OpenMP Application Programming Interface

Examples

Version 6.0 – November 2024

Source codes for OpenMP Examples 6.0 are available at github (https://github.com/OpenMP/Examples/tree/v6.0).

Copyright © 1997-2024 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.

Foreword

The OpenMP Examples document has been updated with new features found in the OpenMP 6.0 Specification. In order to provide users with new feature examples concurrently with the release of the OpenMP 6.0 Specification, the 6.0 Examples document is being released early with a caveat that some of the 6.0 features (such as **`workdistribute`** construct, **`taskgraph`** construct, **`threadset`** clause and free-agent threads) will be covered in the next release of the document. For a list of the new examples and updates in this release, please refer to the Document Revision History of the Appendix on page 603.

Text describing an example with a 6.0 feature specifically states that the feature support begins in the OpenMP 6.0 Specification. Also, an **`omp_6.0`** keyword is included in the metadata of the source code. These distinctions are presented to remind readers that a 6.0 compliant OpenMP implementation is necessary to use these features in codes.

Incremental releases will become available as more feature examples and updates are submitted and approved by the OpenMP Examples Subcommittee. Examples are accepted for this document after discussions, revisions and reviews in the Examples Subcommittee, and two reviews/discussions and two votes in the OpenMP Language Committee. Draft examples are often derived from case studies for new features in the language, and are revised to illustrate the basic application of the features with code comments, and a text description. We are grateful to the numerous members of the Language Committee who took the time to prepare codes and descriptions, and shepherd them through the acceptance process. We sincerely appreciate the Example Subcommittee members, who actively participated and contributed in weekly meetings over the years.

Examples Subcommittee Co-chairs:

Henry Jin (NASA Ames Research Center)
Swaroop Pophale (Oak Ridge National Laboratory)

Past Examples Subcommittee Co-chairs:

- Kent Milfeld (2014 - 2022)

Contents

1 Introduction **1**

1.1 Examples Organization . . . 2

2 OpenMP Directive Syntax **3**

2.1 C/C++ Pragmas . . . 4

2.2 C/C++ Attributes . . . 5

2.3 Fortran Comments (Fixed Source Form) . . . 9

2.4 Fortran Comments (Free Source Form) . . . 10

3 Parallel Execution **15**

3.1 A Simple Parallel Loop . . . 17

3.2 **`parallel`** Construct . . . 18

3.3 **`teams`** Construct on Host . . . 20

3.4 Controlling the Number of Threads on Multiple Nesting Levels . . . 23

3.5 Interaction Between the **`num_threads`** Clause and **`omp_set_dynamic`** . . . 25

3.6 Fortran Restrictions on the **`do`** Construct . . . 27

3.7 **`nowait`** Clause . . . 28

3.8 **`collapse`** Clause . . . 31

3.9 **`linear`** Clause in Loop Constructs . . . 37

3.10 **`parallel sections`** Construct . . . 39

3.11 **`firstprivate`** Clause and **`sections`** Construct . . . 40

3.12 **`single`** Construct . . . 41

3.13 **`workshare`** Construct . . . 43

3.14 **`masked`** Construct . . . 47

3.15 **`loop`** Construct . . . 49

3.16 Parallel Random Access Iterator Loop . . . 53

3.17 **`omp_set_dynamic`** and **`omp_set_num_threads`** Routines . . . 53

3.18 **`omp_get_num_threads`** Routine . . . 56

4 OpenMP Affinity **59**

4.1 **proc_bind** Clause . . . 60
4.1.1 Spread Affinity Policy . . . 60
4.1.2 Close Affinity Policy . . . 63
4.1.3 Primary Affinity Policy . . . 65
4.2 Task Affinity . . . 66
4.3 Affinity Display . . . 67
4.4 Affinity Query Functions . . . 79

5 Tasking **83**
5.1 **task** and **taskwait** Constructs . . . 84
5.2 Task Priority . . . 102
5.3 Task Dependences . . . 103
5.3.1 Flow Dependence . . . 103
5.3.2 Anti-dependence . . . 105
5.3.3 Output Dependence . . . 106
5.3.4 Concurrent Execution with Dependences . . . 107
5.3.5 Matrix multiplication . . . 110
5.3.6 **taskwait** with Dependences . . . 112
5.3.7 Mutually Exclusive Execution with Dependences . . . 118
5.3.8 Multidependences Using Iterators . . . 121
5.3.9 Dependence for Undeferred Tasks . . . 123
5.3.10 Transparent Task Dependences . . . 126
5.4 Task Detachment . . . 130
5.5 **taskgroup** Construct . . . 134
5.6 **taskyield** Construct . . . 137
5.7 **taskloop** Construct . . . 138
5.8 Combined **parallel masked** and **taskloop** Constructs . . . 141
5.9 Task Dependences for **taskloop** Construct . . . 144

6 Devices **149**
6.1 **target** Construct . . . 150
6.1.1 **target** Construct on **parallel** Construct . . . 150
6.1.2 **target** Construct with **map** Clause . . . 151
6.1.3 **map** Clause with **to**/**from** map-types . . . 152

6.1.4 **map** Clause with Array Sections . . . 153
6.1.5 **target** Construct with **if** Clause . . . 155
6.1.6 Target Reverse Offload . . . 158
6.2 **defaultmap** Clause . . . 160
6.3 Pointer Mapping . . . 166
6.4 Structure Mapping . . . 175
6.5 Fortran Allocatable Array Mapping . . . 181
6.6 Array Sections in Device Constructs . . . 185
6.7 Unified Shared Memory . . . 189
6.8 C++ Virtual Functions . . . 191
6.9 Array Shaping . . . 194
6.10 **declare mapper** Directive . . . 197
6.11 **target data** Construct . . . 203
6.11.1 Simple **target data** Construct . . . 203
6.11.2 **target data** Region Enclosing Multiple **target** Regions . . . 204
6.11.3 **target data** Construct with Orphaned Call . . . 208
6.11.4 **target data** Construct with **if** Clause . . . 211
6.12 **target enter data** and **target exit data** Constructs . . . 215
6.13 **target update** Construct . . . 217
6.13.1 Simple **target data** and **target update** Constructs . . . 217
6.13.2 **target update** Construct with **if** Clause . . . 219
6.13.3 **target update** Construct with Mapper . . . 221
6.14 Declare Target Directive . . . 223
6.14.1 Declare Target Directive for a Procedure . . . 223
6.14.2 Declare Target Directive for Indirect Procedure Call . . . 225
6.14.3 Declare Target Directive for Class Type . . . 227
6.14.4 Declare Target Directive for Variables . . . 231
6.14.5 Declare Target Directive with **declare simd** . . . 234
6.14.6 Declare Target Directive with **link** Clause . . . 237
6.14.7 Declare Target Directive with **device_type** Clause . . . 239
6.15 Lambda Expressions . . . 241
6.16 **teams** Construct and Related Combined Constructs . . . 243

6.16.1 **target** and **teams** Constructs with **omp_get_num_teams** and **omp_get_team_num** Routines . . . 243
6.16.2 **target**, **teams**, and **distribute** Constructs . . . 245
6.16.3 **target teams**, and Distribute Parallel Loop Constructs . . . 247
6.16.4 **target teams** and Distribute Parallel Loop Constructs with Scheduling Clauses . . . 249
6.16.5 **target teams** and **distribute simd** Constructs . . . 250
6.16.6 **target teams** and Distribute Parallel Loop SIMD Constructs . . . 252
6.16.7 Evaluation of **num_teams** Clause that Appears inside **target** Region . 253
6.17 Asynchronous **target** Execution and Dependences . . . 255
6.17.1 Asynchronous **target** with Tasks . . . 255
6.17.2 **nowait** Clause on **target** Construct . . . 259
6.17.3 Asynchronous **target** with **nowait** and **depend** Clauses . . . 261
6.17.4 Conditionally Asynchronous **target** Using the **nowait** Clause . . . 263
6.18 Device Routines . . . 265
6.18.1 **omp_is_initial_device** Routine . . . 265
6.18.2 **omp_get_num_devices** Routine . . . 267
6.18.3 **omp_set_default_device** and **omp_get_default_device** Routines . . . 268
6.18.4 Device and Host Memory Association . . . 269
6.18.5 Target Memory and Device Pointers Routines . . . 272
6.19 Traits for Specifying Devices . . . 282

7 SIMD **283**
7.1 **simd** and **declare simd** Directives . . . 283
7.2 **inbranch** and **notinbranch** Clauses . . . 290
7.3 Loop-Carried Lexical Forward Dependence . . . 294
7.4 **ref**, **val**, **uval** Modifiers for **linear** Clause . . . 296

8 Loop Transformations **305**
8.1 **tile** Construct . . . 305
8.2 Incomplete Tiles . . . 309
8.3 **unroll** Construct . . . 317
8.4 **apply** Clause . . . 327

8.4.1 Syntax and Effect . . . 327
8.4.2 Spanning Loop Associations . . . 333
8.4.3 Nested apply . . . 337

9 Synchronization **341**

9.1 **critical** Construct . . . 342
9.2 Worksharing Constructs Inside a **critical** Construct . . . 345
9.3 Binding of **barrier** Regions . . . 346
9.4 **atomic** Construct . . . 349
9.5 Atomic Compare . . . 354
9.6 Restrictions on the **atomic** Construct . . . 357
9.7 Atomic Hint . . . 360
9.8 Synchronization Based on Acquire/Release Semantics . . . 361
9.9 **ordered** Clause and **ordered** Construct . . . 369
9.10 **depobj** Construct . . . 372
9.11 Doacross Loop Nest . . . 376
9.12 Lock Routines . . . 382
9.12.1 **omp_init_lock** Routine . . . 382
9.12.2 **omp_init_lock_with_hint** Routine . . . 383
9.12.3 Ownership of Locks . . . 384
9.12.4 Simple Lock Routines . . . 386
9.12.5 Nestable Lock Routines . . . 388

10 Data Environment **391**

10.1 **threadprivate** Directive . . . 392
10.2 **default(none)** Clause . . . 398
10.3 **private** Clause . . . 399
10.4 Fortran Private Loop Iteration Variables . . . 403
10.5 Fortran Restrictions on **shared** and **private** Clauses with Common Blocks . . . 404
10.6 Fortran Restrictions on Storage Association with the **private** Clause . . . 406
10.7 Passing Shared Variable to Procedure in Fortran . . . 409
10.8 C/C++ Arrays in a **firstprivate** Clause . . . 411
10.9 **lastprivate** Clause . . . 412
10.10 Reduction . . . 415

10.10.1 **reduction** Clause . . . 415
10.10.2 Task Reduction . . . 422
10.10.3 Reduction on Combined Target Constructs . . . 427
10.10.4 Task Reduction with Target Constructs . . . 431
10.10.5 Taskloop Reduction . . . 436
10.10.6 Reduction with the **scope** Construct . . . 443
10.10.7 Reduction on Private Variables in a **parallel** Region . . . 445
10.10.8 User-Defined Reduction . . . 450
10.11 Induction . . . 460
10.11.1 **induction** Clause . . . 460
10.11.2 User-defined Induction . . . 463
10.12 **scan** Directive . . . 465
10.13 **copyin** Clause . . . 470
10.14 **copyprivate** Clause . . . 472
10.15 C++ Reference in Data-Sharing Clauses . . . 476
10.16 Fortran **ASSOCIATE** Construct . . . 477

11 Memory Model **481**
11.1 OpenMP Memory Model . . . 482
11.2 Memory Allocators . . . 492
11.3 Race Conditions Caused by Implied Copies of Shared Variables in Fortran . . . 508

12 Program Control **509**
12.1 Assumption Directives . . . 510
12.2 Conditional Compilation . . . 515
12.3 Internal Control Variables (ICVs) . . . 516
12.3.1 **num_threads** Clause with a List . . . 519
12.4 Placement of **flush**, **barrier**, **taskwait** and **taskyield** Directives . . . 523
12.5 Cancellation Constructs . . . 526
12.6 **requires** Directive . . . 531
12.7 Context-based Variant Selection . . . 533
12.7.1 **declare variant** Directive . . . 534
12.7.2 Metadirectives . . . 541
12.7.3 Context Selector Scoring . . . 552

12.8 **dispatch** Construct . . . 560
12.9 Nested Loop Constructs . . . 563
12.10 Restrictions on Nesting of Regions . . . 566
12.11 Target Offload . . . 572
12.12 **omp_pause_resource** and **omp_pause_resource_all** Routines . . . 576
12.13 Controlling Concurrency and Reproducibility with the **order** Clause . . . 579
12.14 **interop** Construct . . . 585
12.15 Utilities . . . 588
12.15.1 Timing Routines . . . 588
12.15.2 Environment Display . . . 589
12.15.3 **error** Directive . . . 591

13 OMPT Interface **595**
13.1 OMPT Start . . . 596

A Feature Deprecations and Updates in Examples **599**
A.1 Updated Examples for Different Versions . . . 600

B Document Revision History **603**
B.1 Changes from 5.2.2 to 6.0 . . . 603
B.2 Changes from 5.2.1 to 5.2.2 . . . 604
B.3 Changes from 5.2 to 5.2.1 . . . 605
B.4 Changes from 5.1 to 5.2 . . . 606
B.5 Changes from 5.0.1 to 5.1 . . . 607
B.6 Changes from 5.0.0 to 5.0.1 . . . 609
B.7 Changes from 4.5.0 to 5.0.0 . . . 609
B.8 Changes from 4.0.2 to 4.5.0 . . . 610
B.9 Changes from 4.0.1 to 4.0.2 . . . 611
B.10 Changes from 4.0 to 4.0.1 . . . 611
B.11 Changes from 3.1 to 4.0 . . . 612

Index **613**

List of Figures

4.1 A machine architecture with two quad-core processors 60

8.1 Tiling illustrations 309

List of Tables

A.1 Deprecated Features and Their Replacements 599
A.2 Updated Examples for Features Deprecated in Version 6.0 600
A.3 Updated Examples for Features Deprecated in Version 5.2 600
A.4 Updated Examples for Features Deprecated in Version 5.1 602
A.5 Updated Examples for Features Deprecated in Version 5.0 602

1 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.

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 documents and source codes for OpenMP Examples can be downloaded from https://github.com/OpenMP/Examples. Each directory holds the contents of a chapter and has a *sources* subdirectory of its codes. This OpenMP Examples 6.0 document and its codes are tagged as *v6.0*.

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

`https://www.openmp.org`

1.1 Examples Organization

This document includes examples of the OpenMP API directives, constructs, and routines.

Each example is labeled with *ename.seq-id.ext*, where *ename* is the example name, *seq-id* is the sequence identifier in a section, and *ext* is the source file extension to indicate the code type and source form. *ext* is one of the following:

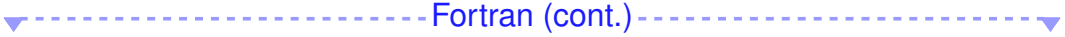
- *c* – C code,
- *cpp* – C++ code,
- *f* – Fortran code in fixed form, and
- *f90* – Fortran code in free form.

Example labels include version information of the form (**omp**_*verno*) to indicate features that are illustrated by an example for a specific OpenMP version, such as "*scan.1.c* (**omp_5.0**)." Some of the example labels include version information of the form (**pre_omp_3.0**) to indicate features that are specified prior to OpenMP version 3.0, such as "*ploop.1.c* (**pre_omp_3.0**)."

Language markers may be used to indicate text or codes that are specific to a particular base language.

This is C/C++ specific: A statement following a directive is compound only when necessary, and a non-compound statement is indented with respect to a directive preceding it.

This is Fortran specific...



This marks the continuation of language specific page.

Throughout the examples document we assume that the number of threads used for a **parallel** region is the same as the number of threads requested, unless explicitly specified otherwise.

2 OpenMP Directive Syntax

OpenMP *directives* use base-language mechanisms to specify OpenMP program behavior. In C/C++ code, the directives are formed with either pragmas or attributes. Fortran directives are formed with comments in free form and fixed form sources (codes). All of these mechanisms allow the compilation to ignore the OpenMP directives if OpenMP is not supported or enabled.

The OpenMP directive is a combination of the base-language mechanism and a *directive-specification*, as shown below. The *directive-specification* consists of the *directive-name* which may seldomly have arguments, followed by optional *clauses*. Full details of the syntax can be found in the OpenMP Specification. Illustrations of the syntax is given in the examples.

The formats for combining a base-language mechanism and a *directive-specification* are:

C/C++ pragmas

#pragma omp *directive−specification*

C/C++ attribute specifiers

[[omp :: directive(*directive−specification* **)]]**
[[omp :: decl(*directive−specification* **)]]**

C++ attribute specifiers

[[using omp : directive(*directive−specification* **)]]**
[[using omp : decl(*directive−specification* **)]]**

where the **decl** attribute may be used for declarative directives alternatively.

Fortran comments

!$omp *directive−specification*

where **c$omp** and ***$omp** may be used in Fortran fixed form sources.

Most OpenMP directives accept clauses that alter the semantics of the directive in some way, and some directives also accept parenthesized arguments that follow the directive name. A clause may just be a keyword (e.g., **untied**) or it may also accept argument lists (e.g., **shared(***x,y,z***)**) and/or optional modifiers (e.g., **tofrom** in **map(tofrom:** *x,y,z***)**). Clause modifiers may be "simple" or "complex" – a complex modifier consists of a keyword followed by one or more parameters, bracketed by parentheses, while a simple modifier does not. An example of a complex modifier is the **iterator** modifier, as in **map(iterator(***i=0:n***), tofrom:** *p[i]***)**, or the **step** modifier, as in **linear(***x***: ref, step(***4***))**. In the preceding examples, **tofrom** and **ref** are simple modifiers.

For Fortran, a declarative directive (such as **declare reduction**) must appear after any **USE**, **IMPORT**, and **IMPLICIT** statements in the specification part.

2.1 C/C++ Pragmas

OpenMP C and C++ directives can be specified with the C/C++ **#pragma** directive. An OpenMP directive begins with **#pragma omp** and is followed by the OpenMP directive name, and required and optional clauses. Lines are continued in the usual manner, and comments may be included at the end. Directives are case sensitive.

The example below illustrates the use of the OpenMP pragma form. The first pragma (PRAG 1) specifies a combined **parallel for** directive, with a **num_threads** clause, and a comment. The second pragma (PRAG 2) shows the same directive split across two lines. The next nested pragmas (PRAG 3 and 4) show the previous combined directive as two separate directives. The executable directives above all apply to the next statement. The **parallel** directive can be applied to a *structured block* as shown in PRAG 5.

Example directive_syntax_pragma.1.c (**pre_omp_3.0**)

```
S-1   #include   <omp.h>
S-2   #include <stdio.h>
S-3   #define NT 4
S-4   #define thrd_no omp_get_thread_num
S-5
S-6   int main(){
S-7       #pragma omp parallel for num_threads(NT)                  // PRAG 1
S-8       for(int i=0; i<NT; i++) printf("thrd no %d\n",thrd_no());
S-9
S-10      #pragma omp parallel for \
S-11                  num_threads(NT)                               // PRAG 2
S-12      for(int i=0; i<NT; i++) printf("thrd no %d\n",thrd_no());
S-13
S-14      #pragma omp parallel num_threads(NT)                      // PRAG 3-4
S-15      #pragma omp for
S-16      for(int i=0; i<NT; i++) printf("thrd no %d\n",thrd_no());
S-17
S-18      #pragma omp parallel num_threads(NT)                      // PRAG 5
S-19      {
S-20         int no = thrd_no();
S-21         if (no%2) { printf("thrd no %d is Odd \n",no);}
S-22         else      { printf("thrd no %d is Even\n",no);}
S-23
S-24         #pragma omp for
S-25         for(int i=0; i<NT; i++) printf("thrd no %d\n",thrd_no());
S-26      }
S-27  }
S-28  /*
S-29        repeated 4 times, any order
```

```
S-30          OUTPUT: thrd no 0
S-31          OUTPUT: thrd no 1
S-32          OUTPUT: thrd no 2
S-33          OUTPUT: thrd no 3
S-34
S-35          any order
S-36          OUTPUT: thrd no 0 is Even
S-37          OUTPUT: thrd no 2 is Even
S-38          OUTPUT: thrd no 1 is Odd
S-39          OUTPUT: thrd no 3 is Odd
S-40    */
```

C / C++

C / C++

2.2 C/C++ Attributes

OpenMP directives for C/C++ can also be specified with the **`directive`** extension for the C23 and C++11 standard *attributes*.

The example below shows two ways to parallelize a **`for`** loop using the **`#pragma`** syntax. The first pragma uses the combined **`parallel for`** directive, and the second applies the uncombined closely nested directives, **`parallel`** and **`for`**, directly to the same statement. These are labeled PRAG 1-3.

Using the attribute syntax, the same construct in PRAG 1 is applied in two different ways in attribute form, as shown in the ATTR 1 and ATTR 2 sections. In ATTR 1 the attribute syntax is used with the **`omp ::`** namespace form. In ATTR 2 the attribute syntax is used with the **`using omp :`** namespace form available for C++ only.

Next, parallelization is attempted by applying directives using two different syntaxes. For ATTR 3 and PRAG 4, the loop parallelization will fail to compile because multiple directives that apply to the same statement must all use either the attribute syntax or the pragma syntax. The lines have been commented out and labeled INVALID.

While multiple attributes may be applied to the same statement, compilation may fail if the ordering of the directive matters. For the ATTR 4-5 loop parallelization, the **`parallel`** directive precedes the **`for`** directive, but the compiler may reorder consecutive attributes. If the directives are reversed, compilation will fail.

The attribute directive of the ATTR 6 section resolves the previous problem (in ATTR 4-5). Here, the **`sequence`** attribute is used to apply ordering to the directives of ATTR 4-5, using the **`omp ::`** namespace qualifier. (The **`using omp :`** namespace form is not available for the **`sequence`** attribute.) Note, for the **`sequence`** attribute a comma must separate the **`directive`** extensions.

The last 3 pairs of sections (PRAG DECL 1-2, 3-4, and 5-6) show cases where directive ordering does not matter for **declare simd** directives.

In section PRAG DECL 1-2, the two loops use different SIMD forms of the *P* function (one with **simdlen(***4***)** and the other with **simdlen(***8***)**), as prescribed by the two different **declare simd** directives applied to the *P* function definitions (at the beginning of the code). The directives use the pragma syntax, and order is not important. For the next set of loops (PRAG DECL 3-4) that use the *Q* function, the attribute syntax is used for the **declare simd** directives. The result is compliant code since directive order is irrelevant. Sections ATTR DECL 5-6 are included for completeness. Here, the attribute form of the **simd** directive is used for loops calling the *Q* function, in combination with the attribute form of the **declare simd** directives declaring the variants for *Q*.

