with the _OPENMP macro, and the Fortran sentinel (!$) for conditionally compiling). The if clause on some constructs can direct the runtime to ignore or alter the behavior of the construct. Of course, the base-language if statements can be used to control the "execution" of stand-alone directives (such as flush, barrier, taskwait, and taskyield). However, the directives must appear in a block structure, and not as a substatement as shown in examples 1 and 2 of this chapter.

CANCELLATION

Cancellation (termination) of the normal sequence of execution for the threads in an OpenMP region can be accomplished with the cancel construct. The construct uses a construct-type-clause to set the region-type to activate for the cancellation. That is, inclusion of one of the construct-type-clause names parallel, for, do, sections or taskgroup on the directive line activates the corresponding region. The cancel construct is activated by the first encountering thread, and it continues execution at the end of the named region. The cancel construct is also a cancellation point for any other thread of the team to also continue execution at the end of the named region.

Also, once the specified region has been activated for cancellation any thread that encounters a cancellation point construct with the same named region (construct-type-clause), continues execution at the end of the region.

For an activated cancel taskgroup construct, the tasks that belong to the taskgroup set of the innermost enclosing taskgroup region will be canceled.

A task that encounters the cancel taskgroup construct continues execution at the end of its task region. Any task of the taskgroup that has already begun execution will run to completion, unless it encounters a cancellation point; tasks that have not begun execution "may" be discarded as completed tasks.
CONTROL VARIABLES

Internal control variables (ICV) are used by implementations to hold values which control the
evaluation of OpenMP regions. Control (and hence the ICVs) may be set as implementation
defaults, or set and adjusted through environment variables, clauses, and API functions. Many of
the ICV control values are accessible through API function calls. Also, initial ICV values are
reported by the runtime if the OMP_DISPLAY_ENV environment variable has been set to TRUE.

NESTED CONSTRUCTS

Certain combinations of nested constructs are permitted, giving rise to a combined construct
consisting of two or more constructs. These can be used when the two (or several) constructs would
be used immediately in succession (closely nested). A combined construct can use the clauses of
the component constructs without restrictions. A composite construct is a combined construct
which has one or more clauses with (an often obviously) modified or restricted meaning, relative to
when the constructs are uncombined.

Certain nestings are forbidden, and often the reasoning is obvious. Worksharing constructs cannot
be nested, and the barrier construct cannot be nested inside a worksharing construct, or a
critical construct. Also, target constructs cannot be nested.

The parallel construct can be nested, as well as the task construct. The parallel execution in
the nested parallel construct(s) is control by the OMP_NESTED and
OMP_MAX_ACTIVE_LEVELS environment variables, and the omp_set_nested() and
omp_set_max_active_levels() functions.

More details on nesting can be found in the Nesting of Regions of the Directives chapter in the
OpenMP Specifications document.
9.1 Conditional Compilation

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

```
#include <stdio.h>

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

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

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

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

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

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

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

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

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

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

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

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

\begin{Verbatim}
Example icv.1.c
\end{Verbatim}
```c
#include <stdio.h>
#include <omp.h>

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

```fortran
program icv
use omp_lib

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

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

OpenMP Examples Version 4.5.0 - November 2016
9.3 Placement of *flush, barrier, taskwait* and *taskyield* Directives

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

---

```c
void standalone_wrong()
{
    int a = 1;
    if (a != 0)
        #pragma omp flush(a)
        /* incorrect as flush cannot be immediate substatement of if statement */
    if (a != 0)
        #pragma omp barrier
        /* incorrect as barrier cannot be immediate substatement of if statement */
    if (a != 0)
        #pragma omp taskyield
        /* incorrect as taskyield cannot be immediate substatement of if statement */
    if (a != 0)
        #pragma omp taskwait
        /* incorrect as taskwait cannot be immediate substatement of if statement */
}
```

---

The following example is non-conforming, because the *flush, barrier, taskwait*, and *taskyield* directives are stand-alone directives and cannot be the action statement of an *if* statement or a labeled branch target.
Example standalone.1.f90

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

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

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

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

  GOTO 100

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

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

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

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

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

---

Example standalone.2.c

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

    #pragma omp parallel
    {
        if (a != 0) {
            #pragma omp flush(a)
        }

        if (a != 0) {
            #pragma omp barrier
        }

        if (a != 0) {
            #pragma omp taskwait
        }

        if (a != 0) {
            #pragma omp taskyield
        }
    }
}
```

---

The following example is conforming because the `flush`, `barrier`, `taskwait`, and `taskyield` directives are enclosed in an `if` construct or follow the labeled branch target.
Example standalone.f90

SUBROUTINE STANDALONE_OK()