Example directive_syntax_attribute.1.cpp (**omp_6.0**)

```
#include <stdio.h>
#include <omp.h>
#define NT 4
#define thrd_no omp_get_thread_num

#pragma omp declare simd linear(i) simdlen(4)
#pragma omp declare simd linear(i) simdlen(8)
double P(int i){ return (double)i * (double)i; }

[[omp::directive(declare simd linear(i) simdlen(4))]]
[[omp::directive(declare simd linear(i) simdlen(8))]]
double Q(int i){ return (double)i * (double)i; }

int main() {

    #pragma omp parallel for num_threads(NT)                  // PRAG 1
    for(int i=0; i<NT; i++) printf("thrd no %d\n",thrd_no());

    #pragma omp parallel num_threads(NT)                      // PRAG 2
    #pragma omp for                                           // PRAG 3
    for(int i=0; i<NT; i++) printf("thrd no %d\n",thrd_no());

                                                              // ATTR 1
    [[omp::directive( parallel for num_threads(NT))]]
    for(int i=0; i<NT; i++) printf("thrd no %d\n",thrd_no());

                                                              // ATTR 2
    [[using omp : directive( parallel for num_threads(NT))]]
    for(int i=0; i<NT; i++) printf("thrd no %d\n",thrd_no());

```

```
S-31     // INVALID-- attribute and non-attribute on same statement
S-32     // [[ omp :: directive( parallel num_threads(NT) ) ]]          ATTR 3
S-33     // #pragma omp for                                             PRAG 4
S-34     // for(int i=0; i<NT; i++) printf("thrd no %d\n",thrd_no());
S-35
S-36
S-37     // INVALID-- directive order not guaranteed
S-38     // [[ omp :: directive( parallel num_threads(NT) ) ]]          ATTR 4
S-39     // [[ omp :: directive( for                       ) ]]          ATTR 5
S-40     // for(int i=0; i<NT; i++) printf("thrd no %d\n",thrd_no());
S-41
S-42                                                                    // ATTR 6
S-43        [[omp::sequence(directive(parallel num_threads(NT)),directive(for))]]
S-44        for(int i=0; i<NT; i++) printf("thrd no %d\n",thrd_no());
S-45
S-46        double tmp=0.0f;
S-47        #pragma omp simd reduction(+:tmp) simdlen(4)
S-48        for(int i=0;i<100;i++) tmp += P(i);                 // PRAG DECL 1
S-49        #pragma omp simd reduction(+:tmp) simdlen(8)
S-50        for(int i=0;i<100;i++) tmp += P(i);                 // PRAG DECL 2
S-51        printf("%f\n",tmp);
S-52
S-53        tmp=0.0f;
S-54        #pragma omp simd reduction(+:tmp) simdlen(4)
S-55        for(int i=0;i<100;i++) tmp += Q(i);                 // ATTR DECL 3
S-56        #pragma omp simd reduction(+:tmp) simdlen(8)
S-57        for(int i=0;i<100;i++) tmp += Q(i);                 // ATTR DECL 4
S-58        printf("%f\n",tmp);
S-59
S-60        tmp=0.0f;
S-61        [[ omp :: directive(simd reduction(+:tmp) simdlen(4))]]
S-62        for(int i=0;i<100;i++) tmp += Q(i);                 // ATTR DECL 5
S-63        [[ omp :: directive(simd reduction(+:tmp) simdlen(8))]]
S-64        for(int i=0;i<100;i++) tmp += Q(i);                 // ATTR DECL 6
S-65        printf("%f\n",tmp);
S-66     }
S-67     //    repeated 5 times, any order:
S-68     //    OUTPUT: thrd no  0
S-69     //    OUTPUT: thrd no  1
S-70     //    OUTPUT: thrd no  2
S-71     //    OUTPUT: thrd no  3
S-72
S-73     //    repeated 3 times:
S-74     //    OUTPUT: 656700.000000
```

The following code snippets show how to use the **omp::decl** attribute as an alternative way for specifying declarative directives. The **omp::decl** attribute can be embedded in the base language declarations as shown for variables in Cases 1 and 2, for function in Case 3, and for C++ template in Case 4. The variable and function name lists are implied from where the attributes are specified.

In Case 1, the prefix attribute applies to all variables (*u* and *v*) in the declaration; in Case 2, the postfix attribute applies to the associated variable (*a* as the directive argument for the **declare_target** directive, and *b* as the clause argument for the **link** clause on **declare_target**); in Case 3, the prefix attribute applies to the function (*f*). The comma to separate directive name (**declare_target**) and clause name (**link**) in the **omp::decl** attribute specifier in Case 2 is optional.

Case 4 shows the use of **omp::decl(declare_target)** for a C++ template function definition and its equivalent using the delimited **begin/end declare_target** pragma form.

Example directive_syntax_attribute.2.cpp (**omp_6.0**)

```
// Case 1
[[ omp::decl(threadprivate) ]] int u, v;
// equivalent to
int u ,v;
#pragma omp threadprivate(u, v)

// Case 2
int a[100] [[ omp::decl(declare_target) ]],
    b[100] [[ omp::decl(declare_target, link) ]];
// equivalent to
int a[100], b[100];
#pragma omp declare_target(a)
#pragma omp declare_target link(b)

// Case 3
[[ omp::decl(declare_target) ]] void f( int c );
// equivalent to
void f( int c );
#pragma omp declare_target(f)

// Case 4
template<typename T>
[[ omp::decl(declare_target) ]]
void foo(T);
// equivalent to
#pragma omp begin declare_target
template<typename T>
void foo(T);
#pragma omp end declare_target
```

C / C++

Fortran

2.3 Fortran Comments (Fixed Source Form)

OpenMP directives in Fortran codes with fixed source form are specified as comments with one of the **!$omp**, **c$omp**, and ***$omp** sentinels, followed by a directive name, and required and optional clauses. The sentinel must begin in column 1.

In the example below the first directive (DIR 1) specifies the **parallel do** combined directive, with a **num_threads** clause, and a comment. The second directive (DIR 2) shows the same directive split across two lines. The next nested directives (DIR 3 and 4) show the previous combined directive as two separate directives. Here, an **end** directive (**end parallel**) must be specified to demarcate the range (region) of the **parallel** directive.

Example directive_syntax_F_fixed_comment.1.f (**pre_omp_3.0**)

```
S-1       program main
S-2       include 'omp_lib.h'
S-3       integer NT
S-4
S-5       NT =4
S-6
S-7   c     sentinel c$omp or *$omp can also be used
S-8
S-9   c$omp parallel do num_threads(NT) !comments allowed here   DIR 1
S-10      do i = 1,NT
S-11        write(*,'("thrd no", i2)') omp_get_thread_num()
S-12      end do
S-13
S-14  !$omp parallel do
S-15  !$omp+ num_threads(NT)          !cont. w. char in col. 6   DIR 2
S-16      do i = 1,NT
S-17        write(*,'("thrd no", i2)') omp_get_thread_num()
S-18      end do
S-19
S-20  *$omp parallel num_threads(NT)  !multi-directive form      DIR 3
S-21  *$omp do                        !                          DIR 4
S-22      do i = 1,NT
S-23         write(*,'("thrd no", i2)') omp_get_thread_num()
S-24      end do
S-25  *$omp end parallel
S-26      end
S-27  !     repeated 3 times, any order
S-28  !     OUTPUT: thrd no  0
S-29  !     OUTPUT: thrd no  1
S-30  !     OUTPUT: thrd no  2
S-31  !     OUTPUT: thrd no  3
```

Fortran

2.4 Fortran Comments (Free Source Form)

OpenMP directives in Fortran codes with free source form are specified as comments that use the **!$omp** sentinel, followed by the directive name, and required and optional clauses. Lines are continued with an ending ampersand (**&**), and the continued line begins with **!$omp** or **!$omp&**. Comments may appear on the same line as the directive. Directives are case insensitive.

In the example below the first directive (DIR 1) specifies the **parallel do** combined directive, with a **num_threads** clause, and a comment. The second directive (DIR 2) shows the same directive split across two lines. The next nested directives (DIR 3 and 4) show the previous combined directive as two separate directives. Here, an **end** directive (**end parallel**) must be specified to demarcate the range (region) of the **parallel** directive.

Example directive_syntax_F_free_comment.1.f90 (**pre_omp_3.0**)

```
S-1     program main
S-2        use omp_lib
S-3        integer,parameter :: NT = 4
S-4
S-5        !$omp parallel do num_threads(NT)                !DIR 1
S-6        do i = 1,NT
S-7          write(*,'("thrd no", i2)') omp_get_thread_num()
S-8        end do
S-9
S-10       !$omp  parallel do  &     !continue line         !DIR 2
S-11       !$omp num_threads(NT)     !or !$omp&
S-12       do i = 1,NT
S-13         write(*,'("thrd no", i2)') omp_get_thread_num()
S-14       end do
S-15
S-16       !$omp parallel num_threads(NT)                   !DIR 3
S-17       !$omp do                                         !DIR 4
S-18       do i = 1,NT
S-19          write(*,'("thrd no", i2)') omp_get_thread_num()
S-20       end do
S-21       !$omp end parallel
S-22
S-23    end program
S-24
S-25    !     repeated 3 times, any order
S-26    !     OUTPUT: thrd no  0
S-27    !     OUTPUT: thrd no  1
S-28    !     OUTPUT: thrd no  2
S-29    !     OUTPUT: thrd no  3
```

As of OpenMP 5.1, **block** and **end block** statements can be used to designate a structured block for an OpenMP region, and any paired OpenMP **end** directive becomes optional, as shown in the next example. Note, the variables *i* and *thrd_no* are declared within the block structure and are hence private. It was necessary to explicitly declare the *i* variable, due to the **implicit none** statement; it could have also been declared outside the structured block.

Example directive_syntax_F_block.1.f90 (**omp_5.1**)

```
S-1   program main
S-2
S-3      use omp_lib
S-4      implicit none
S-5      integer,parameter :: NT = 2, chunks=3
S-6
S-7     !$omp parallel num_threads(NT)
S-8      block                          ! Fortran 2008 OMP 5.1
S-9        integer :: thrd_no,i
S-10       thrd_no= omp_get_thread_num()
S-11        !$omp do schedule(static,chunks)
S-12        do i = 1,NT*chunks
S-13           write(*,'("ndx=",i0.2," thrd_no=", i0.2)') i,thrd_no
S-14        end do
S-15     end block
S-16  end program
S-17
S-18  ! any order
S-19  ! OUTPUT: ndx=01 thrd_no=00
S-20  ! OUTPUT: ndx=02 thrd_no=00
S-21  ! OUTPUT: ndx=03 thrd_no=00
S-22  ! OUTPUT: ndx=04 thrd_no=01
S-23  ! OUTPUT: ndx=05 thrd_no=01
S-24  ! OUTPUT: ndx=06 thrd_no=01
```

A Fortran **BLOCK** construct may eliminate the need for a paired **end** directive for an OpenMP construct, as illustrated in the following example.

The first **parallel** construct is specified with an OpenMP loosely structured block (where the first executable construct is not a Fortran 2008 **BLOCK** construct). A paired **end** directive must end the OpenMP construct. The second **parallel** construct is specified with an OpenMP strictly structured block (consists only of a single Fortran **BLOCK** construct). The paired **end** directive is optional in this case, and is not used here.

The next two **parallel** directives form an enclosing outer **parallel** construct and a nested inner **parallel** construct. The first **end parallel** directive that subsequently appears terminates the inner **parallel** construct, because a paired **end** directive immediately following a

BLOCK construct that is a strictly structured block of an OpenMP construct is treated as the terminating end directive of that construct. The next **end parallel** directive is required to terminate the outer **parallel** construct.

Example directive_syntax_F_block.2.f90 (**omp_5.1**)

```
S-1   program main
S-2
S-3      use omp_lib
S-4      implicit none
S-5
S-6     !$omp parallel num_threads(2)
S-7        if( omp_get_thread_num() == 0 ) &
S-8            print*, "Loosely  structured block  -- end required."
S-9        block                                ! BLOCK Fortran 2008
S-10         if( omp_get_thread_num() == 0 ) &
S-11             print*, "                           --"
S-12       end block
S-13    !$omp end parallel
S-14
S-15    !$omp parallel num_threads(2)
S-16       block
S-17         if( omp_get_thread_num() == 0 ) &
S-18             print*, "Strictly structured block  -- end not required."
S-19       end block
S-20   !!$omp end parallel !is optional for strictly structured block
S-21
S-22    print*, "Sequential part"
S-23
S-24    !$omp parallel num_threads(2)                      !outer parallel
S-25       if( omp_get_thread_num() == 0 ) &
S-26           print*, "Outer, loosely  structured block."
S-27       !$omp parallel num_threads(2)                   !inner parallel
S-28          block
S-29            if( omp_get_thread_num() == 0 ) &
S-30            print*, "Inner, strictly structured block."
S-31          end block
S-32       !$omp end parallel
S-33    !$omp end parallel
S-34    ! Two end directives are required here.
S-35    ! A single "!$omp end parallel" terminator will fail.
S-36    ! 1st end directive is assumed to be for inner parallel construct.
S-37    ! 2nd end directive applies to outer parallel construct.
S-38
S-39    end program
```

```
S-40
S-41    !OUTPUT, in order:
S-42    ! Loosely  structured block  -- end required.
S-43    !                            --
S-44    ! Strictly structured block  -- end not required.
S-45    ! Sequential part
S-46    ! Outer, loosely  structured block.
S-47    ! Inner, strictly structured block.
S-48    ! Inner, strictly structured block.
```

Fortran

This page intentionally left blank

3 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 primary 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 primary 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.

MASKED CONSTRUCT

The **masked** construct is not a worksharing construct. The **masked** region is executed only by the primary thread. There is no implicit barrier (and flush) at the end of the **masked** region; hence the other threads of the team continue execution beyond code statements beyond the **masked** region. The **master** construct, which has been deprecated in OpenMP 5.1, has identical semantics to the **masked** construct with no **filter** clause.

3.1 A Simple Parallel Loop

The following example demonstrates how to parallelize a simple loop using the **parallel** worksharing-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 (**pre_omp_3.0**)

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

C / C++

Fortran

Example ploop.1.f (**pre_omp_3.0**)

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

Fortran

3.2 `parallel` Construct

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

C / C++

Example parallel.1.c (**`pre_omp_3.0`**)

```
S-1   #include <omp.h>
S-2
S-3   void subdomain(float *x, int istart, int ipoints)
S-4   {
S-5     int i;
S-6
S-7     for (i = 0; i < ipoints; i++)
S-8         x[istart+i] = 123.456;
S-9   }
S-10
S-11  void sub(float *x, int npoints)
S-12  {
S-13      int iam, nt, ipoints, istart;
S-14
S-15  #pragma omp parallel default(shared) private(iam,nt,ipoints,istart)
S-16      {
S-17          iam = omp_get_thread_num();
S-18          nt =  omp_get_num_threads();
S-19          ipoints = npoints / nt;     /* size of partition */
S-20          istart = iam * ipoints;  /* starting array index */
S-21          if (iam == nt-1)      /* last thread may do more */
S-22            ipoints = npoints - istart;
S-23          subdomain(x, istart, ipoints);
S-24      }
S-25  }
S-26
S-27  int main()
S-28  {
S-29      float array[10000];
S-30
S-31      sub(array, 10000);
S-32
S-33      return 0;
S-34  }
```

C / C++

Fortran

Example parallel.1.f (`pre_omp_3.0`)

```
S-1         SUBROUTINE SUBDOMAIN(X, ISTART, IPOINTS)
S-2             INTEGER ISTART, IPOINTS
S-3             REAL X(*)
S-4
S-5             INTEGER I
S-6
S-7             DO 100 I=1,IPOINTS
S-8                X(ISTART+I) = 123.456
S-9      100    CONTINUE
S-10
S-11        END SUBROUTINE SUBDOMAIN
S-12
S-13        SUBROUTINE SUB(X, NPOINTS)
S-14            INCLUDE "omp_lib.h"     ! or USE OMP_LIB
S-15
S-16            REAL X(*)
S-17            INTEGER NPOINTS
S-18            INTEGER IAM, NT, IPOINTS, ISTART
S-19
S-20    !$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,NPOINTS)
S-21
S-22            IAM = OMP_GET_THREAD_NUM()
S-23            NT =  OMP_GET_NUM_THREADS()
S-24            IPOINTS = NPOINTS/NT
S-25            ISTART = IAM * IPOINTS
S-26            IF (IAM .EQ. NT-1) THEN
S-27                IPOINTS = NPOINTS - ISTART
S-28            ENDIF
S-29            CALL SUBDOMAIN(X,ISTART,IPOINTS)
S-30
S-31    !$OMP END PARALLEL
S-32        END SUBROUTINE SUB
S-33
S-34        PROGRAM PAREXAMPLE
S-35            REAL ARRAY(10000)
S-36            CALL SUB(ARRAY, 10000)
S-37        END PROGRAM PAREXAMPLE
```

Fortran

3.3 `teams` Construct on Host

Originally the **`teams`** construct was created for devices (such as GPUs) for independent executions of a structured block by teams within a league (on SMs). It was only available through offloading with the **`target`** construct, and the execution of a **`teams`** region could only be directed to host execution by various means such as **`if`** and **`device`** clauses, and the **`OMP_TARGET_OFFLOAD`** environment variable.

In OpenMP 5.0 the **`teams`** construct was extended to enable the host to execute a **`teams`** region (without an associated **`target`** construct), with anticipation of further affinity and threading controls in future OpenMP releases.

In the example below the **`teams`** construct is used to create two teams, one to execute single precision code, and the other to execute double precision code. Two teams are required, and the thread limit for each team is set to 1/2 of the number of available processors.

C / C++

Example host_teams.1.c (**`omp_5.0`**)

```
S-1   #include <stdio.h>
S-2   #include <stdlib.h>
S-3   #include  <math.h>
S-4   #include   <omp.h>
S-5   #define    N 1000
S-6
S-7   int main(){
S-8      int     nteams_required=2, max_thrds, tm_id;
S-9      float   sp_x[N], sp_y[N], sp_a=0.0001e0;
S-10     double  dp_x[N], dp_y[N], dp_a=0.0001e0;
S-11
S-12     max_thrds = omp_get_num_procs()/nteams_required;
S-13
S-14     // Create 2 teams, each team works in a different precision
S-15     #pragma omp teams num_teams(nteams_required) \
S-16                       thread_limit(max_thrds)  private(tm_id)
S-17     {
S-18        tm_id = omp_get_team_num();
S-19
S-20        if( omp_get_num_teams() != 2 )   //if only getting 1, quit
S-21        { printf("error: Insufficient teams on host, 2 required\n");
S-22          exit(0);
S-23        }
S-24
S-25        if(tm_id == 0)  // Do Single Precision Work (SAXPY) with this team
S-26        {
S-27           #pragma omp parallel
S-28           {
```

```
S-29              #pragma omp for                                  //init
S-30              for(int i=0; i<N; i++){sp_x[i] = i*0.0001;   sp_y[i]=i; }
S-31
S-32              #pragma omp for simd simdlen(8)
S-33              for(int i=0; i<N; i++){sp_x[i] = sp_a*sp_x[i] + sp_y[i];}
S-34           }
S-35        }
S-36
S-37        if(tm_id == 1)  // Do Double Precision Work (DAXPY) with this team
S-38        {
S-39           #pragma omp parallel
S-40           {
S-41              #pragma omp for                                  //init
S-42              for(int i=0; i<N; i++){dp_x[i] = i*0.0001;   dp_y[i]=i; }
S-43
S-44              #pragma omp for simd simdlen(4)
S-45              for(int i=0; i<N; i++){dp_x[i] = dp_a*dp_x[i] + dp_y[i];}
S-46           }
S-47        }
S-48     }
S-49
S-50     printf("i=%d  sp|dp  %f %f \n",N-1, sp_x[N-1], dp_x[N-1]);
S-51     printf("i=%d  sp|dp  %f %f \n",N/2, sp_x[N/2], dp_x[N/2]);
S-52   //OUTPUT1:i=999  sp|dp  999.000000 999.000010
S-53   //OUTPUT2:i=500  sp|dp  500.000000 500.000005
S-54
S-55     return 0;
S-56  }
```

C / C++

Fortran

1 *Example host_teams.1.f90* (`omp_5.0`)

```
S-1   program main
S-2      use omp_lib
S-3      integer           :: nteams_required=2, max_thrds, tm_id
S-4      integer,parameter ::  N=1000
S-5      real              :: sp_x(N), sp_y(N), sp_a=0.0001e0
S-6      double precision  :: dp_x(N), dp_y(N), dp_a=0.0001d0
S-7
S-8      max_thrds = omp_get_num_procs()/nteams_required
S-9
S-10     !! Create 2 teams, each team works in a different precision
S-11     !$omp teams num_teams(nteams_required) thread_limit(max_thrds) &
S-12     !$omp&      private(tm_id)
S-13
S-14        tm_id = omp_get_team_num()
```

```
S-15
S-16          if( omp_get_num_teams() /= 2 ) then   !! if only getting 1, quit
S-17             stop "error: Insufficient teams on host, 2 required."
S-18          endif
S-19
S-20          !! Do Single Precision Work (SAXPY) with this team
S-21          if(tm_id == 0) then
S-22
S-23             !$omp parallel
S-24                !$omp do         !! init
S-25                do i = 1,N
S-26                   sp_x(i) = i*0.0001e0
S-27                   sp_y(i) = i
S-28                end do
S-29
S-30                !$omp do simd simdlen(8)
S-31                do i = 1,N
S-32                   sp_x(i) = sp_a*sp_x(i) + sp_y(i)
S-33                end do
S-34             !$omp end parallel
S-35
S-36          endif
S-37
S-38          !! Do Double Precision Work (DAXPY) with this team
S-39          if(tm_id == 1) then
S-40
S-41             !$omp parallel
S-42                !$omp do         !! init
S-43                do i = 1,N
S-44                   dp_x(i) = i*0.0001d0
S-45                   dp_y(i) = i
S-46                end do
S-47
S-48                !$omp do simd simdlen(4)
S-49                do i = 1,N
S-50                   dp_x(i) = dp_a*dp_x(i) + dp_y(i)
S-51                end do
S-52             !$omp end parallel
S-53
S-54          endif
S-55       !$omp end teams
S-56
S-57       write(*,'( "i=",i4," sp|dp= ", e15.7, d25.16  )') &
S-58                N, sp_x(N), dp_x(N)
S-59       write(*,'( "i=",i4," sp|dp= ", e15.7, d25.16  )') &
S-60                N/2, sp_x(N/2), dp_x(N/2)
S-61                !! i=1000 sp|dp=   0.1000000E+04   0.1000000010000000D+04
```

```
S-62              !! i= 500 sp|dp=   0.5000000E+03   0.5000000050000000D+03
S-63  end program
```

Fortran

3.4 Controlling the Number of Threads on Multiple Nesting Levels

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

C / C++

Example nthrs_nesting.1.c (**pre_omp_3.0**)

```
S-1   #include <stdio.h>
S-2   #include <omp.h>
S-3   int main (void)
S-4   {
S-5      omp_set_nested(1);
S-6      omp_set_dynamic(0);
S-7      #pragma omp parallel
S-8      {
S-9         #pragma omp parallel
S-10        {
S-11           #pragma omp single
S-12           {
S-13           /*
S-14           * If OMP_NUM_THREADS=2,3 was set, the following should print:
S-15           * Inner: num_thds=3
S-16           * Inner: num_thds=3
S-17           *
S-18           * If nesting is not supported, the following should print:
S-19           * Inner: num_thds=1
S-20           * Inner: num_thds=1
S-21           */
S-22              printf ("Inner: num_thds=%d\n", omp_get_num_threads());
S-23           }
S-24        }
S-25        #pragma omp barrier
S-26        omp_set_nested(0);
S-27        #pragma omp parallel
S-28        {
S-29           #pragma omp single
S-30           {
S-31           /*
```

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

C / C++

Fortran

Example nthrs_nesting.1.f (**pre_omp_3.0**)

```
        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:
```

```
S-23            ! Inner: num_thds= 1
S-24            ! Inner: num_thds= 1
S-25           print *, "Inner: num_thds=", omp_get_num_threads()
S-26    !$omp end single
S-27    !$omp end parallel
S-28    !$omp barrier
S-29    !$omp single
S-30            ! If OMP_NUM_THREADS=2,3 was set, the following should print:
S-31            ! Outer: num_thds= 2
S-32           print *, "Outer: num_thds=", omp_get_num_threads()
S-33    !$omp end single
S-34    !$omp end parallel
S-35            end
```

Fortran

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

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

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

C / C++

Example nthrs_dynamic.1.c (**`pre_omp_3.0`**)

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

C / C++

Fortran

Example nthrs_dynamic.1.f (**pre_omp_3.0**)

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

Fortran

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

C / C++

Example nthrs_dynamic.2.c (**pre_omp_3.0**)

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

C / C++

Fortran

Example nthrs_dynamic.2.f (**pre_omp_3.0**)

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

Fortran

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.