  INTEGER A
  A = 1
  IF (A .NE. 0) THEN
    !$OMP FLUSH(A)
  ENDIF

  IF (A .NE. 0) THEN
    !$OMP BARRIER
  ENDIF

  IF (A .NE. 0) THEN
    !$OMP TASKWAIT
  ENDIF

  IF (A .NE. 0) THEN
    !$OMP TASKYIELD
  ENDIF

  GOTO 100

100 CONTINUE

  !$OMP FLUSH(A)
  GOTO 200

200 CONTINUE

  !$OMP BARRIER
  GOTO 300

300 CONTINUE

  !$OMP TASKWAIT
  GOTO 400

400 CONTINUE

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

```c++
#include <iostream>
#include <exception>
#include <cstddef>

#define N 10000
extern void causes_an_exception();
extern void phase_1();
extern void phase_2();

void example() {
    std::exception *ex = NULL;
    #pragma omp parallel shared(ex)
    {
        #pragma omp for
        for (int i = 0; i < N; i++) {
            // no 'if' that prevents compiler optimizations
            try {
                causes_an_exception();
            }
            catch (std::exception *e) {
                // still must remember exception for later handling
                #pragma omp atomic write
                ex = e;
                // cancel worksharing construct
            }
        }
        // if an exception has been raised, cancel parallel region
        if (ex) {
            #pragma omp cancel parallel
        }
    }
    phase_1();
}
```
The following example illustrates the use of the `cancel` construct in error handling. If there is an error condition from the `allocate` statement, the cancellation is activated. The encountering thread sets the shared variable `err` and other threads of the binding thread set proceed to the end of the worksharing construct after the cancellation has been activated.

```fortran
subroutine example(n, dim)
    integer, intent(in) :: n, dim(n)
    integer :: i, s, err
    real, allocatable :: B(:)
    err = 0
    !$omp parallel shared(err)
    ! ...
    !$omp do private(s, B)
    do i=1, n
        !$omp cancellation point do
        allocate(B(dim(i)), stat=s)
        if (s .gt. 0) then
            !$omp atomic write
            err = s
        !$omp cancel do
        endif
        ! ...
        !$omp deallocate private array B
        if (allocated(B)) then
            deallocate(B)
        endif
    enddo
    !$omp end parallel
end subroutine
```
The following example shows how to cancel a parallel search on a binary tree as soon as the search value has been detected. The code creates a task to descend into the child nodes of the current tree node. If the search value has been found, the code remembers the tree node with the found value through an atomic write to the result variable and then cancels execution of all search tasks. The function search_tree_parallel groups all search tasks into a single task group to control the effect of the cancel taskgroup directive. The level argument is used to create undeferred tasks after the first ten levels of the tree.

Example cancellation.2.c

```c
#include <stddef.h>

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

binary_tree_t *search_tree(binary_tree_t *tree, int value, int level) {
    binary_tree_t *found = NULL;
    if (tree) {
        if (tree->value == value) {
            found = tree;
        } else {
            #pragma omp task shared(found) if(level < 10)
            {  
                binary_tree_t *found_left = NULL;
                found_left = search_tree(tree->left, value, level + 1);
                if (found_left) {
                    #pragma omp atomic write
                    found = found_left;
                    #pragma omp cancel taskgroup
                }
                #pragma omp task shared(found) if(level < 10)
                {  
                    binary_tree_t *found_right = NULL;
                    found_right = search_tree(tree->right, value, level + 1);
                    if (found_right) {
                        #pragma omp atomic write
                        found = found_right;
                        #pragma omp cancel taskgroup
                    }
                }
            }
        }
    }
    return found;
}
```
The following is the equivalent parallel search example in Fortran.

```fortran
module parallel_search

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

contains

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

    if (associated(tree)) then
      if (tree%value .eq. value) then
        found => tree
      else
        !$omp task shared(found) if(level<10)
        call search_tree(tree%left, value, level+1, found_left)
        if (associated(found_left)) then
          found => found_left
        else
          if (associated(found_right)) then
            found => found_right
          else
            found => NULL()
          endif
        endif
      endif
    endif
  end subroutine search_tree

end module parallel_search
```

Example cancellation.

```c
#include <omp.h>

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

The following is the equivalent parallel search example in Fortran.

```fortran
module parallel_search

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

contains

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

    if (associated(tree)) then
      if (tree%value .eq. value) then
        found => tree
      else
        !$omp task shared(found) if(level<10)
        call search_tree(tree%left, value, level+1, found_left)
        if (associated(found_left)) then
          found => found_left
        else
          if (associated(found_right)) then
            found => found_right
          else
            found => NULL()
          endif
        endif
      endif
    endif
  end subroutine search_tree

end module parallel_search
```
module parallel_search

!$omp critical
    found => found_left
!$omp end critical

!$omp cancel taskgroup
endif
!$omp end task

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

!$omp taskwait
endif
end subroutine

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

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

end module parallel_search
9.5 Nested Loop Constructs

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

C / C++

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

Fortran

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

SUBROUTINE GOOD_NESTING(N)
INTEGER N

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

The following variation of the preceding example is also conforming:

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