3.6 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 **do** constructs:

Example fort_do.1.f (**pre_omp_3.0**)

```
S-1         SUBROUTINE WORK(I, J)
S-2         INTEGER I,J
S-3         END SUBROUTINE WORK
S-4
S-5         SUBROUTINE DO_GOOD()
S-6           INTEGER I, J
S-7           REAL A(1000)
S-8
S-9           DO 100 I = 1,10
S-10    !$OMP   DO
S-11          DO 100 J = 1,10
S-12            CALL WORK(I,J)
S-13    100   CONTINUE      !  !$OMP ENDDO implied here
S-14
S-15    !$OMP   DO
S-16          DO 200 J = 1,10
S-17    200     A(I) = I + 1
S-18    !$OMP   ENDDO
S-19
S-20    !$OMP   DO
S-21          DO 300 I = 1,10
S-22            DO 300 J = 1,10
S-23              CALL WORK(I,J)
S-24    300   CONTINUE
S-25    !$OMP   ENDDO
S-26        END SUBROUTINE DO_GOOD
```

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 (**pre_omp_3.0**)

```
S-1         SUBROUTINE WORK(I, J)
S-2         INTEGER I,J
S-3         END SUBROUTINE WORK
S-4
S-5         SUBROUTINE DO_WRONG
S-6           INTEGER I, J
```

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

Fortran

3.7 `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 worksharing-loop construct, as follows:

C / C++

Example nowait.1.c (**`pre_omp_3.0`**)

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

C / C++

Fortran

Example nowait.1.f (`pre_omp_3.0`)

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

Fortran

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

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

C / C++

Example nowait.2.c (**pre_omp_3.0**)

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

C / C++

Fortran

Example nowait.2.f90 (**pre_omp_3.0**)

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

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

Fortran

3.8 `collapse` Clause

In the following example, the *k* and *j* loops are associated with the worksharing-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 worksharing-loop construct, it is not collapsed, and the *i* loop is executed sequentially in its entirety in every iteration of the collapsed *k* and *j* loop.

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

C / C++

Example collapse.1.c (`omp_3.0`)

```
S-1     void bar(float *a, int i, int j, int k);
S-2
S-3     int kl, ku, ks, jl, ju, js, il, iu,is;
S-4
S-5     void sub(float *a)
S-6     {
S-7         int i, j, k;
S-8
S-9         #pragma omp for collapse(2) private(i, k, j)
S-10        for (k=kl; k<=ku; k+=ks)
S-11           for (j=jl; j<=ju; j+=js)
S-12              for (i=il; i<=iu; i+=is)
S-13                 bar(a,i,j,k);
S-14    }
```

C / C++

Fortran

Example collapse.1.f (**omp_3.0**)

```
      subroutine sub(a)

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

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

      end subroutine
```

Fortran

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

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

C / C++

Example collapse.2.c (**omp_3.0**)

```
#include <stdio.h>
int main()
{
   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;
```

```
S-12                klast=k;
S-13             }
S-14          #pragma omp single
S-15          printf("%d %d\n", klast, jlast); //2 3
S-16       }
S-17    }
```

C / C++

Fortran

Example collapse.2.f (**omp_3.0**)

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

Fortran

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

In the example, the worksharing-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 worksharing-loop construct because an ordered region binds to the loop region arising from the worksharing-loop construct.

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

The code prints

```
0 1 1
0 1 2
0 2 1
1 2 2
1 3 1
1 3 2
```

C / C++

Example collapse.3.c (**omp_3.0**)

```
#include <omp.h>
#include <stdio.h>
void work(int a, int j, int k);
void sub()
{
   int j, k, a = 5;
   #pragma omp parallel num_threads(2)
   {
      #pragma omp for collapse(2) ordered private(j,k) schedule(static,3)
      for (k=1; k<=3; k++)
         for (j=1; j<=2; j++)
         {
            #pragma omp ordered
            printf("%d %d %d\n", omp_get_thread_num(), k, j);
            /* end ordered */
            work(a,j,k);
         }
   }
}
```

C / C++

Fortran

Example collapse.3.f (**omp_3.0**)

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

```
      enddo
!$omp end do
!$omp end parallel
      end program test
```

Fortran

The following example illustrates the collapse of a non-rectangular loop nest, a new feature in OpenMP 5.0. In a loop nest, a non-rectangular loop has a loop bound that references the iteration variable of an enclosing loop.

The motivation for this feature is illustrated in the example below that creates a symmetric correlation matrix for a set of variables. Note that the initial value of the second loop depends on the index variable of the first loop for the loops to be collapsed. Here the data are represented by a 2D array, each row corresponds to a variable and each column corresponds to a sample of the variable – the last two columns are the sample mean and standard deviation (for Fortran, rows and columns are swapped).

C / C++

Example collapse.4.c (`omp_5.0`)

```
#include <stdio.h>
#define N 20
#define M 10

// routine to calculate a
// For variable a[i]:
// a[i][0],...,a[i][n-1]   contains the n samples
// a[i][n]                 contains the sample mean
// a[i][n+1]               contains the standard deviation
extern void calc_a(int n,int m, float a[][N+2]);

int main(){
  float a[M][N+2], b[M][M];

  calc_a(N,M,a);

  #pragma omp parallel for collapse(2)
  for (int i = 0; i < M; i++)
     for (int j = i; j < M; j++)
     {
        float temp = 0.0f;
        for (int k = 0; k < N; k++)
           temp += (a[i][k]-a[i][N])*(a[j][k]-a[j][N]);

        b[i][j] = temp / (a[i][N+1] * a[j][N+1] * (N - 1));
        b[j][i] = b[i][j];
     }
```

```
S-28
S-29      printf("b[0][0] = %f, b[M-1][M-1] = %f\n", b[0][0], b[M-1][M-1]);
S-30
S-31      return 0;
S-32    }
```

C / C++

Fortran

Example collapse.4.f90 (`omp_5.0`)

```
S-1     module calc_m
S-2       interface
S-3       subroutine calc_a(n, m, a)
S-4       integer n, m
S-5       real a(n+2,m)
S-6       ! routine to calculate a
S-7       ! For variable a(*,j):
S-8       ! a(1,j),...,a(n,j)  contains the n samples
S-9       ! a(n+1,j)           contains the sample mean
S-10      ! a(n+2,j)           contains the standard deviation
S-11      end subroutine
S-12      end interface
S-13    end module
S-14
S-15    program main
S-16      use calc_m
S-17      integer, parameter :: N=20, M=10
S-18      real a(N+2,M), b(M,M)
S-19      real temp
S-20      integer i, j, k
S-21
S-22      call calc_a(N,M,a)
S-23
S-24      !$omp parallel do collapse(2) private(k,temp)
S-25      do i = 1, M
S-26         do j = i, M
S-27            temp = 0.0
S-28            do k = 1, N
S-29               temp = temp + (a(k,i)-a(N+1,i))*(a(k,j)-a(N+1,j))
S-30            end do
S-31
S-32            b(i,j) = temp / (a(N+2,i) * a(N+2,j) * (N - 1))
S-33            b(j,i) = b(i,j)
S-34         end do
S-35      end do
S-36
S-37      print *,"b(1,1) = ",b(1,1),", b(M,M) = ",b(M,M)
```

```
end program
```

Fortran

3.9 `linear` Clause in Loop Constructs

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

C / C++

Example linear_in_loop.1.c (**omp_4.5**)

```
#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;
}
```

C / C++

Fortran

Example linear_in_loop.1.f90 (**omp_4.5**)

```
S-1   program linear_loop
S-2      implicit none
S-3      integer, parameter :: N = 100
S-4      real :: a(N), b(N/2)
S-5      integer :: i, j
S-6
S-7      do i = 1, N
S-8         a(i) = i
S-9      end do
S-10
S-11     j = 0
S-12     !$omp parallel
S-13     !$omp do linear(j:1)
S-14     do i = 1, N, 2
S-15        j = j + 1
S-16        b(j) = a(i) * 2.0
S-17     end do
S-18     !$omp end parallel
S-19
S-20     print *, j, b(1), b(j)
S-21     ! print out: 50 2.0 198.0
S-22
S-23  end program
```

Fortran

3.10 `parallel sections` Construct

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

C / C++

Example psections.1.c (**pre_omp_3.0**)

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

C / C++

Fortran

Example psections.1.f (**pre_omp_3.0**)

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

Fortran

3.11 `firstprivate` Clause and `sections` Construct

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

C / C++

Example fpriv_sections.1.c (**`pre_omp_3.0`**)

```
S-1   #include <omp.h>
S-2   #include <stdio.h>
S-3   #define NT 4
S-4   int main( ) {
S-5       int section_count = 0;
S-6       omp_set_dynamic(0);
S-7       omp_set_num_threads(NT);
S-8   #pragma omp parallel
S-9   #pragma omp sections firstprivate( section_count )
S-10  {
S-11  #pragma omp section
S-12      {
S-13          section_count++;
S-14          /* may print the number one or two */
S-15          printf( "section_count %d\n", section_count );
S-16      }
S-17  #pragma omp section
S-18      {
S-19          section_count++;
S-20          /* may print the number one or two */
S-21          printf( "section_count %d\n", section_count );
S-22      }
S-23  }
S-24      return 0;
S-25  }
```

C / C++

Fortran

Example fpriv_sections.1.f90 (`pre_omp_3.0`)

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

Fortran

3.12 `single` Construct

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

C / C++

Example single.1.c (**pre_omp_3.0**)

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

C / C++

Fortran

Example single.1.f (**pre_omp_3.0**)

```
S-1         SUBROUTINE WORK1()
S-2         END SUBROUTINE WORK1
S-3
S-4         SUBROUTINE WORK2()
S-5         END SUBROUTINE WORK2
S-6
S-7         PROGRAM SINGLE_EXAMPLE
S-8   !$OMP PARALLEL
S-9
S-10  !$OMP SINGLE
S-11          print *, "Beginning work1."
S-12  !$OMP END SINGLE
S-13
S-14          CALL WORK1()
S-15
S-16  !$OMP SINGLE
```

```
S-17            print *, "Finishing work1."
S-18    !$OMP END SINGLE
S-19
S-20    !$OMP SINGLE
S-21            print *, "Finished work1 and beginning work2."
S-22    !$OMP END SINGLE NOWAIT
S-23
S-24            CALL WORK2()
S-25
S-26    !$OMP END PARALLEL
S-27
S-28          END PROGRAM SINGLE_EXAMPLE
```

Fortran

Fortran

3.13 `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 (`pre_omp_3.0`)

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

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 (**pre_omp_3.0**)

```
S-1         SUBROUTINE WSHARE2(AA, BB, CC, DD, EE, FF, N)
S-2         INTEGER N
S-3         REAL AA(N,N), BB(N,N), CC(N,N)
S-4         REAL DD(N,N), EE(N,N), FF(N,N)
S-5
S-6   !$OMP   PARALLEL
S-7   !$OMP     WORKSHARE
S-8               AA = BB
S-9               CC = DD
S-10  !$OMP     END WORKSHARE NOWAIT
S-11  !$OMP     WORKSHARE
S-12              EE = FF
S-13  !$OMP     END WORKSHARE
S-14  !$OMP   END PARALLEL
S-15        END SUBROUTINE WSHARE2
```

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

Example workshare.3.f (**pre_omp_3.0**)

```
S-1         SUBROUTINE WSHARE3(AA, BB, CC, DD, N)
S-2         INTEGER N
S-3         REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
S-4         REAL R
S-5           R=0
S-6   !$OMP   PARALLEL
S-7   !$OMP     WORKSHARE
S-8               AA = BB
S-9   !$OMP       ATOMIC UPDATE
S-10                R = R + SUM(AA)
S-11              CC = DD
S-12  !$OMP     END WORKSHARE
S-13  !$OMP   END PARALLEL
S-14        END SUBROUTINE WSHARE3
```

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

Each task gets worked on in order by the threads:

AA = BB then
CC = DD then

EE .ne. 0 then
FF = 1 / EE then
GG = HH

Example workshare.4.f (**pre_omp_3.0**)

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

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

Example workshare.5.f (**pre_omp_3.0**)

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

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

scalar variable is undefined after the assignment statement.

Example workshare.6.f (**pre_omp_3.0**)

```
      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 (**pre_omp_3.0**)

```
      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

3.14 `masked` Construct

The following example demonstrates the **`masked`** construct. In the example, the primary thread (thread number 0) keeps track of how many iterations have been executed and prints out a progress report in the iteration loop. The other threads skip the **`masked`** region without waiting. The **`filter`** clause can be used to specify a thread number other than the primary thread to execute a structured block, as illustrated by the second **`masked`** construct after the iteration loop. If the thread specified in a **`filter`** clause does not exist in the team then the structured block is not executed by any thread.

C / C++

Example masked.1.c (**`omp_5.1`**)

```
S-1   #include <stdio.h>
S-2
S-3   extern float average(float,float,float);
S-4
S-5   void masked_example( float* x, float* xold, int n, float tol )
S-6   {
S-7     int c, i, toobig;
S-8     float error, y;
S-9     c = 0;
S-10    #pragma omp parallel
S-11    {
S-12      do {
S-13        #pragma omp for private(i)
S-14        for( i = 1; i < n-1; ++i ){
S-15          xold[i] = x[i];
S-16        }
S-17        #pragma omp single
S-18        {
S-19          toobig = 0;
S-20        }
S-21        #pragma omp for private(i,y,error) reduction(+:toobig)
S-22        for( i = 1; i < n-1; ++i ){
S-23          y = x[i];
S-24          x[i] = average( xold[i-1], x[i], xold[i+1] );
S-25          error = y - x[i];
S-26          if( error > tol || error < -tol ) ++toobig;
S-27        }
S-28        #pragma omp masked          // primary thread (thread 0)
S-29        {
S-30          ++c;
S-31          printf( "iteration %d, toobig=%d\n", c, toobig );
S-32        }
S-33      } while( toobig > 0 );
S-34      #pragma omp barrier
```

```
      #pragma omp masked filter(1)  // thread 1
      {
        // The printf statement will not be executed
        // if the number of threads is less than 2.
        printf( "total number of iterations = %d\n", c );
      }
    }
}
```

C / C++

Fortran

Example masked.1.f (`omp_5.1`)

```
      SUBROUTINE MASKED_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     MASKED            ! primary thread (thread 0)
            C = C + 1
            PRINT *, 'Iteration ', C, 'TOOBIG=', TOOBIG
!$OMP     END MASKED
        ENDDO
!$OMP   BARRIER
!$OMP   MASKED FILTER(1)    ! thread 1
          ! The print statement will not be executed
          ! if the number of threads is less than 2.
          PRINT *, 'Total number of iterations =', C
```

```
S-35    !$OMP   END MASKED
S-36    !$OMP END PARALLEL
S-37          END SUBROUTINE MASKED_EXAMPLE
```

Fortran

3.15 `loop` Construct

The following example illustrates the use of the OpenMP 5.0 **`loop`** construct for the execution of a loop. The **`loop`** construct asserts to the compiler that the iterations of the loop are free of data dependencies and may be executed concurrently. It allows the compiler to use heuristics to select the parallelization scheme and compiler-level optimizations for the concurrency.

C / C++

Example loop.1.c (**`omp_5.0`**)

```
S-1     #include  <stdio.h>
S-2     #define N 100
S-3     int main()
S-4     {
S-5       float x[N], y[N];
S-6       float a = 2.0;
S-7       for(int i=0;i<N;i++){ x[i]=i; y[i]=0;}   // initialize
S-8
S-9       #pragma omp parallel
S-10      {
S-11        #pragma omp loop
S-12        for(int i = 0; i < N; ++i) y[i] = a*x[i] + y[i];
S-13      }
S-14      if(y[N-1] != (N-1)*2.0) printf("Error: 2*(N-1) != y[N-1]=%f",y[N-1]);
S-15    }
```

C / C++

Fortran

Example loop.1.f90 (**omp_5.0**)

```
S-1   program main
S-2     integer, parameter :: N=100
S-3     real :: x(N), y(N)
S-4     real :: a = 2.0e0
S-5
S-6     x=(/ (i,i=1,N) /); y=0.0e0                    !! initialize
S-7
S-8     !$omp parallel
S-9       !$omp loop
S-10         do i=1,N; y(i) = a*x(i) + y(i); enddo
S-11    !$omp end parallel
S-12
S-13    if(y(N) /= N*2.0e0) print*,"Error: 2*N /= y(N); y(N)=",y(N)
S-14  end program
```

Fortran

The following example shows the use of the orphaned **loop** construct. Since the function *foo()* is not lexically nested inside of the **teams** region it needs to specify the **bind** clause. The first **loop** construct binds to the **teams** region from where the function *foo* is called. Binding to **teams** allows thread-level parallelism to be available for the second **loop** construct. The loop iterations can be executed concurrently, thus allowing implementations to perform various loop nest optimizations including reordering of the *i* and *j* loops. The **loop** construct can be implemented with the use of additional threads or some other concurrency mechanism, which allows better use of hardware resources while also allowing sequential optimizations, reordering, tiling etc.

For example, the first **loop** construct could be implemented as if it was specified as **distribute parallel for** and the second **loop** construct as if it was specified as **simd** if the hardware can support SIMD operations.

C / C++

Example loop.2.c (**omp_5.0**)

```
S-1   #include  <stdio.h>
S-2   #define N 1024
S-3
S-4   int x[N][N];
S-5   int y[N], z[N];
S-6
S-7   void foo() {
S-8   // i-loop distributed across encountering league of teams
S-9   #pragma omp loop bind(teams)
S-10      for (int i = 0; i < N; i++) {
S-11      // this loop has an implicit bind(thread)
```

```
S-12        #pragma omp loop
S-13          for (int j = 0; j < N; j++) {
S-14            x[i][j] += y[i]*z[i];
S-15          }
S-16        }
S-17    }
S-18
S-19    int main(){
S-20     int error = 0;
S-21
S-22      for (int i = 0; i < N; i++) {
S-23        for (int j = 0; j < N; j++) {
S-24           x[i][j] = 0;
S-25        }
S-26      }
S-27
S-28      for (int i = 0; i < N; i++) {
S-29        y[i] = i;
S-30        z[i] = i+1;
S-31      }
S-32
S-33    #pragma omp teams num_teams(4)
S-34      {
S-35          foo();
S-36      }
S-37
S-38    //check values
S-39       for (int i = 0; i < N; i++) {
S-40        for (int j = 0; j < N; j++) {
S-41           if( x[i][j] != i * (i+1))
S-42             error++;
S-43        }
S-44      }
S-45      if(error) {
S-46        printf("FAILED\n");
S-47        return 1;
S-48      }
S-49      printf("PASSED\n");
S-50      return 0;
S-51    }
```

C / C++

Example loop.2.f90 (`omp_5.0`)

```
S-1   module xyz_data
S-2     integer, parameter :: N=1024
S-3     integer :: x(N,N)
S-4     integer :: y(N), z(N)
S-5
S-6   contains
S-7     subroutine foo()
S-8       integer :: i, j
S-9
S-10      !! i-loop distributed across encountering league of
S-11      !! teams
S-12      !$omp loop bind(teams)
S-13      do i = 1, N
S-14        !! this loop has an implicit bind(thread)
S-15        !$omp loop
S-16        do j = 1, N
S-17          x(j,i) = x(j,i) + y(i)*z(i)
S-18        end do
S-19      end do
S-20    end subroutine
S-21  end module
S-22
S-23  program main
S-24   use xyz_data
S-25   integer ::  error = 0
S-26
S-27    do i = 1, N
S-28      do j = 1, N
S-29        x(j,i) = 0
S-30      end do
S-31    end do
S-32
S-33    do i = 1, N
S-34      y(i) = i
S-35      z(i) = i + 1
S-36    end do
S-37
S-38    !$omp teams num_teams(4)
S-39    call foo()
S-40    !$omp end teams
S-41
S-42  !!check values
S-43    do i = 1, N
S-44      do j = 1, N
```

```
S-45          if( x(j,i) /= i * (i+1) ) then
S-46            error = error + 1
S-47          endif
S-48        enddo
S-49      enddo
S-50
S-51      if(error .gt. 0) then
S-52        print*, "FAILED"
S-53        stop 1
S-54      end if
S-55
S-56      print*, "PASSED"
S-57
S-58    end program
```

Fortran

C++

3.16 Parallel Random Access Iterator Loop

The following example shows a parallel random access iterator loop.

Example pra_iterator.1.cpp (`omp_3.0`)

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

C++

3.17 `omp_set_dynamic` and `omp_set_num_threads` Routines

Some programs rely on a fixed, pre-specified number of threads to execute correctly. Because the default setting for the dynamic adjustment of the number of threads is implementation defined, such

programs can choose to turn off the dynamic threads capability and set the number of threads explicitly to ensure portability. The following example shows how to do this using **omp_set_dynamic**, and **omp_set_num_threads**.

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

C / C++

Example set_dynamic_nthrs.1.c (**pre_omp_3.0**)

```
S-1   #include <omp.h>
S-2   #include <stdlib.h>
S-3
S-4   void do_by_16(float *x, int iam, int ipoints) {}
S-5
S-6   void dynthreads(float *x, int npoints)
S-7   {
S-8     int iam, ipoints;
S-9
S-10    omp_set_dynamic(0);
S-11    omp_set_num_threads(16);
S-12
S-13    #pragma omp parallel shared(x, npoints) private(iam, ipoints)
S-14    {
S-15      if (omp_get_num_threads() != 16)
S-16        abort();
S-17
S-18      iam = omp_get_thread_num();
S-19      ipoints = npoints/16;
S-20      do_by_16(x, iam, ipoints);
S-21    }
S-22  }
```

C / C++

Fortran

Example set_dynamic_nthrs.1.f (`pre_omp_3.0`)

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

Fortran

3.18 `omp_get_num_threads` Routine

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

C / C++

Example get_nthrs.1.c (**`pre_omp_3.0`**)

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

C / C++

Fortran

Example get_nthrs.1.f (**`pre_omp_3.0`**)

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

Fortran

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

C / C++

Example get_nthrs.2.c (`pre_omp_3.0`)

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

C / C++

Fortran

Example get_nthrs.2.f (`pre_omp_3.0`)

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

Fortran

This page intentionally left blank

4 OpenMP Affinity

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-Threading) 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 primary thread's place, by setting the **OMP_PROC_BIND** environment variable or the **proc_bind** clause to **close**, **spread**, or **primary** (**master** has been deprecated), 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.

4.1 `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 Figure 4.1. 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.

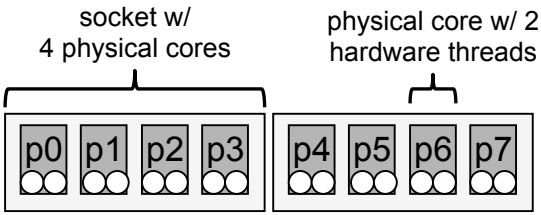


FIGURE 4.1: A machine architecture with two quad-core processors

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

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

or

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

4.1.1 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 / C++

Example affinity.1.c (**`omp_4.0`**)

```
S-1  void work();
S-2
S-3  int main()
S-4  {
S-5
S-6  #pragma omp parallel proc_bind(spread) num_threads(4)
S-7     {
```

```
S-8         work();
S-9       }
S-10
S-11      return 0;
S-12
S-13    }
```

C / C++

Fortran

Example affinity.1.f (`omp_4.0`)

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

Fortran

It is unspecified on which place the primary thread is initially started. If the primary 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 primary 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 primary thread) execute on the parent's place. The next T/P threads execute on the next place in the place partition, and so on, with wrap around.

C / C++

Example affinity.2.c (`omp_4.0`)

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

C / C++

Fortran

Example affinity.2.f90 (`omp_4.0`)

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

Fortran

It is unspecified on which place the primary thread is initially started. If the primary 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 primary 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

4.1.2 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.

C / C++

Example affinity.3.c (**omp_4.0**)

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

C / C++

Fortran

Example affinity.3.f (**omp_4.0**)

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

Fortran

It is unspecified on which place the primary thread is initially started. If the primary 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 primary 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 primary thread) execute on the parent's place. The next *T/P* threads execute on the next place in the place partition, and so on, with wrap around. The place partition is not changed by the **close** policy.

C / C++

Example affinity.4.c (**omp_4.0**)

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

C / C++

Fortran

Example affinity.4.f90 (**omp_4.0**)

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

Fortran

It is unspecified on which place the primary thread is initially started. If the primary 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 primary 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

4.1.3 Primary Affinity Policy

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

C / C++

Example affinity.5.c (**omp_5.1**)

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

C / C++

Fortran

Example affinity.5.f (**omp_5.1**)

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

Fortran

It is unspecified on which place the primary thread is initially started. If the primary 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 primary 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

4.2 Task Affinity

The next example illustrates the use of the **affinity** clause with a **task** construct. The variables in the **affinity** clause provide a hint to the runtime that the task should execute "close" to the physical storage location of the variables. For example, on a two-socket platform with a local memory component close to each processor socket, the runtime will attempt to schedule the task execution on the socket where the storage is located.

Because the C/C++ code employs a pointer, an array section is used in the **affinity** clause. Fortran code can use an array reference to specify the storage, as shown here.

Note, in the second task of the C/C++ code the *B* pointer is declared shared. Otherwise, by default, it would be firstprivate since it is a local variable, and would probably be saved for the second task before being assigned a storage address by the first task. Also, one might think it reasonable to use the **affinity** clause **affinity(***B[:N]***)** on the second **task** construct. However, the storage behind *B* is created in the first task, and the array section reference may not be valid when the second task is generated. The use of the *A* array is sufficient for this case, because one would expect the storage for *A* and *B* would be physically "close" (as provided by the hint in the first task).

C / C++

Example affinity.6.c (**omp_5.0**)

```
S-1   double * alloc_init_B(double *A, int N);
S-2   void     compute_on_B(double *B, int N);
S-3
S-4   void task_affinity(double *A, int N)
S-5   {
S-6      double * B;
S-7      #pragma omp task depend(out:B) shared(B) affinity(A[0:N])
S-8      {
S-9        B = alloc_init_B(A,N);
S-10     }
S-11
S-12     #pragma omp task depend( in:B) shared(B) affinity(A[0:N])
S-13     {
S-14       compute_on_B(B,N);
```

```
S-15      }
S-16
S-17      #pragma omp taskwait
S-18   }
```

C / C++

Fortran

Example affinity.6.f90 (`omp_5.0`)

```
 S-1   subroutine task_affinity(A, N)
 S-2
 S-3     external alloc_init_B
 S-4     external compute_on_B
 S-5     double precision, allocatable :: B(:)
 S-6
 S-7      !$omp task depend(out:B) shared(B) affinity(A)
 S-8        call alloc_init_B(B,A)
 S-9      !$omp end task
S-10
S-11      !$omp task depend(in:B) shared(B) affinity(A)
S-12        call compute_on_B(B)
S-13      !$omp end task
S-14
S-15      !$omp taskwait
S-16
S-17   end subroutine
```

Fortran

4.3 Affinity Display

The following examples illustrate ways to display thread affinity. Automatic display of affinity can be invoked by setting the **OMP_DISPLAY_AFFINITY** environment variable to TRUE. The format of the output can be customized by setting the **OMP_AFFINITY_FORMAT** environment variable to an appropriate string. Also, there are API calls for the user to display thread affinity at selected locations within code.

For the first example the environment variable **OMP_DISPLAY_AFFINITY** has been set to TRUE, and execution occurs on an 8-core system with **OMP_NUM_THREADS** set to 8.

The affinity for the primary thread is reported through a call to the API **omp_display_affinity()** routine. For default affinity settings the report shows that the primary thread can execute on any of the cores. In the following parallel region the affinity for each of the team threads is reported automatically since the **OMP_DISPLAY_AFFINITY** environment variable has been set to TRUE.

These two reports are often useful (as in hybrid codes using both MPI and OpenMP) to observe the affinity (for an MPI task) before the parallel region, and during an OpenMP parallel region. Note: the next parallel region uses the same number of threads as in the previous parallel region and affinities are not changed, so affinity is NOT reported.

In the last parallel region, the thread affinities are reported because the thread affinity has changed.

C / C++

Example affinity_display.1.c (`omp_5.0`)

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

int main(void){                    //MAX threads = 8, single socket system

   //API call-- Displays Affinity of Primary Thread
   omp_display_affinity(NULL);

   // API CALL OUTPUT (default format):
   // team_num= 0, nesting_level= 0, thread_num= 0,
   // thread_affinity= 0,1,2,3,4,5,6,7

   // OMP_DISPLAY_AFFINITY=TRUE, OMP_NUM_THREADS=8
   #pragma omp parallel num_threads(omp_get_num_procs())
   {
     if(omp_get_thread_num()==0)
        printf("1st Parallel Region -- Affinity Reported \n");

   // DISPLAY OUTPUT (default format) has been sorted:
   // team_num= 0, nesting_level= 1, thread_num= 0, thread_affinity= 0
   // team_num= 0, nesting_level= 1, thread_num= 1, thread_affinity= 1
   // ...
   // team_num= 0, nesting_level= 1, thread_num= 7, thread_affinity= 7

      // doing work here
   }

   #pragma omp parallel num_threads( omp_get_num_procs() )
   {
      if(omp_get_thread_num()==0)
         printf("%s%s\n","Same Affinity as in Previous Parallel Region",
                         " -- no Affinity Reported\n");

   // NO AFFINITY OUTPUT:
   //(output in 1st parallel region only for OMP_DISPLAY_AFFINITY=TRUE)

      // doing more work here
```

```
S-38        }
S-39
S-40        // Report Affinity for 1/2 number of threads
S-41        #pragma omp parallel num_threads( omp_get_num_procs()/2 )
S-42        {
S-43          if(omp_get_thread_num()==0)
S-44             printf("Report Affinity for using 1/2 of max threads.\n");
S-45
S-46        // DISPLAY OUTPUT (default format) has been sorted:
S-47        // team_num= 0, nesting_level= 1, thread_num= 0, thread_affinity= 0,1
S-48        // team_num= 0, nesting_level= 1, thread_num= 1, thread_affinity= 2,3
S-49        // team_num= 0, nesting_level= 1, thread_num= 2, thread_affinity= 4,5
S-50        // team_num= 0, nesting_level= 1, thread_num= 3, thread_affinity= 6,7
S-51
S-52          // do work
S-53        }
S-54
S-55        return 0;
S-56    }
```

C / C++

Fortran

1 *Example affinity_display.1.f90* (`omp_5.0`)

```
S-1     program affinity_display        ! MAX threads = 8, single socket system
S-2
S-3        use omp_lib
S-4        implicit none
S-5        character(len=0) :: null
S-6
S-7        ! API call - Displays Affinity of Primary Thread
S-8        call omp_display_affinity(null)
S-9
S-10       ! API CALL OUTPUT (default format):
S-11       ! team_num= 0, nesting_level= 0, thread_num= 0, &
S-12       !   thread_affinity= 0,1,2,3,4,5,6,7
S-13
S-14
S-15       ! OMP_DISPLAY_AFFINITY=TRUE, OMP_NUM_THREADS=8
S-16
S-17       !$omp parallel num_threads(omp_get_num_procs())
S-18
S-19         if(omp_get_thread_num()==0) then
S-20            print*, "1st Parallel Region  -- Affinity Reported"
S-21         endif
S-22
S-23         ! DISPLAY OUTPUT (default format) has been sorted:
```

```
S-24          ! team_num= 0, nesting_level= 1, thread_num= 0, thread_affinity= 0
S-25          ! team_num= 0, nesting_level= 1, thread_num= 1, thread_affinity= 1
S-26          ! ...
S-27          ! team_num= 0, nesting_level= 1, thread_num= 7, thread_affinity= 7
S-28
S-29           ! doing work here
S-30
S-31        !$omp end parallel
S-32
S-33        !$omp parallel num_threads( omp_get_num_procs() )
S-34
S-35          if(omp_get_thread_num()==0) then
S-36             print*, "Same Affinity in Parallel Region -- no Affinity Reported"
S-37          endif
S-38
S-39          ! NO AFFINITY OUTPUT:
S-40          ! (output in 1st parallel region only for
S-41          !  OMP_DISPLAY_AFFINITY=TRUE)
S-42
S-43           ! doing more work here
S-44
S-45        !$omp end parallel
S-46
S-47        ! Report Affinity for 1/2 number of threads
S-48        !$omp parallel num_threads( omp_get_num_procs()/2 )
S-49
S-50          if(omp_get_thread_num()==0) then
S-51             print*, "Altered Affinity in Parallel Region -- Affinity Reported"
S-52          endif
S-53
S-54          ! DISPLAY OUTPUT (default format) has been sorted:
S-55          ! team_num= 0, nesting_level= 1, thread_num= 0, &
S-56          !   thread_affinity= 0,1
S-57          ! team_num= 0, nesting_level= 1, thread_num= 1, &
S-58          !   thread_affinity= 2,3
S-59          ! team_num= 0, nesting_level= 1, thread_num= 2, &
S-60          !   thread_affinity= 4,5
S-61          ! team_num= 0, nesting_level= 1, thread_num= 3, &
S-62          !   thread_affinity= 6,7
S-63
S-64           ! do work
S-65
S-66        !$omp end parallel
S-67
S-68     end program
```

Fortran

In the following example 2 threads are forked, and each executes on a socket. Next, a nested parallel region runs half of the available threads on each socket.

These OpenMP environment variables have been set:

```
export OMP_PROC_BIND="TRUE"
export OMP_NUM_THREADS="2,4"
export OMP_PLACES="{0,2,4,6},{1,3,5,7}"
export OMP_AFFINITY_FORMAT="nest_level= %L, parent_thrd_num= %a,
thrd_num= %n, thrd_affinity= %A"
```

where the numbers correspond to core ids for the system. Note, **OMP_DISPLAY_AFFINITY** is not set and is FALSE by default. This example shows how to use API routines to perform affinity display operations.

For each of the two first-level threads the **OMP_PLACES** variable specifies a place with all the core-ids of the socket ({0,2,4,6} for one thread and {1,3,5,7} for the other). (As is sometimes the case in 2-socket systems, one socket may consist of the even id numbers, while the other may have the odd id numbers.) The affinities are printed according to the **OMP_AFFINITY_FORMAT** format: providing the parallel nesting level (*%L*), the ancestor thread number (*%a*), the thread number (*%n*) and the thread affinity (*%A*). In the nested parallel region within the *socket_work* routine the affinities for the threads on each socket are printed according to this format.

C / C++

Example affinity_display.2.c (**omp_5.0**)

```
S-1   #include <stdio.h>
S-2   #include <stdlib.h>
S-3   #include <omp.h>
S-4
S-5   void socket_work(int socket_num, int n_thrds);
S-6
S-7   int main(void)
S-8   {
S-9      int n_sockets, socket_num, n_thrds_on_socket;
S-10
S-11     omp_set_nested(1);             // or env var= OMP_NESTED=true
S-12     omp_set_max_active_levels(2); // or env var= OMP_MAX_ACTIVE_LEVELS=2
S-13
S-14     n_sockets         = omp_get_num_places();
S-15     n_thrds_on_socket = omp_get_place_num_procs(0);
S-16
S-17  // OMP_NUM_THREADS=2,4
S-18  // OMP_PLACES="{0,2,4,6},{1,3,5,7}"  #2 sockets; even/odd proc-ids
S-19  // OMP_AFFINITY_FORMAT=\
S-20  // "nest_level= %L, parent_thrd_num= %a, thrd_num= %n, thrd_affinity= %A"
S-21
S-22     #pragma omp parallel num_threads(n_sockets) private(socket_num)
```

```
S-23        {
S-24           socket_num = omp_get_place_num();
S-25
S-26           if(socket_num==0)
S-27              printf(" LEVEL 1 AFFINITIES 1 thread/socket, %d sockets:\n\n",
S-28                     n_sockets);
S-29
S-30           // not needed if OMP_DISPLAY_AFFINITY=TRUE
S-31           omp_display_affinity(NULL);
S-32
S-33     // OUTPUT:
S-34     // LEVEL 1 AFFINITIES 1 thread/socket, 2 sockets:
S-35     // nest_level= 1, parent_thrd_num= 0, thrd_num= 0, thrd_affinity= 0,2,4,6
S-36     // nest_level= 1, parent_thrd_num= 0, thrd_num= 1, thrd_affinity= 1,3,5,7
S-37
S-38           socket_work(socket_num, n_thrds_on_socket);
S-39        }
S-40
S-41      return 0;
S-42     }
S-43
S-44     void socket_work(int socket_num, int n_thrds)
S-45     {
S-46        #pragma omp parallel num_threads(n_thrds)
S-47        {
S-48           if(omp_get_thread_num()==0)
S-49              printf(" LEVEL 2 AFFINITIES, %d threads on socket %d\n",
S-50                     n_thrds, socket_num);
S-51
S-52           // not needed if OMP_DISPLAY_AFFINITY=TRUE
S-53           omp_display_affinity(NULL);
S-54
S-55      // OUTPUT:
S-56      // LEVEL 2 AFFINITIES, 4 threads on socket 0
S-57      // nest_level= 2, parent_thrd_num= 0, thrd_num= 0, thrd_affinity= 0
S-58      // nest_level= 2, parent_thrd_num= 0, thrd_num= 1, thrd_affinity= 2
S-59      // nest_level= 2, parent_thrd_num= 0, thrd_num= 2, thrd_affinity= 4
S-60      // nest_level= 2, parent_thrd_num= 0, thrd_num= 3, thrd_affinity= 6
S-61
S-62      // LEVEL 2 AFFINITIES, 4 threads on socket 1
S-63      // nest_level= 2, parent_thrd_num= 1, thrd_num= 0, thrd_affinity= 1
S-64      // nest_level= 2, parent_thrd_num= 1, thrd_num= 1, thrd_affinity= 3
S-65      // nest_level= 2, parent_thrd_num= 1, thrd_num= 2, thrd_affinity= 5
S-66      // nest_level= 2, parent_thrd_num= 1, thrd_num= 3, thrd_affinity= 7
S-67
S-68           // ... Do Some work on Socket
S-69        }
```

```
}
```

C / C++

Fortran

Example affinity_display.2.f90 (`omp_5.0`)

```
program affinity_display

   use omp_lib
   implicit none
   character(len=0) :: null
   integer          :: n_sockets, socket_num, n_thrds_on_socket;

   call omp_set_nested(.true.)        ! or env var= OMP_NESTED=true
   call omp_set_max_active_levels(2)  ! or env var= OMP_MAX_ACTIVE_LEVELS=2

   n_sockets         = omp_get_num_places()
   n_thrds_on_socket = omp_get_place_num_procs(0)

    ! OMP_NUM_THREADS=2,4
    ! OMP_PLACES="{0,2,4,6},{1,3,5,7}"  #2 sockets; even/odd proc-ids
    ! OMP_AFFINITY_FORMAT=\
    !"nest_level= %L, parent_thrd_num= %a, thrd_num= %n, thrd_affinity= %A"

   !$omp parallel num_threads(n_sockets) private(socket_num)

     socket_num = omp_get_place_num()

     if(socket_num==0) then
       write(*,'("LEVEL 1 AFFINITIES 1 thread/socket ",i0," sockets")') &
             n_sockets
     endif

     call omp_display_affinity(null)  ! not needed
                                      ! if OMP_DISPLAY_AFFINITY=TRUE

       ! OUTPUT:
       ! LEVEL 1 AFFINITIES 1 thread/socket, 2 sockets:
       ! nest_level= 1, parent_thrd_num= 0, thrd_num= 0, &
       !   thrd_affinity= 0,2,4,6
       ! nest_level= 1, parent_thrd_num= 0, thrd_num= 1, &
       !   thrd_affinity= 1,3,5,7

     call socket_work(socket_num, n_thrds_on_socket)

   !$omp end parallel

```

```
S-42    end program
S-43
S-44    subroutine socket_work(socket_num, n_thrds)
S-45       use omp_lib
S-46       implicit none
S-47       integer :: socket_num, n_thrds
S-48       character(len=0) :: null
S-49
S-50       !$omp parallel num_threads(n_thrds)
S-51
S-52          if(omp_get_thread_num()==0) then
S-53          write(*,'("LEVEL 2 AFFINITIES, ",i0," threads on socket ",i0)') &
S-54                n_thrds,socket_num
S-55          endif
S-56
S-57          call omp_display_affinity(null)  ! not needed
S-58                                           ! if OMP_DISPLAY_AFFINITY=TRUE
S-59
S-60          ! OUTPUT:
S-61          ! LEVEL 2 AFFINITIES, 4 threads on socket 0
S-62          ! nest_level= 2, parent_thrd_num= 0, thrd_num= 0, thrd_affinity= 0
S-63          ! nest_level= 2, parent_thrd_num= 0, thrd_num= 1, thrd_affinity= 2
S-64          ! nest_level= 2, parent_thrd_num= 0, thrd_num= 2, thrd_affinity= 4
S-65          ! nest_level= 2, parent_thrd_num= 0, thrd_num= 3, thrd_affinity= 6
S-66
S-67          ! LEVEL 2 AFFINITIES, 4 thrds on socket 1
S-68          ! nest_level= 2, parent_thrd_num= 1, thrd_num= 0, thrd_affinity= 1
S-69          ! nest_level= 2, parent_thrd_num= 1, thrd_num= 1, thrd_affinity= 3
S-70          ! nest_level= 2, parent_thrd_num= 1, thrd_num= 2, thrd_affinity= 5
S-71          ! nest_level= 2, parent_thrd_num= 1, thrd_num= 3, thrd_affinity= 7
S-72
S-73          ! ... Do Some work on Socket
S-74
S-75       !$omp end parallel
S-76
S-77    end subroutine
```

Fortran

The next example illustrates more details about affinity formatting. First, the **omp_get_affinity_format()** API routine is used to obtain the default format. The code checks to make sure the storage provides enough space to hold the format. Next, the **omp_set_affinity_format()** API routine sets a user-defined format: *host=%20H thrd_num=%0.4n binds_to=%A*.

The host, thread number and affinity fields are specified by *%20H*, *%0.4n* and *%A*: *H*, *n* and *A* are single character "short names" for the host, thread_num and thread_affinity data to be printed, with format sizes of *20*, *4*, and "size as needed". The period (.) indicates that the field is displayed

right-justified (default is left-justified) and the "0" indicates that any unused space is to be prefixed with zeros (e.g. instead of "1", "0001" is displayed for the field size of 4).

Within the parallel region the affinity for each thread is captured by **omp_capture_affinity()** into a buffer array with elements indexed by the thread number (*thrd_num*). After the parallel region, the thread affinities are printed in thread-number order.

If the storage area in buffer is inadequate for holding the affinity data, the stored affinity data is truncated. The maximum value for the number of characters (*nchars*) returned by **omp_capture_affinity** is captured by the **reduction(max:** *max_req_store***)** clause and the *if(nchars >= max_req_store) max_req_store=nchars* statement. It is used to report possible truncation (if *max_req_store* > *buffer_store*).

C / C++

Example affinity_display.3.c (**omp_5.0**)

```
S-1   #include <stdio.h>
S-2   #include <stdlib.h>  // also null is in <stddef.h>
S-3   #include <stddef.h>
S-4   #include <string.h>
S-5   #include <omp.h>
S-6
S-7   #define FORMAT_STORE   80
S-8   #define BUFFER_STORE   80
S-9
S-10  int main(void){
S-11
S-12     int i, n, thrd_num, max_req_store;
S-13     size_t nchars;
S-14
S-15     char default_format[FORMAT_STORE];
S-16     char my_format[]  = "host=%20H thrd_num=%0.4n binds_to=%A";
S-17     char **buffer;
S-18
S-19
S-20     // CODE SEGMENT 1         AFFINITY FORMAT
S-21
S-22     // Get and Display Default Affinity Format
S-23
S-24     nchars = omp_get_affinity_format(default_format,(size_t)FORMAT_STORE);
S-25     printf("Default Affinity Format is: %s\n",default_format);
S-26
S-27     if(nchars >= FORMAT_STORE){
S-28        printf("Caution: Reported Format is truncated.  Increase\n");
S-29        printf("         FORMAT_STORE to %d.\n", nchars+1);
S-30     }
S-31
```