```c
void good_nesting2(int n)
{
    int i;
    #pragma omp parallel default(shared)
    {
        #pragma omp for
        for (i=0; i<n; i++)
            work1(i, n);
    }
}
```
Example nested_loop.2.f

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
The examples in this section illustrate the region nesting rules.

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

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

```
SUBROUTINE WORK(I, J)
INTEGER I, J
END SUBROUTINE WORK
SUBROUTINE WRONG1(N)
INTEGER N
INTEGER I,J
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
   DO I = 1, N
      !$OMP DO ! incorrect nesting of loop regions
         DO J = 1, N
                  CALL WORK(I,J)
      END DO
   END DO
!$OMP END PARALLEL
END SUBROUTINE WRONG1
```
The following orphaned version of the preceding example is also non-conforming:

```
C / C++
Example nesting_restrict.2.c

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

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

```
Fortran
Example nesting_restrict.2.f

SUBROUTINE WORK1(I,N)
INTEGER I, N
INTEGER J
!$OMP DO ! incorrect nesting of loop regions
DO J = 1, N
  CALL WORK(I,J)
END DO
END SUBROUTINE WORK1
SUBROUTINE WRONG2(N)
INTEGER N
INTEGER I
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
  DO I = 1, N
    CALL WORK1(I,N)
END DO
```

OpenMP Examples Version 4.5.0 - November 2016
The following example is non-conforming because the loop and `single` regions are closely nested:

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

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

```
SUBROUTINE WRONG3(N)
    INTEGER N
    INTEGER I
    !$OMP PARALLEL DEFAULT(SHARED)
    !$OMP DO
    DO I = 1, N
    !$OMP SINGLE ! incorrect nesting of regions
    CALL WORK(I, 1)
    END SINGLE
    !$OMP END PARALLEL
END SUBROUTINE WRONG3
```
Example nesting_restrict.4.c

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

Example nesting_restrict.4.f

```fortran
SUBROUTINE WRONG4(N)  
  INTEGER N  
  INTEGER I  
  !$OMP PARALLEL DEFAULT(SHARED)  
  !$OMP DO  
    DO I = 1, N  
      CALL WORK(I, 1)  
    END DO  
  END PARALLEL
END SUBROUTINE WRONG4
```

The following example is non-conforming because the **barrier** region cannot be closely nested inside the **critical** region. If this were permitted, it would result in deadlock due to the fact that only one thread at a time can enter the **critical** region:
The following example is non-conforming because the barrier region cannot be closely nested inside the single region. If this were permitted, it would result in deadlock due to the fact that only one thread executes the single region:
Example nesting_restrict.6.c

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

Example nesting_restrict.6.f

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

Document Revision History

A.1 Changes from 4.0.2 to 4.5.0

- Reorganized into chapters of major topics
- Included file extensions in example labels to indicate source type
- Applied the explicit `map(tofrom)` for scalar variables in a number of examples to comply with the change of the default behavior for scalar variables from `map(tofrom)` to `firstprivate` in the 4.5 specification
- Added the following new examples:
  - `linear` clause in loop constructs (Section 1.8 on page 22)
  - task priority (Section 3.2 on page 71)
  - `taskloop` construct (Section 3.6 on page 85)
  - `directive-name` modifier in multiple `if` clauses on a combined construct (Section 4.1.5 on page 93)
  - unstructured data mapping (Section 4.3 on page 108)
  - `link` clause for `declare target` directive (Section 4.5.5 on page 123)
  - asynchronous target execution with `nowait` clause (Section 4.7 on page 135)
  - device memory routines and device pointers (Section 4.9.4 on page 152)
  - doacross loop nest (Section 6.8 on page 194)
  - locks with hints (Section 6.9 on page 200)
  - C/C++ array reduction (Section 7.9 on page 233)
  - C++ reference types in data sharing clauses (Section 7.12 on page 246)
### A.2 Changes from 4.0.1 to 4.0.2

- Names of examples were changed from numbers to mnemonics
- Added SIMD examples (Section 5.1 on page 155)
- Applied miscellaneous fixes in several source codes
- Added the revision history

### A.3 Changes from 4.0 to 4.0.1

Added the following new examples:

- the `proc_bind` clause (Section 2.1 on page 42)
- the `taskgroup` construct (Section 3.4 on page 80)

### A.4 Changes from 3.1 to 4.0

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

Version 4.0 added the following new examples:

- task dependences (Section 3.3 on page 73)
- `target` construct (Section 4.1 on page 88)
- `target data` construct (Section 4.2 on page 96)
- `target update` construct (Section 4.4 on page 111)
- `declare target` construct (Section 4.5 on page 115)
- `teams` constructs (Section 4.6 on page 126)
- asynchronous execution of a `target` region using tasks (Section 4.7.1 on page 135)
- array sections in device constructs (Section 4.8 on page 144)
- device runtime routines (Section 4.9 on page 148)
1. Fortran ASSOCIATE construct (Section 7.13 on page 247)
2. cancellation constructs (Section 9.4 on page 267)