```
S-32        // Set Affinity Format
S-33
S-34        omp_set_affinity_format(my_format);
S-35        printf("Affinity Format set to: %s\n",my_format);
S-36
S-37
S-38        // CODE SEGMENT 2          CAPTURE AFFINITY
S-39
S-40        // Set up buffer for affinity of n threads
S-41
S-42        n = omp_get_num_procs();
S-43        buffer = (char **)malloc( sizeof(char *) * n );
S-44        for(i=0;i<n;i++){
S-45           buffer[i]=(char *)malloc( sizeof(char) * BUFFER_STORE);
S-46        }
S-47
S-48        // Capture Affinity using Affinity Format set above.
S-49        // Use max reduction to check size of buffer areas
S-50        max_req_store = 0;
S-51        #pragma omp parallel private(thrd_num,nchars) \
S-52                             reduction(max:max_req_store)
S-53        {
S-54           //safety: don't exceed # of buffers
S-55           if(omp_get_num_threads()>n) exit(1);
S-56
S-57           thrd_num=omp_get_thread_num();
S-58           nchars=omp_capture_affinity(buffer[thrd_num],
S-59                                       (size_t)BUFFER_STORE,NULL);
S-60           if(nchars > max_req_store) max_req_store=nchars;
S-61
S-62           // ...
S-63        }
S-64
S-65        for(i=0;i<n;i++){
S-66           printf("thrd_num= %d, affinity: %s\n", i,buffer[i]);
S-67        }
S-68           // For 4 threads with OMP_PLACES='{0,1},{2,3},{4,5},{6,7}'
S-69           // Format    host=%20H thrd_num=%0.4n binds_to=%A
S-70
S-71           // affinity: host=hpc.cn567            thrd_num=0000 binds_to=0,1
S-72           // affinity: host=hpc.cn567            thrd_num=0001 binds_to=2,3
S-73           // affinity: host=hpc.cn567            thrd_num=0002 binds_to=4,5
S-74           // affinity: host=hpc.cn567            thrd_num=0003 binds_to=6,7
S-75
S-76
S-77        if(max_req_store>=BUFFER_STORE){
S-78           printf("Caution: Affinity string truncated.  Increase\n");
```

```
S-79          printf("         BUFFER_STORE to %d\n",max_req_store+1);
S-80       }
S-81
S-82       for(i=0;i<n;i++) free(buffer[i]);
S-83       free (buffer);
S-84
S-85       return 0;
S-86    }
```

C / C++

Fortran

Example affinity_display.3.f90 (`omp_5.0`)

```
 S-1    program affinity_display
 S-2       use omp_lib
 S-3       implicit none
 S-4       integer, parameter :: FORMAT_STORE=80
 S-5       integer, parameter :: BUFFER_STORE=80
 S-6
 S-7       integer            :: i, n, thrd_num, nchars, max_req_store
 S-8
 S-9       character(FORMAT_STORE)     :: default_format
S-10       character(*), parameter     :: my_format = &
S-11                                      "host=%20H thrd_num=%0.4n binds_to=%A"
S-12       character(:), allocatable   :: buffer(:)
S-13       character(len=0)            :: null
S-14
S-15
S-16    !  CODE SEGMENT 1         AFFINITY FORMAT
S-17
S-18    !                         Get and Display Default Affinity Format
S-19
S-20       nchars = omp_get_affinity_format(default_format)
S-21       print*,"Default Affinity Format: ", trim(default_format)
S-22
S-23       if( nchars > FORMAT_STORE) then
S-24          print*,"Caution: Reported Format is truncated.  Increase"
S-25          print*,"         FORMAT_STORE to ", nchars
S-26       endif
S-27
S-28    !                         Set Affinity Format
S-29
S-30       call omp_set_affinity_format(my_format)
S-31       print*,"Affinity Format set to: ", my_format
S-32
S-33
S-34    !  CODE SEGMENT 2         CAPTURE AFFINITY
```

```
S-35
S-36    !                          Set up buffer for affinity of n threads
S-37
S-38       n = omp_get_num_procs()
S-39       allocate( character(len=BUFFER_STORE)::buffer(0:n-1) )
S-40
S-41    !                          Capture Affinity using Affinity Format set above.
S-42    !                          Use max reduction to check size of buffer areas
S-43       max_req_store = 0
S-44       !$omp parallel private(thrd_num,nchars) reduction(max:max_req_store)
S-45
S-46          if(omp_get_num_threads()>n) stop "ERROR: increase buffer lines"
S-47
S-48          thrd_num=omp_get_thread_num()
S-49          nchars=omp_capture_affinity(buffer(thrd_num),null)
S-50          if(nchars>max_req_store) max_req_store=nchars
S-51          !  ...
S-52
S-53       !$omp end parallel
S-54
S-55       do i = 0, n-1
S-56          print*, "thrd_num= ",i,"   affinity:", trim(buffer(i))
S-57       end do
S-58             !  For 4 threads with OMP_PLACES='{0,1},{2,3},{4,5},{6,7}'
S-59             !  Format:   host=%20H thrd_num=%0.4n binds_to=%A
S-60
S-61             !  affinity: host=hpc.cn567            thrd_num=0000 binds_to=0,1
S-62             !  affinity: host=hpc.cn567            thrd_num=0001 binds_to=2,3
S-63             !  affinity: host=hpc.cn567            thrd_num=0002 binds_to=4,5
S-64             !  affinity: host=hpc.cn567            thrd_num=0003 binds_to=6,7
S-65
S-66       if(max_req_store > BUFFER_STORE) then
S-67          print*,  "Caution: Affinity string truncated.  Increase"
S-68          print*,  "         BUFFER_STORE to ",max_req_store
S-69       endif
S-70
S-71       deallocate(buffer)
S-72    end program
```

Fortran

4.4 Affinity Query Functions

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 sub-partition of the original partition. Hence, the **omp_get_place_num_procs** query function returns the number of elements (here procs = cores) in the sub-partition 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.

C / C++

Example affinity_query.1.c (**omp_4.5**)

```
S-1   #include <stdio.h>
S-2   #include <omp.h>
S-3
S-4   void socket_init(int socket_num)
S-5   {
S-6      int n_procs;
S-7
S-8      n_procs = omp_get_place_num_procs(socket_num);
S-9      #pragma omp parallel num_threads(n_procs) proc_bind(close)
S-10     {
S-11        printf("Reporting in from socket num, thread num:  %d %d\n",
S-12                                  socket_num,omp_get_thread_num() );
S-13     }
```

```
}

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);
   }

   return 0;
}
```

C / C++

Fortran

Example affinity_query.1.f90 (`omp_4.5`)

```
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()
   !$omp end parallel
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)

```

```
S-24          socket_num = omp_get_place_num()
S-25          call socket_init(socket_num)
S-26
S-27       !$omp end parallel
S-28    end program
```

Fortran

This page intentionally left blank

5 Tasking

Tasking constructs provide units of work to a thread for execution. Worksharing constructs do this, too (e.g. **for**, **do**, **sections**, and **single** 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 an expression that evaluates to *false* results in an *undeferred* task, which instructs the runtime to suspend the generating task until the undeferred task completes its execution. 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.

5.1 `task` and `taskwait` Constructs

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

C / C++

Example tasking.1.c (`omp_3.0`)

```
S-1   struct node {
S-2     struct node *left;
S-3     struct node *right;
S-4   };
S-5
S-6   extern void process(struct node *);
S-7
S-8   void traverse( struct node *p )
S-9   {
S-10    if (p->left)
S-11  #pragma omp task   // p is firstprivate by default
S-12        traverse(p->left);
S-13    if (p->right)
S-14  #pragma omp task    // p is firstprivate by default
S-15        traverse(p->right);
S-16    process(p);
S-17  }
```

C / C++

Fortran

Example tasking.1.f90 (`omp_3.0`)

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

```
!$OMP END TASK
ENDIF
CALL process ( P )

END SUBROUTINE
```

Fortran

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

C / C++

Example tasking.2.c (**omp_3.0**)

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

C / C++

Fortran

Example tasking.2.f90 (`omp_3.0`)

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

Fortran

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

C / C++

Example tasking.3.c (`omp_3.0`)

```
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
```

```
S-17              {
S-18                 node * p = head;
S-19                 while (p) {
S-20                      #pragma omp task
S-21                       // p is firstprivate by default
S-22                             process(p);
S-23                       p = p->next;
S-24                    }
S-25              }
S-26         }
S-27    }
```

C / C++

Fortran

1 *Example tasking.3.f90* (`omp_3.0`)

```
S-1         MODULE LIST
S-2             TYPE NODE
S-3                INTEGER :: PAYLOAD
S-4                TYPE (NODE), POINTER :: NEXT
S-5             END TYPE NODE
S-6         CONTAINS
S-7
S-8             SUBROUTINE PROCESS(p)
S-9                TYPE (NODE), POINTER :: P
S-10                   ! do work here
S-11            END SUBROUTINE
S-12
S-13            SUBROUTINE INCREMENT_LIST_ITEMS (HEAD)
S-14
S-15               TYPE (NODE), POINTER :: HEAD
S-16               TYPE (NODE), POINTER :: P
S-17               !$OMP PARALLEL PRIVATE(P)
S-18                   !$OMP SINGLE
S-19                       P => HEAD
S-20                       DO
S-21                          !$OMP TASK
S-22                              ! P is firstprivate by default
S-23                              CALL PROCESS(P)
S-24                          !$OMP END TASK
S-25                          P => P%NEXT
S-26                          IF ( .NOT. ASSOCIATED (P) ) EXIT
S-27                       END DO
S-28                   !$OMP END SINGLE
S-29               !$OMP END PARALLEL
S-30
S-31            END SUBROUTINE
```

```
      END MODULE
```

Fortran

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

C / C++

Example tasking.4.c (**omp_3.0**)

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

C / C++

Fortran

Example tasking.4.f (**omp_3.0**)

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

Fortran

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

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

C / C++

Example tasking.5.c (**omp_3.0**)

```
S-1   #define LARGE_NUMBER 10000000
S-2   double item[LARGE_NUMBER];
S-3   extern void process(double);
S-4
S-5   int main()
S-6   {
S-7   #pragma omp parallel
S-8     {
S-9       #pragma omp single
S-10      {
S-11        int i;
S-12        for (i=0; i<LARGE_NUMBER; i++)
S-13              #pragma omp task    // i is firstprivate, item is shared
S-14                   process(item[i]);
S-15      }
S-16    }
S-17  }
```

C / C++

Fortran

Example tasking.5.f (**omp_3.0**)

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

```
!$omp end parallel

      end
```

Fortran

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

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

C / C++

Example tasking.6.c (**omp_3.0**)

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

C / C++

Fortran

Example tasking.6.f (`omp_3.0`)

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

Fortran

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

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

C / C++

Example tasking.7.c (`omp_3.0`)

```
S-1   int tp;
S-2   #pragma omp threadprivate(tp)
S-3   int var;
S-4   void work()
S-5   {
S-6   #pragma omp task
S-7       {
S-8           /* do work here */
S-9   #pragma omp task
S-10          {
S-11              tp = 1;
S-12              /* do work here */
S-13  #pragma omp task
S-14              {
S-15                  /* no modification of tp */
```

```
S-16                }
S-17                var = tp; //value of tp can be 1 or 2
S-18            }
S-19            tp = 2;
S-20        }
S-21    }
```

C / C++

Fortran

Example tasking.7.f (`omp_3.0`)

```
S-1         module example
S-2         integer tp
S-3     !$omp threadprivate(tp)
S-4         integer var
S-5         contains
S-6         subroutine work
S-7     !$omp task
S-8            ! do work here
S-9     !$omp task
S-10           tp = 1
S-11           ! do work here
S-12    !$omp task
S-13              ! no modification of tp
S-14    !$omp end task
S-15           var = tp    ! value of var can be 1 or 2
S-16    !$omp end task
S-17          tp = 2
S-18    !$omp end task
S-19        end subroutine
S-20        end module
```

Fortran

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

C / C++

Example tasking.8.c (`omp_3.0`)

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

C / C++

Fortran

Example tasking.8.f (`omp_3.0`)

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

```
!$omp end parallel
      end subroutine
```

Fortran

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

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

C / C++

Example tasking.9.c (**omp_3.0**)

```
void work()
{
   #pragma omp task
   { //Task 1
       #pragma omp task
       { //Task 2
            #pragma omp critical //Critical region 1
            {/*do work here */ }
       }
       #pragma omp critical //Critical Region 2
       {
           //Capture data for the following task
           #pragma omp task
           { /* do work here */ } //Task 3
       }
   }
}
```

C / C++

Fortran

Example tasking.9.f (omp_3.0)

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

Fortran

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

C / C++

Example tasking.10.c (omp_3.0)

```
S-1   #include <omp.h>
S-2   void work() {
S-3       omp_lock_t lock;
S-4       omp_init_lock(&lock);
S-5   #pragma omp parallel
S-6       {
S-7           int i;
S-8   #pragma omp for
S-9           for (i = 0; i < 100; i++) {
S-10  #pragma omp task
S-11              {
S-12          // lock is shared by default in the task
```

```
S-13            omp_set_lock(&lock);
S-14                    // Capture data for the following task
S-15    #pragma omp task
S-16            // Task Scheduling Point 1
S-17                    { /* do work here */ }
S-18                    omp_unset_lock(&lock);
S-19                }
S-20            }
S-21        }
S-22        omp_destroy_lock(&lock);
S-23    }
```

C / C++

Fortran

Example tasking.10.f90 (`omp_3.0`)

```
S-1         module example
S-2         include 'omp_lib.h'
S-3         integer (kind=omp_lock_kind) lock
S-4         integer i
S-5
S-6         contains
S-7
S-8         subroutine work
S-9         call omp_init_lock(lock)
S-10    !$omp parallel
S-11        !$omp do
S-12        do i=1,100
S-13           !$omp task
S-14                ! Outer task
S-15                call omp_set_lock(lock)     ! lock is shared by
S-16                                            ! default in the task
S-17                      ! Capture data for the following task
S-18                      !$omp task      ! Task Scheduling Point 1
S-19                                ! do work here
S-20                      !$omp end task
S-21                call omp_unset_lock(lock)
S-22           !$omp end task
S-23        end do
S-24    !$omp end parallel
S-25        call omp_destroy_lock(lock)
S-26        end subroutine
S-27
S-28        end module
```

Fortran

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

C / C++

Example tasking.11.c (**omp_3.1**)

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

C / C++

Fortran

Example tasking.11.f90 (**omp_3.1**)

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

Fortran

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

C / C++

Example tasking.12.c (**omp_3.1**)

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

C / C++

Fortran

Example tasking.12.f90 (**omp_3.1**)

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

Fortran

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

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

C / C++

Example tasking.13.c (`omp_3.1`)

```
S-1   #include <string.h>
S-2   #include <omp.h>
S-3   #define LIMIT  3 /* arbitrary limit on recursion depth */
S-4   void check_solution(char *);
S-5   void bin_search (int pos, int n, char *state)
S-6   {
S-7      if ( pos == n ) {
S-8         check_solution(state);
S-9         return;
S-10     }
S-11     #pragma omp task final( pos > LIMIT ) mergeable
S-12     {
S-13        char new_state[n];
S-14        if (!omp_in_final() ) {
S-15          memcpy(new_state, state, pos );
S-16          state = new_state;
S-17        }
S-18        state[pos] = 0;
S-19        bin_search(pos+1, n, state );
S-20     }
S-21     #pragma omp task final( pos > LIMIT ) mergeable
S-22     {
S-23        char new_state[n];
S-24        if (! omp_in_final() ) {
S-25          memcpy(new_state, state, pos );
S-26          state = new_state;
S-27        }
S-28        state[pos] = 1;
S-29        bin_search(pos+1, n, state );
S-30     }
S-31     #pragma omp taskwait
S-32  }
```

C / C++

Fortran

Example tasking.13.f90 (`omp_3.1`)

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

Fortran

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

C / C++

Example tasking.14.c (`omp_3.1`)

```
S-1   void bar(void);
S-2
S-3   void foo ( )
S-4   {
S-5      int i;
S-6      #pragma omp task if(0)  // This task is undeferred
S-7      {
S-8         #pragma omp task     // This task is a regular task
S-9         for (i = 0; i < 3; i++) {
S-10            #pragma omp task     // This task is a regular task
S-11            bar();
S-12        }
S-13     }
S-14     #pragma omp task final(1) // This task is a regular task
S-15     {
S-16        #pragma omp task  // This task is included
S-17        for (i = 0; i < 3; i++) {
S-18            #pragma omp task     // This task is also included
S-19            bar();
S-20        }
S-21     }
S-22  }
```

C / C++

Fortran

Example tasking.14.f90 (`omp_3.1`)

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

```
S-18      enddo
S-19    !$omp end task
S-20    !$omp end task
S-21    end subroutine
```

Fortran

5.2 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++

Example task_priority.1.c (**omp_4.5**)

```
S-1     void compute_array (float *node, int M);
S-2
S-3     void compute_matrix (float *array, int N, int M)
S-4     {
S-5        int i;
S-6        #pragma omp parallel private(i)
S-7        #pragma omp single
S-8        {
S-9           for (i=0;i<N; i++) {
S-10             #pragma omp task priority(i)
S-11             compute_array(&array[i*M], M);
S-12          }
S-13       }
S-14    }
```

C / C++

Fortran

Example task_priority.1.f90 (**omp_4.5**)

```
S-1   subroutine compute_matrix(matrix, M, N)
S-2      implicit none
S-3      integer :: M, N
S-4      real :: matrix(M, N)
S-5      integer :: i
S-6      interface
S-7         subroutine compute_array(node, M)
S-8         implicit none
S-9         integer :: M
S-10        real :: node(M)
S-11        end subroutine
S-12     end interface
S-13     !$omp parallel private(i)
S-14     !$omp single
S-15     do i=1,N
S-16        !$omp task priority(i)
S-17        call compute_array(matrix(:, i), M)
S-18        !$omp end task
S-19     enddo
S-20     !$omp end single
S-21     !$omp end parallel
S-22  end subroutine compute_matrix
```

Fortran

5.3 Task Dependences

5.3.1 Flow Dependence

This example shows a simple flow dependence using a **depend** clause on the **task** construct.

C / C++

Example task_dep.1.c (**omp_4.0**)

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

C / C++

Fortran

Example task_dep.1.f90 (**omp_4.0**)

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

Fortran

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.

5.3.2 Anti-dependence

This example shows an anti-dependence using the **depend** clause on the **task** construct.

C / C++

Example task_dep.2.c (**omp_4.0**)

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

C / C++

Fortran

Example task_dep.2.f90 (**omp_4.0**)

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

Fortran

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.

5.3.3 Output Dependence

This example shows an output dependence using the **depend** clause on the **task** construct.

C / C++

Example task_dep.3.c (**omp_4.0**)

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

C / C++

Fortran

Example task_dep.3.f90 (**omp_4.0**)

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

Fortran

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.

5.3.4 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.

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.

C / C++

Example task_dep.4.c (**omp_4.0**)

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

C / C++

Fortran

Example task_dep.4.f90 (**omp_4.0**)

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

```
      !$omp end task

      !$omp task shared(x) depend(in: x)
         print*, "x + 2 = ", x+2, "."
      !$omp end task

   !$omp end single
   !$omp end parallel
end program
```

Fortran

The following example illustrates the semantic difference between **inout** and **inoutset** dependence types. In Case 1, tasks generated at T1 inside the loop have dependences among themselves due to the **inout** dependence type and with task T2. As a result, these tasks are executed sequentially before the print statement from task T2. In Case 2, tasks generated at T3 inside the loop have no dependences among themselves from the **inoutset** dependence type, but have dependences with task T4. As a result, these tasks are executed concurrently before the print statement from task T4.

C / C++

Example task_dep.4b.c (**omp_5.1**)

```
#include <stdio.h>

extern int f(int i);

void task_dep(int N)
{
  int i, v, R;

  #pragma omp parallel private(i,v) shared(R)
  #pragma omp single
  {
    // CASE 1: tasks with inout dependence type.
    //         tasks are serialized here.
    R = 0;
    for ( i = 0; i < N; i++ ) {
      #pragma omp task depend(inout: R)    // T1
      {
        v = f(i);
        R += v;
      }
    }

    #pragma omp task depend(in: R)         // T2
      printf("result is %d\n", R);
    #pragma omp taskwait      // to avoid race with CASE 2
```

```
S-26
S-27        // CASE 2: tasks with inoutset dependence type.
S-28        //         tasks are executed concurrently.
S-29        R = 0;
S-30        for ( i = 0; i < N; i++ ) {
S-31          #pragma omp task depend(inoutset: R) // T3
S-32          {
S-33            v = f(i);
S-34            #pragma omp atomic
S-35            R += v;
S-36          }
S-37        }
S-38
S-39        #pragma omp task depend(in: R)          // T4
S-40          printf("result is %d\n", R);
S-41      }
S-42    }
```

C / C++

Fortran

1 *Example task_dep.4b.f90* (`omp_5.1`)

```
S-1     subroutine task_dep(N)
S-2       implicit none
S-3       integer :: N
S-4
S-5       integer :: i, v, R
S-6       integer, external :: f
S-7
S-8       !$omp parallel private(i,v) shared(R)
S-9       !$omp single
S-10        !! CASE 1: tasks with inout dependence type.
S-11        !!         tasks are serialized here.
S-12        R = 0
S-13        do i = 1, N
S-14          !$omp task depend(inout: R)     !! T1
S-15            v = f(i)
S-16            R = R + v
S-17          !$omp end task
S-18        end do
S-19
S-20        !$omp task depend(in: R)          !! T2
S-21          print *, "result is ", R
S-22        !$omp end task
S-23        !$omp taskwait      !! to avoid race with CASE 2
S-24
S-25        !! CASE 2: tasks with inoutset dependence type.
```

```
      !!         tasks are executed concurrently.
      R = 0
      do i = 1, N
        !$omp task depend(inoutset: R) !! T3
          v = f(i)
          !$omp atomic
          R = R + v
        !$omp end task
      end do

      !$omp task depend(in: R)         !! T4
        print *, "result is ", R
      !$omp end task

    !$omp end single
    !$omp end parallel
end subroutine
```

Fortran

5.3.5 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 / C++

Example task_dep.5.c (`omp_4.0`)

```
#define N 100
// Assume BS divides N perfectly
void matmul_depend(int BS, float A[N][N], float B[N][N],
                   float C[N][N])
{
   int i, j, k, ii, jj, kk;
   for (i = 0; i < N; i+=BS) {
      for (j = 0; j < N; j+=BS) {
         for (k = 0; k < N; k+=BS) {
// Note 1: i, j, k, A, B, C are firstprivate by default
// Note 2: A, B and C are just pointers
#pragma omp task private(ii, jj, kk) \
            depend ( in: A[i:BS][k:BS], B[k:BS][j:BS] ) \
            depend ( inout: C[i:BS][j:BS] )
            for (ii = i; ii < i+BS; ii++ )
               for (jj = j; jj < j+BS; jj++ )
                  for (kk = k; kk < k+BS; kk++ )
                     C[ii][jj] = C[ii][jj] + A[ii][kk] * B[kk][jj];
```

```
S-19            }
S-20         }
S-21      }
S-22   }
```

C / C++

Fortran

Example task_dep.5.f90 (`omp_4.0`)

```
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   !$omp task shared(A,B,C) private(ii,jj,kk) &
S-12   !$omp depend ( in: A(i:i+BM, k:k+BM), B(k:k+BM, j:j+BM) ) &
S-13   !$omp depend ( inout: C(i:i+BM, j:j+BM) )
S-14   !  I,J,K are firstprivate by default
S-15                do ii = i, i+BM
S-16                   do jj = j, j+BM
S-17                      do kk = k, k+BM
S-18                         C(jj,ii) = C(jj,ii) + A(kk,ii) * B(jj,kk)
S-19                      end do
S-20                   end do
S-21                end do
S-22   !$omp end task
S-23            end do
S-24         end do
S-25      end do
S-26   end subroutine
```

Fortran

5.3.6 `taskwait` with Dependences

In this subsection three examples illustrate how the **depend** clause can be applied to a **taskwait** construct to make the generating task wait for specific child tasks to complete. This is an OpenMP 5.0 feature. In the same manner that dependences can order executions among child tasks with **depend** clauses on **task** constructs, the generating task can be scheduled to wait on child tasks at a **taskwait** before it can proceed.

Note: Since the **depend** clause on a **taskwait** construct relaxes the default synchronization behavior (waiting for all children to finish), it is important to realize that child tasks that are not predecessor tasks, as determined by the **depend** clause of the **taskwait** construct, may be running concurrently while the generating task is executing after the taskwait.

In the first example the generating task waits at the **taskwait** construct for the completion of the first child task because a dependence on the first task is produced by *x* with an **in** dependence type within the **depend** clause of the **taskwait** construct. Immediately after the first **taskwait** construct it is safe to access the *x* variable by the generating task, as shown in the print statement. There is no completion restraint on the second child task. Hence, immediately after the first **taskwait** it is unsafe to access the *y* variable since the second child task may still be executing. The second **taskwait** ensures that the second child task has completed; hence it is safe to access the *y* variable in the following print statement.

C / C++

Example task_dep.6.c (**omp_5.0**)

```
S-1   #include<stdio.h>
S-2
S-3   void foo()
S-4   {
S-5       int x = 0, y = 2;
S-6
S-7       #pragma omp task depend(inout: x) shared(x)
S-8       x++;                                          // 1st child task
S-9
S-10      #pragma omp task shared(y)
S-11      y--;                                          // 2nd child task
S-12
S-13      #pragma omp taskwait depend(in: x)            // 1st taskwait
S-14
S-15      printf("x=%d\n",x);
S-16
S-17      // Second task may not be finished.
S-18      // Accessing y here will create a race condition.
S-19
S-20      #pragma omp taskwait                          // 2nd taskwait
S-21
S-22      printf("y=%d\n",y);
```

```
S-23    }
S-24
S-25    int main()
S-26    {
S-27        #pragma omp parallel
S-28        #pragma omp single
S-29        foo();
S-30
S-31        return 0;
S-32    }
```

C / C++

Fortran

1 *Example task_dep.6.f90* (`omp_5.0`)

```
S-1     subroutine foo()
S-2         implicit none
S-3         integer :: x, y
S-4
S-5         x = 0
S-6         y = 2
S-7
S-8         !$omp task depend(inout: x) shared(x)
S-9             x = x + 1                          !! 1st child task
S-10        !$omp end task
S-11
S-12        !$omp task shared(y)
S-13            y = y - 1                          !! 2nd child task
S-14        !$omp end task
S-15
S-16        !$omp taskwait depend(in: x)           !! 1st taskwait
S-17
S-18        print*, "x=", x
S-19
S-20        !! Second task may not be finished.
S-21        !! Accessing y here will create a race condition.
S-22
S-23        !$omp taskwait                         !! 2nd taskwait
S-24
S-25        print*, "y=", y
S-26
S-27    end subroutine foo
S-28
S-29    program p
S-30        implicit none
S-31        !$omp parallel
S-32        !$omp single
```

```
      call foo()
    !$omp end single
    !$omp end parallel
end program p
```

Fortran

In this example the first two tasks are serialized, because a dependence on the first child is produced by *x* with the **in** dependence type in the **depend** clause of the second task. However, the generating task at the first **taskwait** waits only on the first child task to complete, because a dependence on only the first child task is produced by *x* with an **in** dependence type within the **depend** clause of the **taskwait** construct. The second **taskwait** (without a **depend** clause) is included to guarantee completion of the second task before *y* is accessed. (While unnecessary, the **depend(inout:** *y***)** clause on the 2nd child task is included to illustrate how the child task dependences can be completely annotated in a data-flow model.)

C / C++

Example task_dep.7.c (**omp_5.0**)

```
#include<stdio.h>

void foo()
{
    int x = 0, y = 2;

    #pragma omp task depend(inout: x) shared(x)
    x++;                                              // 1st child task

    #pragma omp task depend(in: x) depend(inout: y) shared(x, y)
    y -= x;                                           // 2nd child task

    #pragma omp taskwait depend(in: x)                // 1st taskwait

    printf("x=%d\n",x);

    // Second task may not be finished.
    // Accessing y here would create a race condition.

    #pragma omp taskwait                              // 2nd taskwait

    printf("y=%d\n",y);

}

int main()
{
    #pragma omp parallel
    #pragma omp single
```

```
S-30        foo();
S-31
S-32        return 0;
S-33    }
```

C / C++

Fortran

Example task_dep.7.f90 (`omp_5.0`)

```
 S-1    subroutine foo()
 S-2    implicit none
 S-3    integer :: x, y
 S-4
 S-5        x = 0
 S-6        y = 2
 S-7
 S-8        !$omp task depend(inout: x) shared(x)
 S-9            x = x + 1                          !! 1st child task
S-10        !$omp end task
S-11
S-12        !$omp task depend(in: x) depend(inout: y) shared(x, y)
S-13            y = y - x                          !! 2nd child task
S-14        !$omp end task
S-15
S-16        !$omp taskwait depend(in: x)           !! 1st taskwait
S-17
S-18        print*, "x=", x
S-19
S-20        !! Second task may not be finished.
S-21        !! Accessing y here would create a race condition.
S-22
S-23        !$omp taskwait                         !! 2nd taskwait
S-24
S-25        print*, "y=", y
S-26
S-27    end subroutine foo
S-28
S-29    program p
S-30    implicit none
S-31        !$omp parallel
S-32        !$omp single
S-33           call foo()
S-34        !$omp end single
S-35        !$omp end parallel
S-36    end program p
```

Fortran

This example is similar to the previous one, except the generating task is directed to also wait for completion of the second task.

The **depend** clause of the **taskwait** construct now includes an **in** dependence type for *y*. Hence the generating task must now wait on completion of any child task having *y* with an **out** (here **inout**) dependence type in its **depend** clause. So, the **depend** clause of the **taskwait** construct now constrains the second task to complete at the taskwait, too. (This change makes the second taskwait of the previous example unnecessary– it has been removed in this example.)

Note: While a **taskwait** construct ensures that all child tasks have completed; a depend clause on a **taskwait** construct only waits for specific child tasks (prescribed by the dependence type and list items in the **taskwait**'s **depend** clause). This and the previous example illustrate the need to carefully determine the dependence type of variables in the **depend** clause of the **taskwait** construct. when selecting child tasks that the generating task must wait on, so that its execution after the taskwait does not produce race conditions on variables accessed by non-completed child tasks.

C / C++

Example task_dep.8.c (**omp_5.0**)

```
S-1   #include<stdio.h>
S-2
S-3   void foo()
S-4   {
S-5       int x = 0, y = 2;
S-6
S-7       #pragma omp task depend(inout: x) shared(x)
S-8       x++;                                              // 1st child task
S-9
S-10      #pragma omp task depend(in: x) depend(inout: y) shared(x, y)
S-11      y -= x;                                           // 2st child task
S-12
S-13      #pragma omp taskwait depend(in: x,y)
S-14
S-15      printf("x=%d\n",x);
S-16      printf("y=%d\n",y);
S-17
S-18  }
S-19
S-20  int main()
S-21  {
S-22      #pragma omp parallel
S-23      #pragma omp single
S-24      foo();
S-25
S-26      return 0;
S-27  }
```

C / C++

Fortran

Example task_dep.8.f90 (`omp_5.0`)

```
S-1   subroutine foo()
S-2   implicit nonE
S-3   integer :: x, y
S-4
S-5       x = 0
S-6       y = 2
S-7
S-8       !$omp task depend(inout: x) shared(x)
S-9           x = x + 1                          !! 1st child task
S-10      !$omp end task
S-11
S-12      !$omp task depend(in: x) depend(inout: y) shared(x, y)
S-13          y = y - x                          !! 2nd child task
S-14      !$omp end task
S-15
S-16      !$omp taskwait depend(in: x,y)
S-17
S-18      print*, "x=", x
S-19      print*, "y=", y
S-20
S-21  end subroutine foo
S-22
S-23  program p
S-24  implicit none
S-25      !$omp parallel
S-26      !$omp single
S-27         call foo()
S-28      !$omp end single
S-29      !$omp end parallel
S-30  end program p
```

Fortran

5.3.7 Mutually Exclusive Execution with Dependences

In this example we show a series of tasks, including mutually exclusive tasks, expressing dependences using the **depend** clause on the **task** construct.

The program will always print 6. Tasks T1, T2 and T3 will be scheduled first, in any order. Task T4 will be scheduled after tasks T1 and T2 are completed. T5 will be scheduled after tasks T1 and T3 are completed. Due to the **mutexinoutset** dependence type on *c*, T4 and T5 may be scheduled in any order with respect to each other, but not at the same time. Tasks T6 will be scheduled after both T4 and T5 are completed.

C / C++

Example task_dep.9.c (**omp_5.0**)

```
S-1   #include <stdio.h>
S-2   int main()
S-3   {
S-4      int a, b, c, d;
S-5      #pragma omp parallel
S-6      #pragma omp single
S-7      {
S-8         #pragma omp task depend(out: c)
S-9            c = 1;   /* Task T1 */
S-10        #pragma omp task depend(out: a)
S-11           a = 2;   /* Task T2 */
S-12        #pragma omp task depend(out: b)
S-13           b = 3;   /* Task T3 */
S-14        #pragma omp task depend(in: a) depend(mutexinoutset: c)
S-15           c += a;  /* Task T4 */
S-16        #pragma omp task depend(in: b) depend(mutexinoutset: c)
S-17           c += b;  /* Task T5 */
S-18        #pragma omp task depend(in: c)
S-19           d = c;   /* Task T6 */
S-20     }
S-21     printf("%d\n", d);
S-22     return 0;
S-23  }
```

C / C++

Fortran

Example task_dep.9.f90 (`omp_5.0`)

```
S-1   program example
S-2      integer :: a, b, c, d
S-3      !$omp parallel
S-4      !$omp single
S-5         !$omp task depend(out: c)
S-6         c = 1       ! Task T1
S-7         !$omp end task
S-8         !$omp task depend(out: a)
S-9         a = 2       ! Task T2
S-10        !$omp end task
S-11        !$omp task depend(out: b)
S-12        b = 3       ! Task T3
S-13        !$omp end task
S-14        !$omp task depend(in: a) depend(mutexinoutset: c)
S-15        c = c + a   ! Task T4
S-16        !$omp end task
S-17        !$omp task depend(in: b) depend(mutexinoutset: c)
S-18        c = c + b   ! Task T5
S-19        !$omp end task
S-20        !$omp task depend(in: c)
S-21        d = c       ! Task T6
S-22        !$omp end task
S-23     !$omp end single
S-24     !$omp end parallel
S-25     print *, d
S-26  end program
```

Fortran

The following example demonstrates a situation where the **`mutexinoutset`** dependence type is advantageous. If *`shortTaskB`* completes before *`longTaskA`*, the runtime can take advantage of this by scheduling *`longTaskBC`* before *`shortTaskAC`*.

C / C++

Example task_dep.10.c (`omp_5.0`)

```
S-1   extern int longTaskA(), shortTaskB();
S-2   extern int shortTaskAC(int,int), longTaskBC(int,int);
S-3   void foo (void)
S-4   {
S-5     int a, b, c;
S-6     c = 0;
S-7     #pragma omp parallel
S-8     #pragma omp single
S-9     {
```

```
S-10        #pragma omp task depend(out: a)
S-11           a = longTaskA();
S-12        #pragma omp task depend(out: b)
S-13           b = shortTaskB();
S-14        #pragma omp task depend(in: a) depend(mutexinoutset: c)
S-15           c = shortTaskAC(a,c);
S-16        #pragma omp task depend(in: b) depend(mutexinoutset: c)
S-17           c = longTaskBC(b,c);
S-18     }
S-19  }
```

C / C++

Fortran

1 *Example task_dep.10.f90* (`omp_5.0`)

```
 S-1  subroutine foo
 S-2     integer :: a,b,c
 S-3     c = 0
 S-4     !$omp parallel
 S-5     !$omp single
 S-6        !$omp task depend(out: a)
 S-7           a = longTaskA()
 S-8        !$omp end task
 S-9        !$omp task depend(out: b)
S-10           b = shortTaskB()
S-11        !$omp end task
S-12        !$omp task depend(in: a) depend(mutexinoutset: c)
S-13           c = shortTaskAC(a,c)
S-14        !$omp end task
S-15        !$omp task depend(in: b) depend(mutexinoutset: c)
S-16           c = longTaskBC(b,c)
S-17        !$omp end task
S-18     !$omp end single
S-19     !$omp end parallel
S-20  end subroutine foo
```

Fortran

5.3.8 Multidependences Using Iterators

The following example uses an iterator to define a dynamic number of dependences.

In the **single** construct of a parallel region a loop generates *n* tasks and each task has an **out** dependence specified through an element of the *v* array. This is followed by a single task that defines an **in** dependence on each element of the array. This is accomplished by using the **iterator** modifier in the **depend** clause, supporting a dynamic number of dependences (*n* here).

The task for the *print_all_elements* procedure is not executed until all dependences prescribed (or registered) by the iterator are fulfilled; that is, after all the tasks generated by the loop have completed.

Note, one cannot simply use an array section in the **depend** clause of the second task construct because this would violate the **depend** clause restriction:

"List items used in **depend** clauses of the same task or sibling tasks must indicate identical storage locations or disjoint storage locations".

In this case each of the loop tasks use a single disjoint (different storage) element in their **depend** clause; however, the array-section storage area prescribed in the commented directive is neither identical nor disjoint to the storage prescribed by the elements of the loop tasks. The iterator overcomes this restriction by effectively creating *n* disjoint storage areas.

C / C++

Example task_dep.11.c (**omp_5.0**)

```
S-1   #include<stdio.h>
S-2
S-3   void set_an_element(int *p, int val) {
S-4       *p = val;
S-5   }
S-6
S-7   void print_all_elements(int *v, int n) {
S-8       int i;
S-9       for (i = 0; i < n; ++i) {
S-10          printf("%d, ", v[i]);
S-11      }
S-12      printf("\n");
S-13  }
S-14
S-15  void parallel_computation(int n) {
S-16      int v[n];
S-17      #pragma omp parallel
S-18      #pragma omp single
S-19      {
S-20          int i;
```

```
S-21            for (i = 0; i < n; ++i)
S-22                #pragma omp task depend(out: v[i])
S-23                set_an_element(&v[i], i);
S-24
S-25            #pragma omp task depend(iterator(it = 0:n), in: v[it])
S-26          // The following violates array-section restriction:
S-27          // #pragma omp task depend(in: v[0:n])
S-28            print_all_elements(v, n);
S-29        }
S-30    }
```

C / C++

Fortran

1 *Example task_dep.11.f90* (`omp_5.0`)

```
 S-1    subroutine set_an_element(e, val)
 S-2        implicit none
 S-3        integer :: e, val
 S-4
 S-5        e = val
 S-6
 S-7    end subroutine
 S-8
 S-9    subroutine print_all_elements(v, n)
S-10        implicit none
S-11        integer :: n, v(n)
S-12
S-13        print *, v
S-14
S-15    end subroutine
S-16
S-17    subroutine parallel_computation(n)
S-18        implicit none
S-19        integer :: n
S-20        integer :: i, v(n)
S-21
S-22        !$omp parallel
S-23        !$omp single
S-24            do i=1, n
S-25                !$omp task depend(out: v(i))
S-26                     call set_an_element(v(i), i)
S-27                !$omp end task
S-28            enddo
S-29
S-30            !$omp task depend(iterator(it = 1:n), in: v(it))
S-31           !!$omp task depend(in: v(1:n)) Violates Array section restriction.
S-32                call print_all_elements(v, n)
```

```
S-33            !$omp end task
S-34
S-35        !$omp end single
S-36        !$omp end parallel
S-37    end subroutine
```

Fortran

5.3.9 Dependence for Undeferred Tasks

In the following example, we show that even if a task is undeferred as specified by an **if** clause that evaluates to `false`, task dependences are still honored.

The **depend** clauses of the first and second explicit tasks specify that the first task is completed before the second task.

The second explicit task has an **if** clause that evaluates to `false`. This means that the execution of the generating task (the implicit task of the **single** region) must be suspended until the second explicit task is completed. But, because of the dependence, the first explicit task must complete first, then the second explicit task can execute and complete, and only then the generating task can resume to the print statement. Thus, the program will always print `x = 2`.

C / C++

Example task_dep.12.c (**omp_4.0**)

```
S-1     #include <stdio.h>
S-2     int main ()
S-3     {
S-4       int x = 0;
S-5       #pragma omp parallel
S-6       #pragma omp single
S-7       {
S-8         /* first explicit task */
S-9         #pragma omp task shared(x) depend(out: x)
S-10          x = 1;
S-11
S-12        /* second explicit task */
S-13        #pragma omp task shared(x) depend(inout: x) if(0)
S-14          x = 2;
S-15
S-16        /* statement executed by parent implicit task
S-17           prints: x = 2 */
S-18        printf("x = %d\n", x);
S-19      }
S-20      return 0;
S-21    }
```

C / C++

Fortran

Example task_dep.12.f90 (`omp_4.0`)

```
program example
   integer :: x
   x = 0
   !$omp parallel
   !$omp single
     !... first explicit task
      !$omp task shared(x) depend(out: x)
         x = 1
      !$omp end task

     !... second explicit task
      !$omp task shared(x) depend(inout: x) if(.false.)
         x = 2
      !$omp end task

     !... statement executed by parent implicit task
     ! prints: x = 2
      print*, "x = ", x
   !$omp end single
   !$omp end parallel
end program
```

Fortran

In OpenMP 5.1 the **`omp_all_memory`** *reserved locator* was introduced to specify storage of all objects in memory. In the following example, it is used in Task 4 as a convenient way to specify that the locator (list item) denotes the storage of all objects (locations) in memory, and will therefore match the a and d locators of Task 2, Task 3 and Task 6. The dependences guarantee the ordered execution of Tasks 2 and 3 before 4, and Task 4 before Task 6. Since there are no dependences imposed on Task 1 and Task 5, they can be scheduled to execute at any time, with no ordering.

C / C++

Example task_dep.13.c (`omp_5.1`)

```
#include <stdio.h>

int main(){
   int a=1, d=1;

   #pragma omp parallel masked num_threads(5)
   {
      #pragma omp task                                  // Task 1
      { printf("T1\n"); }

```

```
S-11          #pragma omp task depend(out: a)                 // Task 2
S-12          { a++;
S-13            printf("T2 a=%i\n", a); }
S-14
S-15          #pragma omp task depend(out: d)                 // Task 3
S-16          { d++;
S-17            printf("T3 d=%i\n", d); }
S-18
S-19          #pragma omp task depend(inout: omp_all_memory) // Task 4
S-20          { a++; d++;
S-21            printf("T4 a=%i d=%i\n",   a,d);}
S-22
S-23          #pragma omp task                                // Task 5
S-24          { printf("T5\n"); }
S-25
S-26          #pragma omp task depend(in: a,d)                // Task 6
S-27          { a++; d++;
S-28            printf("T6 a=%i d=%i\n", a,d); }
S-29       }
S-30    }
S-31
S-32    /* OUTPUT: ordered {T2,T3 any order}, {T4}, {T6}
S-33        T2 a=2
S-34        T3 d=2
S-35        T4 a=3 d=3
S-36        T6 a=4 d=4
S-37
S-38      OUTPUT: unordered (can appear interspersed in ordered output)
S-39        T1
S-40        T5
S-41    */
```

C / C++

Fortran

1 *Example task_dep.13.f90* (`omp_5.1`)

```
S-1     program main
S-2       integer :: a=1, d=1
S-3
S-4       !$omp parallel masked  num_threads(5)
S-5
S-6         !$omp task                                  !! Task 1
S-7            write(*,'("T1")')
S-8         !$omp end task
S-9
S-10        !$omp task depend(out: a)                   !! Task 2
S-11           a=a+1
```

```
S-12          write(*,'("T2 a=",i1)') a
S-13       !$omp end task
S-14
S-15       !$omp task depend(out: d)                    !! Task 3
S-16          d=d+1
S-17          write(*,'("T3 d=",i1)') d
S-18       !$omp end task
S-19
S-20
S-21       !$omp task depend(inout: omp_all_memory)     !! Task 4
S-22          a=a+1; d=d+1
S-23          write(*,'("T4 a=",i1," d=",i1)') a, d
S-24       !$omp end task
S-25
S-26       !$omp task                                   !! Task 5
S-27          write(*,'("T5")')
S-28       !$omp end task
S-29
S-30       !$omp task depend(in: a,d)                   !! Task 6
S-31          a=a+1; d=d+1
S-32          write(*,'("T6 a=",i1," d=",i1)') a, d
S-33       !$omp end task
S-34
S-35     !$omp end parallel masked
S-36
S-37   end program
S-38
S-39   ! OUTPUT: ordered  {T2,T3 any order}, {T4}, {T6}
S-40   ! T2 a=2
S-41   ! T3 d=2
S-42   ! T4 a=3 d=3
S-43   ! T6 a=4 d=4
S-44   ! OUTPUT: unordered (can appear interspersed in ordered output)
S-45   ! T1
S-46   ! T5
```

Fortran

5.3.10 Transparent Task Dependences

In the following example, each iteration of the *h*-loop updates all elements of array *M* and task dependences are used to synchronize updates across different iterations of the loop. The code uses two levels of dependent tasks and assumes that *N_ROWS* is evenly divisible by *ROWS_PER_TASK*. The *h*-loop generates the first level of tasks, with the **depend** clause serializing their execution and each task calling *my_func*. A second level of tasks are generated by the *i*-loop in *my_func*.

However, the dependences for this second level of tasks are between tasks from different calls to *my_func*. In order to enforce these dependences, the first-level tasks are specified as transparent tasks with the **transparent(omp_impex)** clause. The **omp_impex** argument (which is the default if not explicitly specified) indicates that the task is both an exporting and importing task. For the purposes of dependence matching, an exporting task is one that makes its child tasks visible to its successors and an importing task is one that makes its preceding tasks (such as earlier sibling tasks) visible to its child tasks. As a result of the exposed dependences, the task generated in the i^{th} iteration of the h=h_0 instance of *my_func* is guaranteed to be ordered before the task generated in the i^{th} iteration of the h=h_1 instance of *my_func*, where $h_0 < h_1$.

C / C++

Example task_dep.14.c (**omp_6.0**)

```
#include <omp.h>

void my_func(int *M, int *v);

#define N_ROWS 20
#define N_COLS 20
#define NUM_VS 5
#define ROWS_PER_TASK 5
int M[N_ROWS*N_COLS], v[NUM_VS][N_COLS];

int main()
{
  for (int i = 0; i < N_ROWS*N_COLS; i++)
    M[i] = 1;

  for (int i = 0; i < NUM_VS; i++)
    for (int j = 0; j < N_COLS; j++)
      v[i][j] = 2;

  #pragma omp parallel single
  for (int h = 0;  h < NUM_VS; h++) {
    // Generate transparent task to establish dependences
    // between child tasks that don't share the same parent.
    #pragma omp task depend(inout:h) transparent(omp_impex)
    my_func(M, v[h]);
  }

  int check_value = 1;
  for (int i = 0; i < NUM_VS; i++)
    check_value *= 2;
  for (int i = 0; i < N_ROWS*N_COLS; i++)
    if (M[i] != check_value)
      return 1;
```

```
  return 0;
}

void my_func(int *M, int *v)
{
  for (int i = 0; i < N_ROWS; i += ROWS_PER_TASK) {
    // This task is dependency-ordered with respect to the corresponding
    // task in iteration i generated by other transparent tasks.
    #pragma omp task depend(inout:M[i*N_COLS])
    for (int j = 0; j < ROWS_PER_TASK; j++)
      for (int k = 0; k < N_COLS; k++)
        M[(i+j)*N_COLS + k] *= v[k];
  }
}
```

C / C++

Fortran

Example task_dep.14.f90 (`omp_6.0`)

```
program main
  use omp_lib
  integer, parameter :: N_ROWS = 20
  integer, parameter :: N_COLS = 20
  integer, parameter :: NUM_VS = 5
  integer, parameter :: ROWS_PER_TASK = 5
  integer :: h
  integer :: M(0:N_ROWS*N_COLS-1), v(0:N_COLS-1,0:NUM_VS-1)
  integer :: check_value

  M(:) = 1
  v(:,:) = 2

  !$omp parallel single
    do h = 0, NUM_VS-1
      ! Generate transparent task to establish dependences
      ! between child tasks that don't share the same parent.
      !$omp task depend(inout:h) transparent(omp_impex)
        call my_func(M, v(:,h))
      !$omp end task
    end do
  !$omp end parallel single

  check_value = 2**NUM_VS
  if (any(M /= check_value)) error stop

contains
  subroutine my_func(M, v)
```

```
S-29        integer :: M(0:), v(0:)
S-30        integer :: i,j,k
S-31
S-32        do i = 0, N_ROWS-1, ROWS_PER_TASK
S-33          ! This task is dependency-ordered with respect to the corresponding
S-34          ! task in iteration i generated by other transparent tasks.
S-35          !$omp task depend(inout:M(i*N_COLS))
S-36            do j = 0, ROWS_PER_TASK-1
S-37              do k = 0, N_COLS-1
S-38                M((i+j)*N_COLS+k) = M((i+j)*N_COLS+k) * v(k)
S-39              end do
S-40            end do
S-41          !$omp end task
S-42        end do
S-43      end subroutine
S-44    end program
```

Fortran

5.4 Task Detachment

The **detach** clause on a **task** construct provides a mechanism for an asynchronous routine to be called within a task block, and for the routine's callback to signal completion to the OpenMP runtime, through an event fulfillment, triggered by a call to the **omp_fulfill_event** routine. When a **detach** clause is used on a **task** construct, completion of the detachable task occurs when the task's structured block is completed AND an *allow-completion* event is fulfilled by a call to the **omp_fulfill_event** routine with the *event-handle* argument.

The first example illustrates the basic components used in a detachable task. The second example is a program that executes asynchronous IO, and illustrates methods that are also inherent in asynchronous messaging within MPI and asynchronous commands in streams within GPU codes. Interfaces to asynchronous operations found in IO, MPI and GPU parallel computing platforms and their programming models are not standardized.

———————-

The first example creates a detachable task that executes the asynchronous *async_work* routine, passing the **omp_fulfill_event** function and the (firstprivate) event handle to the function. Here, the OpenMP **omp_fulfill_event** procedure is the "callback" function to be executed at the end of the *async_work* function's asynchronous operations, with the associated data, *event*.

C / C++

Example task_detach.1.c (**omp_5.0**)

```
S-1   #include <omp.h>
S-2
S-3   void async_work(void (*)(void*), void*);
S-4   void work();
S-5
S-6   int main() {
S-7     int async=1;
S-8     #pragma omp parallel
S-9     #pragma omp masked
S-10    {
S-11
S-12      omp_event_handle_t event;
S-13      #pragma omp task detach(event)
S-14      {
S-15        if(async) {
S-16          async_work( (void (*)(void*)) omp_fulfill_event, (void*) event );
S-17        } else {
S-18          work();
S-19          omp_fulfill_event(event);
S-20        }
S-21      }
S-22                    // Other work
```

```
S-23        #pragma omp taskwait
S-24      }
S-25      return 0;
S-26    }
```

C / C++

Fortran

Example task_detach.1.f90 (`omp_5.0`)

```
S-1     program main
S-2       use omp_lib
S-3       implicit none
S-4
S-5       external :: async_work, work
S-6
S-7       logical :: async=.true.
S-8       integer(omp_event_handle_kind) :: event
S-9
S-10      !$omp parallel
S-11      !$omp masked
S-12
S-13        !$omp task detach(event)
S-14
S-15          if(async) then
S-16            call async_work(omp_fulfill_event, event)
S-17          else
S-18            call work()
S-19            call omp_fulfill_event(event)
S-20          endif
S-21
S-22        !$omp end task
S-23                      !! Other work
S-24
S-25        !$omp taskwait
S-26
S-27      !$omp end masked
S-28      !$omp end parallel
S-29
S-30    end program
```

Fortran

In the following example, text data is written asynchronously to the file *async_data*, using POSIX asynchronous IO (aio). An aio "control block", *cb*, is set up to send a signal when IO is complete, and the *sigaction* function registers the signal action, a callback to *callback_aioSigHandler*.

The first task (Task1) starts the asynchronous IO and runs as a detachable task. The second and third tasks (Task2 and Task3) perform synchronous IO to stdout with print statements. The difference between the two types of tasks is that the thread for Task1 is freed for other execution within the **parallel** region, while the threads for Task2 and Task3 wait on the (synchronous) IO to complete, and cannot perform other work while the operating system is performing the synchronous IO. The **if** clause ensures that the detachable task is launched and the call to the *aio_write* function returns before Task2 and Task3 are generated (while the async IO occurs in the "background" and eventually executes the callback function). The barrier at the end of the **parallel** region ensures that the detachable task has completed.

C / C++

Example task_detach.2.c (**omp_5.0**)

```
// use -lrt on loader line
#include  <stdio.h>
#include <unistd.h>
#include  <fcntl.h>
#include    <aio.h>
#include  <errno.h>
#include <signal.h>
#include <stdint.h>

#include    <omp.h>

#define IO_SIGNAL SIGUSR1        // Signal used to notify I/O completion

                                 // Handler for I/O completion signal
static void callback_aioSigHandler(int sig, siginfo_t *si,
                                   void *ucontext) {
   if (si->si_code == SI_ASYNCIO){
      printf( "OUT: I/O completion signal received.\n");
      omp_fulfill_event( (omp_event_handle_t)(uintptr_t)
                         (si->si_value.sival_ptr) );
   }
}

void work(int i){ printf("OUT: Executing work(%d)\n", i);}

int main() {
   // Write "Written Asynchronously." to file data, using POSIX
   // asynchronous IO. Error checking not included for clarity
   // and simplicity.
```

```
S-30
S-31        char      data[] = "Written Asynchronously.";
S-32
S-33        struct     aiocb cb;
S-34        struct sigaction sa;
S-35
S-36        omp_event_handle_t event;
S-37
S-38        int fd = open(  "async_data", O_CREAT|O_RDWR|O_TRUNC,0664);
S-39
S-40        // Setup async io (aio) control block (cb)
S-41        cb.aio_nbytes  = sizeof(data)-1;
S-42        cb.aio_fildes  = fd;
S-43        cb.aio_buf     = data;
S-44        cb.aio_reqprio = 0;
S-45        cb.aio_offset  = 0;
S-46        cb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
S-47        cb.aio_sigevent.sigev_signo  = IO_SIGNAL;
S-48
S-49        // Setup Signal Handler  Callback
S-50        sigemptyset(&sa.sa_mask);
S-51        sa.sa_flags = SA_RESTART | SA_SIGINFO;
S-52        sa.sa_sigaction = callback_aioSigHandler;   //callback
S-53        sigaction(IO_SIGNAL, &sa, NULL);
S-54
S-55        #pragma omp parallel num_threads(2)
S-56        #pragma omp masked
S-57        {
S-58
S-59           #pragma omp task detach(event) if(0)               // TASK1
S-60           {
S-61              cb.aio_sigevent.sigev_value.sival_ptr = (void *) event;
S-62              aio_write(&cb);
S-63           }
S-64
S-65           #pragma omp task                                   // TASK2
S-66              work(1);
S-67           #pragma omp task                                   // TASK3
S-68              work(2);
S-69
S-70        } // Parallel region barrier ensures completion of detachable task.
S-71
S-72        // Making sure the aio operation completed.
S-73        // With OpenMP detachable task the condition will always be false:
S-74        while(aio_error(&cb) == EINPROGRESS) {
S-75        printf(" INPROGRESS\n");} //Safeguard
S-76
```

```
S-77        close(fd);
S-78        return 0;
S-79     }
S-80     /* Any Order:
S-81     OUT: I/O completion signal received.
S-82     OUT: Executing work(1)
S-83     OUT: Executing work(2)
S-84     */
```

C / C++

5.5 `taskgroup` Construct

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

Initially, one task (the task executing the *`start_background_work()`* routine) 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 behavior of the **`taskwait`** construct, which would include the background tasks in the synchronization.

C / C++

Example taskgroup.1.c (**`omp_4.0`**)

```
S-1      extern void start_background_work(void);
S-2      extern void check_step(void);
S-3      extern void print_results(void);
S-4      struct tree_node
S-5      {
S-6         struct tree_node *left;
S-7         struct tree_node *right;
S-8      };
S-9      typedef struct tree_node* tree_type;
S-10     extern void init_tree(tree_type);
S-11     #define max_steps 100
S-12     void compute_something(tree_type tree)
S-13     {
S-14        // some computation
S-15     }
S-16     void compute_tree(tree_type tree)
S-17     {
S-18        if (tree->left)
S-19        {
S-20          #pragma omp task
```

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

C / C++

Fortran

1 *Example taskgroup.1.f90* (`omp_4.0`)

```
S-1     module tree_type_mod
S-2       integer, parameter :: max_steps=100
S-3       type tree_type
S-4         type(tree_type), pointer :: left, right
S-5       end type
S-6       contains
S-7         subroutine compute_something(tree)
S-8           type(tree_type), pointer :: tree
S-9     ! some computation
```

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

Fortran

5.6 `taskyield` Construct

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

C / C++

Example taskyield.1.c (`omp_3.1`)

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

C / C++

Fortran

Example taskyield.1.f90 (`omp_3.1`)

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

```
S-14          call omp_unset_lock(lock)
S-15        !$omp end task
S-16      end do
S-17
S-18  end subroutine
```

Fortran

5.7 `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 / C++

Example taskloop.1.c (**`omp_4.5`**)

```
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
S-11  #pragma omp taskloop private(j) grainsize(500) nogroup
S-12        for (i = 0; i < 10000; i++) { // can execute concurrently
S-13           for (j = 0; j < i; j++) {
S-14              loop_body(i, j);
S-15           }
S-16        }
S-17     }
S-18  }
```

C / C++

Fortran

Example taskloop.1.f90 (**omp_4.5**)

```
S-1   subroutine parallel_work
S-2      integer i
S-3      integer j
S-4   !$omp taskgroup
S-5
S-6   !$omp task
S-7      call long_running_task()
S-8   !$omp end task
S-9
S-10  !$omp taskloop private(j) grainsize(500) nogroup
S-11     do i=1,10000
S-12        do j=1,i
S-13           call loop_body(i, j)
S-14        end do
S-15     end do
S-16  !$omp end taskloop
S-17
S-18  !$omp end taskgroup
S-19  end subroutine
```

Fortran

Because a **taskloop** construct encloses a loop, it is often incorrectly perceived as a worksharing construct (when it is directly nested in a **parallel** region).

While a worksharing construct distributes the loop iterations across all threads in a team, the entire loop of a **taskloop** construct is executed by every thread of the team.

In the example below the first taskloop occurs closely nested within a **parallel** region and the entire loop is executed by each of the *T* threads; hence the reduction sum is executed *T***N* times.

The loop of the second taskloop is within a **single** region and is executed by a single thread so that only *N* reduction sums occur. (The other *N*-1 threads of the **parallel** region will participate in executing the tasks. This is the common use case for the **taskloop** construct.)

In the example, the code thus prints `x1 = 16384` (*T***N*) and `x2 = 1024` (*N*).

C / C++

Example taskloop.2.c (`omp_4.5`)

```
S-1   #include <stdio.h>
S-2
S-3   #define T 16
S-4   #define N 1024
S-5
S-6   void parallel_work() {
S-7       int x1 = 0, x2 = 0;
S-8
S-9       #pragma omp parallel shared(x1,x2) num_threads(T)
S-10      {
S-11          #pragma omp taskloop
S-12          for (int i = 0; i < N; ++i) {
S-13              #pragma omp atomic
S-14              x1++;           // executed T*N times
S-15          }
S-16
S-17          #pragma omp single
S-18          #pragma omp taskloop
S-19          for (int i = 0; i < N; ++i) {
S-20              #pragma omp atomic
S-21              x2++;           // executed N times
S-22          }
S-23      }
S-24
S-25      printf("x1 = %d, x2 = %d\n", x1, x2);
S-26  }
```

C / C++

Fortran

Example taskloop.2.f90 (`omp_4.5`)

```
S-1   subroutine parallel_work
S-2       implicit none
S-3       integer :: x1, x2
S-4       integer :: i
S-5       integer, parameter :: T = 16
S-6       integer, parameter :: N = 1024
S-7
S-8       x1 = 0
S-9       x2 = 0
S-10      !$omp parallel shared(x1,x2) num_threads(T)
S-11      !$omp taskloop
S-12      do i = 1,N
S-13          !$omp atomic
```

```
S-14            x1 = x1 + 1     ! executed T*N times
S-15            !$omp end atomic
S-16        end do
S-17        !$omp end taskloop
S-18
S-19        !$omp single
S-20        !$omp taskloop
S-21        do i = 1,N
S-22            !$omp atomic
S-23            x2 = x2 + 1     ! executed N times
S-24            !$omp end atomic
S-25        end do
S-26        !$omp end taskloop
S-27        !$omp end single
S-28        !$omp end parallel
S-29
S-30        write (*,'(A,I0,A,I0)') 'x1 = ', x1, ', x2 = ',x2
S-31    end subroutine
```

Fortran

5.8 Combined `parallel masked` and `taskloop` Constructs

Just as the **`for`** and **`do`** constructs were combined with the **`parallel`** construct for convenience, so too, the combined **`parallel masked taskloop`** and **`parallel masked taskloop simd`** constructs have been created for convenience when using the **`taskloop`** construct.

In the following example the first **`taskloop`** construct is enclosed by the usual **`parallel`** and **`masked`** constructs to form a team of threads, and a single task generator (primary thread) for the **`taskloop`** construct.

The same OpenMP operations for the first taskloop are accomplished by the second taskloop with the **`parallel masked taskloop`** combined construct. The third taskloop uses the combined **`parallel masked taskloop simd`** construct to accomplish the same behavior as closely nested **`parallel masked`**, and **`taskloop simd`** constructs.

As with any combined construct the clauses of the components may be used with appropriate restrictions. The combination of the **`parallel masked`** construct with the **`taskloop`** or **`taskloop simd`** construct produces no additional restrictions.

C / C++

Example parallel_masked_taskloop.1.c (`omp_5.1`)

```
#include <stdio.h>
#define N 100

int main()
{
   int i, a[N],b[N],c[N];

   for(int i=0; i<N; i++){ b[i]=i; c[i]=i; }  //init

   #pragma omp parallel
   #pragma omp masked
   #pragma omp taskloop                        // taskloop 1
   for(i=0;i<N;i++){ a[i] = b[i] + c[i]; }

   #pragma omp parallel masked taskloop        // taskloop 2
   for(i=0;i<N;i++){ b[i] = a[i] + c[i]; }

   #pragma omp parallel masked taskloop simd // taskloop 3
   for(i=0;i<N;i++){ c[i] = a[i] + b[i]; }

   printf(" %d %d\n",c[0],c[N-1]);  // 0 and 495
}
```

C / C++

Fortran

Example parallel_masked_taskloop.1.f90 (`omp_5.1`)

```
S-1   program main
S-2
S-3      integer, parameter :: N=100
S-4      integer :: i, a(N),b(N),c(N)
S-5
S-6      do i=1,N                              !! initialize
S-7         b(i) = i
S-8         c(i) = i
S-9      enddo
S-10
S-11     !$omp parallel
S-12     !$omp masked
S-13     !$omp taskloop                        !! taskloop 1
S-14     do i=1,N
S-15        a(i) = b(i) + c(i)
S-16     enddo
S-17     !$omp end taskloop
S-18     !$omp end masked
S-19     !$omp end parallel
S-20
S-21     !$omp parallel masked taskloop        !! taskloop 2
S-22     do i=1,N
S-23        b(i) = a(i) + c(i)
S-24     enddo
S-25     !$omp end parallel masked taskloop
S-26
S-27     !$omp parallel masked taskloop simd !! taskloop 3
S-28     do i=1,N
S-29        c(i) = a(i) + b(i)
S-30     enddo
S-31     !$omp end parallel masked taskloop simd
S-32
S-33     print*,c(1),c(N)  !! 5 and 500
S-34
S-35  end program
```

Fortran

5.9 Task Dependences for `taskloop` Construct

Dependences for tasks generated from a **`taskloop`** construct can be specified using the **`task_iteration`** directive nested in the beginning of the associated loop body.

In the following example, taskloop TL1 contains a **`task_iteration`** directive with the **`depend`** clauses that specify task dependences across loop iterations on variable *A* (*A[i]* → *A[i-1]*). The **`nogroup`** clause for the **`taskloop`** construct removes the implicit taskgroup for a taskloop so that dependences across taskloops and with other tasks can be specified. For taskloop TL2, the dependence (*A[i]* → *A[i-4]*) is specified for every 4 loop iterations as defined by the **`if`** clause that matches with the chunk size 4 specified in the **`grainsize`** clause for taskloop tasks. The dependences are generated only for those iterations where the **`if`** condition evaluates to *true*. For instance, the first task generated from TL2 will update elements *A[1:4]* with depend clauses **`depend(inout:`** *A[4]***`)`** and **`depend(in:`** *A[0]***`)`**. This ensures element *A[4]* (thus elements *A[1:3]*) will be available from TL1 before executing the task. The last task T3 will wait for the availability of *A[n-1]* (or *A(n)* in Fortran) before printing the result.

C / C++

Example taskloop_dep.1.c (**`omp_6.0`**)

```
S-1   #include <stdio.h>
S-2
S-3   void process_work_a(int n, float *A)
S-4   {
S-5      // Dependences for taskloop iterations and across taskloops
S-6
S-7      // TL1 taskloop
S-8      // nogroup removes the implicit taskgroup
S-9      #pragma omp taskloop nogroup
S-10     for (int i = 1; i < n; i++)
S-11     {
S-12        #pragma omp task_iteration depend(inout: A[i]) depend(in: A[i-1])
S-13        A[i] += A[i] * A[i-1];
S-14     }
S-15
S-16     // TL2 taskloop + grainsize
S-17     #pragma omp taskloop grainsize(strict: 4) nogroup
S-18     for (int i = 1; i < n; i++)
S-19     {
S-20        #pragma omp task_iteration depend(inout: A[i]) depend(in: A[i-4]) \
S-21                                  if ((i % 4) == 0 || i == n-1)
S-22        A[i] += A[i] * A[i-1];
S-23     }
S-24
S-25     // T3 other task
S-26     #pragma omp task depend(in: A[n-1])
```

```
S-27      printf("A[n-1] = %f\n",  A[n-1]);
S-28   }
```

C / C++

Fortran

Example taskloop_dep.1.f90 (`omp_6.0`)

```
S-1    subroutine process_work_a(n, A)
S-2       implicit none
S-3       integer :: n
S-4       real :: A(*)
S-5       integer :: i
S-6
S-7       ! Dependences for taskloop iterations and across taskloops
S-8
S-9       ! TL1 taskloop
S-10      ! nogroup removes the implicit taskgroup
S-11      !$omp taskloop nogroup
S-12      do i = 2, n
S-13         !$omp task_iteration depend(inout: A(i)) depend(in: A(i-1))
S-14         A(i) = A(i) + A(i) * A(i-1)
S-15      end do
S-16      !$omp end taskloop
S-17
S-18      ! TL2 taskloop + grainsize
S-19      !$omp taskloop grainsize(strict: 4) nogroup
S-20      do i = 2, n
S-21         !$omp task_iteration depend(inout: A(i)) depend(in: A(i-4)) &
S-22         !$omp&               if (mod(i, 4) == 1 .or. i == n)
S-23         A(i) = A(i) + A(i) * A(i-1)
S-24      end do
S-25      !$omp end taskloop
S-26
S-27      ! T3 other task
S-28      !$omp task depend(in: A(n))
S-29      print *, "A(n) =", A(n)
S-30      !$omp end task
S-31   end subroutine
```

Fortran

The following example shows the use of the **`task_iteration`** directive for specifying task dependences in a multi-dimensional loop nest from multiple loop iterations in taskloop TL4. Similar to the previous example, the **`nogroup`** clause removes the implicit taskgroup for the **`taskloop`** construct so that dependences with other tasks (T5 in this case) can be specified.

C / C++

Example taskloop_dep.2.c (`omp_6.0`)

```
S-1   #include <stdio.h>
S-2
S-3   void process_work_b(int n, float *B[n])
S-4   {
S-5      // Dependences for taskloop iterations in multi-dimensional loop nest
S-6
S-7      // TL4 taskloop + collapse
S-8      #pragma omp taskloop collapse(2) nogroup
S-9      for (int i = 1; i < n; i++)
S-10     {
S-11        for (int j = 1; j < n; j++)
S-12        {
S-13           #pragma omp task_iteration depend(inout: B[i][j]) \
S-14                                      depend(in: B[i-1][j], B[i][j-1])
S-15           B[i][j] += B[i][j] * B[i-1][j] * B[i][j-1];
S-16        }
S-17     }
S-18
S-19     // T5 other task
S-20     #pragma omp task depend(in: B[n-1][n-1])
S-21     printf("B[n-1][n-1] = %f\n", B[n-1][n-1]);
S-22  }
```

C / C++

Fortran

Example taskloop_dep.2.f90 (`omp_6.0`)

```
S-1   subroutine process_work_b(n, B)
S-2      implicit none
S-3      integer :: n
S-4      real :: B(n,*)
S-5      integer :: i, j
S-6
S-7      ! Dependences for taskloop iterations in multi-dimensional loop nest
S-8
S-9      ! TL4 taskloop + collapse
S-10     !$omp taskloop collapse(2) nogroup
S-11     do j = 2, n
S-12        do i = 2, n
S-13           !$omp task_iteration depend(inout: B(i,j))  &
S-14           !$omp&               depend(in: B(i-1,j), B(i,j-1))
S-15           B(i,j) = B(i,j) + B(i,j) * B(i-1,j) * B(i,j-1)
S-16        end do
S-17     end do
```

```
S-18        !$omp end taskloop
S-19
S-20        ! T5 other task
S-21        !$omp task depend(in: B(n,n))
S-22        print *, "B(n,n) =", B(n,n)
S-23        !$omp end task
S-24    end subroutine
```

Fortran

This page intentionally left blank

6 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 a 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.

6.1 `target` Construct

6.1.1 `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.

C / C++

Example target.1.c (**`omp_4.0`**)

```
S-1   extern void init(float*, float*, int);
S-2   extern void output(float*, int);
S-3   void vec_mult(int N)
S-4   {
S-5      int i;
S-6      float p[N], v1[N], v2[N];
S-7      init(v1, v2, N);
S-8      #pragma omp target
S-9      #pragma omp parallel for private(i)
S-10     for (i=0; i<N; i++)
S-11       p[i] = v1[i] * v2[i];
S-12     output(p, N);
S-13  }
```

C / C++

Fortran

Example target.1.f90 (**`omp_4.0`**)

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

Fortran

6.1.2 `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 (**`omp_4.0`**)

```
S-1   extern void init(float*, float*, int);
S-2   extern void output(float*, int);
S-3   void vec_mult(int N)
S-4   {
S-5      int i;
S-6      float p[N], v1[N], v2[N];
S-7      init(v1, v2, N);
S-8      #pragma omp target map(v1, v2, p)
S-9      #pragma omp parallel for
S-10     for (i=0; i<N; i++)
S-11       p[i] = v1[i] * v2[i];
S-12     output(p, N);
S-13  }
```

C / C++

Fortran

Example target.2.f90 (**`omp_4.0`**)

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

Fortran

6.1.3 `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 (**`omp_4.0`**)

```
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.

Fortran

Example target.3.f90 (**omp_4.0**)

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

Fortran

6.1.4 `map` Clause with Array Sections

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

C / C++

Example target.4.c (**omp_4.0**)

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

C / C++

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

Fortran

Example target.4.f90 (**omp_4.0**)

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

Fortran

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

Fortran

Example target.4b.f90 (**omp_4.0**)

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

```
S-14    end subroutine
S-15    end module
```

Fortran

6.1.5 `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.

C / C++

Example target.5.c (**`omp_4.0`**)

```
S-1     #define THRESHOLD1 1000000
S-2     #define THRESHOLD2 1000
S-3
S-4     extern void init(float*, float*, int);
S-5     extern void output(float*, int);
S-6
S-7     void vec_mult(float *p, float *v1, float *v2, int N)
S-8     {
S-9        int i;
S-10
S-11       init(v1, v2, N);
S-12
S-13       #pragma omp target if(N>THRESHOLD1) map(to: v1[0:N], v2[:N])\
S-14            map(from: p[0:N])
S-15       #pragma omp parallel for if(N>THRESHOLD2)
S-16       for (i=0; i<N; i++)
S-17         p[i] = v1[i] * v2[i];
S-18
S-19       output(p, N);
S-20    }
```

C / C++

Fortran

Example target.5.f90 (**omp_4.0**)

```
S-1   module params
S-2   integer,parameter :: THRESHOLD1=1000000, THRESHHOLD2=1000
S-3   end module
S-4
S-5   subroutine vec_mult(p, v1, v2, N)
S-6      use params
S-7      real    ::  p(N), v1(N), v2(N)
S-8      integer ::  i
S-9
S-10     call init(v1, v2, N)
S-11
S-12     !$omp target if(N>THRESHHOLD1) map(to: v1, v2 ) map(from: p)
S-13        !$omp parallel do if(N>THRESHOLD2)
S-14        do i=1,N
S-15           p(i) = v1(i) * v2(i)
S-16        end do
S-17     !$omp end target
S-18
S-19     call output(p, N)
S-20  end subroutine
```

Fortran

The following example is a modification of the above *target.5* code to show the combined **target** and **parallel** 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.

C / C++

Example target.6.c (**omp_4.5**)

```
S-1   #define THRESHOLD1 1000000
S-2   #define THRESHOLD2 1000
S-3
S-4   extern void init(float*, float*, int);
S-5   extern void output(float*, int);
S-6
S-7   void vec_mult(float *p, float *v1, float *v2, int N)
S-8   {
S-9      int i;
S-10
S-11     init(v1, v2, N);
```

```
S-12
S-13        #pragma omp target parallel for \
S-14             if(target: N>THRESHOLD1) if(parallel: N>THRESHOLD2) \
S-15             map(to: v1[0:N], v2[:N]) map(from: p[0:N])
S-16        for (i=0; i<N; i++)
S-17          p[i] = v1[i] * v2[i];
S-18
S-19        output(p, N);
S-20     }
```

C / C++

Fortran

1 *Example target.6.f90* (`omp_4.5`)

```
S-1     module params
S-2     integer,parameter :: THRESHOLD1=1000000, THRESHHOLD2=1000
S-3     end module
S-4
S-5     subroutine vec_mult(p, v1, v2, N)
S-6        use params
S-7        real    ::  p(N), v1(N), v2(N)
S-8        integer ::  i
S-9
S-10       call init(v1, v2, N)
S-11
S-12       !$omp target parallel do  &
S-13       !$omp&   if(target: N>THRESHHOLD1) if(parallel: N>THRESHOLD2) &
S-14       !$omp&   map(to: v1, v2 ) map(from: p)
S-15          do i=1,N
S-16             p(i) = v1(i) * v2(i)
S-17          end do
S-18       !$omp end target parallel do
S-19
S-20       call output(p, N)
S-21    end subroutine
```

Fortran

6.1.6 Target Reverse Offload

Beginning with OpenMP 5.0, implementations are allowed to offload back to the host (reverse offload).

In the example below the `error_handler` function is executed back on the host, if an erroneous value is detected in the `A` array on the device.

This is accomplished by specifying the *device-modifier* **ancestor** modifier, along with a device number of `1`, to indicate that the execution is to be performed on the immediate parent (*1st ancestor*)– the host.

The **requires** directive (another 5.0 feature) uses the **reverse_offload** clause to guarantee that the reverse offload is implemented.

Note that the **declare target** directive uses the **device_type** clause (another 5.0 feature) to specify that the `error_handler` function is compiled to execute on the *host* only. This ensures that no attempt will be made to create a device version of the function. This feature may be necessary if the function exists in another compile unit.

C / C++

Example target_reverse_offload.7.c (**omp_5.2**)

```
S-1   #include <stdio.h>
S-2   #include <stdlib.h>
S-3
S-4   #define N 100
S-5
S-6   #pragma omp requires reverse_offload
S-7
S-8   void error_handler(int wrong_value, int index)
S-9   {
S-10     printf(" Error in offload: A[%d]=%d\n", index,wrong_value);
S-11     printf("        Expecting: A[i ]=i\n");
S-12     exit(1);
S-13  // output:  Error in offload: A[99]=-1
S-14  //                 Expecting: A[i ]=i
S-15
S-16  }
S-17  #pragma omp declare target device_type(host) enter(error_handler)
S-18
S-19  int main()
S-20  {
S-21     int A[N];
S-22
S-23     for (int i=0; i<N; i++) A[i] = i;
S-24
S-25     A[N-1]=-1;
```

```
   #pragma omp target map(A)
   {
      for (int i=0; i<N; i++)
      {
         if (A[i] != i)
         {
            #pragma omp target device(ancestor: 1) map(always,to: A[i:1])
               error_handler(A[i], i);
         }
      }
   }
   return 0;
}
```

C / C++

Fortran

Example target_reverse_offload.7.f90 (`omp_5.0`)

```
subroutine error_handler(wrong_value, index)
  implicit none
  integer :: wrong_value,index
  !$omp requires reverse_offload
  !$omp declare target device_type(host)

   write( *,'("Error in offload: A(",i3,")=",i3)' ) index,wrong_value
   write( *,'("       Expecting: A(  i)=  i")'    )
   stop
     !!output: Error in offload: A( 99)= -1
     !!               Expecting: A(  i)=  i
end subroutine

program rev_off
  implicit none
  !$omp requires reverse_offload
  integer, parameter :: N=100
  integer            :: i
  integer            :: A(N) = (/ (i, i=1,100) /)

   A(N-1)=-1

   !$omp target map(A)
      do i=1,N
         if (A(i) /= i)  then
           !$omp target device(ancestor: 1) map(always,to: A(i))
               call error_handler(A(i), i)
           !$omp end target
```

```
S-29              endif
S-30           end do
S-31        !$omp end target
S-32
S-33     end program
```

Fortran

6.2 `defaultmap` Clause

The implicitly determined data-mapping and data-sharing attribute rules of variables referenced in a **target** construct can be changed by the **defaultmap** clause. As of OpenMP 5.0, the implicit behavior is specified as **alloc**, **to**, **from**, **tofrom**, **firstprivate**, **none**, **default** or **present**, and is optionally applied to a variable category specified as **scalar**, **aggregate**, **allocatable**, or **pointer**.

A referenced variable that is in a specified "category" is treated as having the specified implicit behavior. In C/C++, **scalar** refers to base-language scalar variables, except pointers. In Fortran it refers to a scalar variable, as defined by the base language, of intrinsic type but excluding the character type. The **aggregate** category refers to arrays and structures (which includes variables of any derived type and of character type for Fortran). Fortran has the additional category of **allocatable** for variables that have the allocatable attribute. The **pointer** category refers to pointers, which for Fortran are variables that have the pointer attribute.

In the example below, the first **target** construct uses **defaultmap** clauses to set data-mapping and possibly data-sharing attributes that reproduce the default rules for implicitly determined data-mapping and data-sharing attributes for variables in the construct. That is, if the **defaultmap** clauses were removed, the results would be identical. As of OpenMP 5.2 the same effect can now be achieved by **defaultmap(default)** with the **target** construct.

In the second **target** construct all implicit behavior is removed by specifying the **none** implicit behavior in the **defaultmap** clause. Hence, all variables that don't have predetermined attributes must be given an explicit data-mapping or data-sharing attribute. A scalar (*s*), an array (*A*) and a structure (*S* for the C/C++ example and *D* for the Fortran example) are explicitly mapped with the **tofrom** map type.

The third **target** construct shows another usual case for using the **defaultmap** clause. The default mapping for (non-pointer) scalar variables is specified. Here, the default implicit mapping for *s3* is **tofrom** as specified in the **defaultmap** clause, while *s1* and *s2* are instead explicitly treated as **firstprivate**.

In the fourth **target** construct all arrays and structures are given **firstprivate** implicit behavior by default with the use of the **aggregate** variable category. For the Fortran example, the **allocatable** category is used in a separate **defaultmap** clause to specify default **firstprivate** implicit behavior for referenced allocatable variables (in this case, *H*).

The fifth **target** construct shows a case for using the **defaultmap** clause with the **all** variable category which was introduced in OpenMP 5.2. The scalar variables *s1* and *s2* are mapped **to**. *s3* is only mapped **from** due to the explicit map specified.

C / C++

Example target_defaultmap.1.c (**omp_5.2**)

```
S-1   #include <stdlib.h>
S-2   #include <stdio.h>
S-3   #define N 2
S-4
S-5   int main(){
S-6     typedef struct S_struct { int s; int A[N]; } S_struct_t;
S-7
S-8     int         s;       //scalar int variable (scalar)
S-9     int         A[N];    //aggregate variable  (array)
S-10    S_struct_t  S;       //aggregate variable  (structure)
S-11    int         *ptr;    //scalar, pointer variable (pointer)
S-12
S-13    int         s1, s2, s3;
S-14
S-15  // Initialize everything to zero;
S-16     s=2; s1=s2=s3=0;
S-17     A[0]=0; A[1]=0;
S-18     S.s=0; S.A[0]=0; S.A[1]=0;
S-19
S-20  // Target Region 1
S-21                    // Uses defaultmap to set scalars, aggregates &
S-22                    // pointers to normal defaults.
S-23      #pragma omp target \
S-24              defaultmap(firstprivate: scalar)   /* may also be default */ \
S-25              defaultmap(tofrom:       aggregate)/* may also be default */ \
S-26              defaultmap(default:      pointer)  /* must be default */
S-27      {
S-28          s       = 3;            //SCALAR firstprivate, value not returned
S-29
S-30          A[0]    = 3;  A[1] = 3; //AGGREGATE array, default map tofrom
S-31
S-32                                  //AGGREGATE structure, default tofrom
S-33          S.s     = 2;
S-34          S.A[0]  = 2;  S.A[1] = 2;
S-35
S-36          ptr = &A[0];            //POINTER is private
S-37          ptr[0] = 2;   ptr[1] = 2;
S-38      }
S-39     if(s==2 && A[0]==2 && S.s==2 && S.A[0]==2)
S-40        printf(" PASSED 1 of 5\n");
```

```
S-41
S-42
S-43    // Target Region 2
S-44                        // no implicit mapping allowed.
S-45        #pragma omp target defaultmap(none) map(tofrom: s, A, S)
S-46        {
S-47            s     +=5;            // All variables must be explicitly mapped
S-48            A[0]  +=5; A[1]+=5;
S-49            S.s   +=5;
S-50            S.A[0]+=5; S.A[1]+=5;
S-51        }
S-52        if(s==7 && A[0]==7 && S.s==7 && S.A[0]==7)
S-53            printf(" PASSED 2 of 5\n");
S-54
S-55
S-56    // Target Region 3
S-57                // defaultmap & explicit data-sharing clause
S-58                // with variables in same category
S-59        s1=s2=s3=1;
S-60        #pragma omp target defaultmap(tofrom: scalar) firstprivate(s1,s2)
S-61        {
S-62            s1 += 5;          // firstprivate (s1 value not returned to host)
S-63            s2 += 5;          // firstprivate (s2 value not returned to host)
S-64            s3 += s1 + s2;    // mapped as tofrom
S-65        }
S-66        if(s1==1 && s2==1 && s3==13 ) printf(" PASSED 3 of 5\n");
S-67
S-68
S-69    // Target Region 4
S-70        A[0]=0; A[1]=0;
S-71        S.A[0]=0; S.A[1]=0;
S-72
S-73        // arrays and structure are firstprivate, and scalars are from
S-74        #pragma omp target defaultmap(firstprivate: aggregate) \
S-75                           map(from: s1, s2)
S-76        {
S-77            A[0]+=1; S.A[0]+=1; //Aggregate changes not returned to host
S-78            A[1]+=1; S.A[1]+=1; //Aggregate changes not returned to host
S-79            s1 = A[0]+S.A[0]; //s1 value returned to host
S-80            s2 = A[1]+S.A[1]; //s1 value returned to host
S-81        }
S-82        if( A[0]==0 && S.A[0]==0 && s1==2 ) printf(" PASSED 4 of 5\n");
S-83
S-84    // Target Region 5
S-85                // defaultmap using all variable category
S-86
S-87        s1=s2=s3=1;
```

```
S-88
S-89        #pragma omp target defaultmap(to: all) map(from: s3)
S-90        {
S-91            s1 += 5;         // mapped as to
S-92            s2 += 5;         // mapped as to
S-93            s3 = s1 + s2;    // mapped as from
S-94        }
S-95        if(s1==1 && s2==1 && s3==12 ) printf(" PASSED 5 of 5\n");
S-96
S-97    }
```

C / C++

Fortran

1 *Example target_defaultmap.1.f90* (**omp_5.2**)

```
 S-1    program defaultmap
 S-2      integer, parameter :: N=2
 S-3
 S-4      type DDT_sA
 S-5        integer  :: s
 S-6        integer  :: A(N)
 S-7      end type
 S-8
 S-9      integer             :: s,s1,s2,s3 !! SCALAR: variable (integer)
S-10      integer,target      :: A(N)       !! AGGREGATE: Array
S-11      type(DDT_sA)        :: D          !! AGGREGATE: Derived Data Type (D)
S-12      integer,allocatable :: H(:)       !! ALLOCATABLE: Heap allocated array
S-13      integer,pointer     :: ptrA(:)    !! POINTER: points to array
S-14
S-15      ! Assign values to scalar, array, allocatable, and pointers
S-16
S-17        s=2
S-18        s1=0;   s2=0;     s3=0
S-19        D%s=0;  D%A(1)=0; D%A(2)=0
S-20        A(1)=0; A(2)=0
S-21
S-22        allocate( H(2) )
S-23        H(1)=0; H(2)=0
S-24
S-25    !! Target Region 1
S-26                       !! Using defaultmap to set scalars, aggregates &
S-27                       !! pointers and allocatables to normal defaults.
S-28        !$omp target                                              &
S-29        !$omp&       defaultmap( firstprivate: scalar)      &
S-30        !$omp&       defaultmap( tofrom:       aggregate)   &
S-31        !$omp&       defaultmap( tofrom:       allocatable) &
S-32        !$omp&       defaultmap( default:      pointer)
```

```
S-33
S-34            s = 3                       !! SCALAR firstprivate, val not returned
S-35
S-36            A(1) = 3                    !! AGGREGATE array, default map tofrom
S-37            A(2) = 3
S-38
S-39            D%s = 2                     !! AGGR. Derived Type, default map tofrom
S-40            D%A(1) = 2;  D%A(2) = 2
S-41
S-42            H(1) = 2;    H(2) = 2       !! ALLOCATABLE, default map tofrom
S-43
S-44            ptrA=>A                     !! POINTER is private
S-45            ptrA(1) = 2; ptrA(2) = 2
S-46
S-47        !$omp end target
S-48
S-49        if(s==2 .and. A(1)==2 .and. D%s==2 .and. D%A(1)==2 .and. H(1) == 2) &
S-50           print*," PASSED 1 of 5"
S-51
S-52    !! Target Region 2
S-53                        !! no implicit mapping allowed
S-54        !$omp target defaultmap(none) map(tofrom: s, A, D)
S-55
S-56            s=s+5                   !! All variables must be explicitly mapped
S-57            A(1)=A(1)+5;       A(2)=A(2)+5
S-58            D%s=D%s+5
S-59            D%A(1)=D%A(1)+5; D%A(2)=D%A(2)+5
S-60
S-61        !$omp end target
S-62        if(s==7 .and. A(1)==7 .and. D%s==7 .and. D%A(1)==7) &
S-63            print*," PASSED 2 of 5"
S-64
S-65    !! Target Region 3
S-66                !! defaultmap & explicit data-sharing clause
S-67                !! with variables in same category
S-68        s1=1; s2=1; s3=1
S-69        !$omp target defaultmap(tofrom: scalar) firstprivate(s1,s2)
S-70
S-71            s1 = s1+5           !! firstprivate (s1 value not returned to host)
S-72            s2 = s2+5           !! firstprivate (s2 value not returned to host)
S-73            s3 = s3 +s1 + s2    !! mapped as tofrom
S-74
S-75        !$omp end target
S-76        if(s1==1 .and. s2==1 .and. s3==13) print*," PASSED 3 of 5"
S-77
S-78    !! Target Region 4
S-79        A(1)=0;   A(2)=0
```

```
S-80        D%A(1)=0; D%A(2)=0
S-81        H(1)=0;   H(2)=0
S-82                  !! non-allocated arrays & derived types are in AGGREGATE cat
S-83                  !! Allocatable arrays are in ALLOCATABLE category
S-84                  !! Scalars are explicitly mapped from
S-85        !$omp target defaultmap(firstprivate: aggregate )  &
S-86        !$omp&       defaultmap(firstprivate: allocatable) &
S-87        !$omp&       map(from: s1, s2)
S-88
S-89            A(1)=A(1)+1; D%A(1)=D%A(1)+1; H(1)=H(1)+1 !! changes to A, D%A, H
S-90            A(2)=A(2)+1; D%A(2)=D%A(2)+1; H(2)=H(2)+1 !! not returned to host
S-91            s1 = A(1)+D%A(1)+H(1)                     !! s1 returned to host
S-92            s2 = A(2)+D%A(2)+H(1)                     !! s2 returned to host
S-93
S-94        !$omp end target
S-95        if(A(1)==0 .and. D%A(1)==0 .and. H(1)==0 .and. s1==3) &
S-96           print*," PASSED 4 of 5"
S-97
S-98    !! Target Region 5
S-99                !! defaultmap & explicit data-sharing clause
S-100               !! with variables in same category
S-101       s1=1; s2=1; s3=1
S-102       !$omp target defaultmap(to: all) map(from: s3)
S-103
S-104           s1 = s1+5        !! mapped as to
S-105           s2 = s2+5        !! mapped as to
S-106           s3 = s1 + s2     !! mapped as from
S-107
S-108       !$omp end target
S-109       if(s1==1 .and. s2==1 .and. s3==12) print*," PASSED 5 of 5"
S-110
S-111       deallocate(H)
S-112
S-113   end program
```

Fortran

6.3 Pointer Mapping

Pointers that contain host addresses require that those addresses are translated to device addresses for them to be useful in the context of a device data environment. Broadly speaking, there are two scenarios where this is important.

The first scenario is where the pointer is mapped to the device data environment, such that references to the pointer inside a **target** region are to the corresponding pointer. Pointer *attachment* ensures that the corresponding pointer will contain a device address when all of the following conditions are true:

- the pointer is mapped by directive A to a device;
- a list item that uses the pointer as its base pointer (call it the *pointee*) is mapped, to the same device, by directive B, which may be the same as A;
- the effect of directive B is to create either the corresponding pointer or pointee in the device data environment of the device.

Given the above conditions, pointer attachment is initiated as a result of directive B and subsequent references to the pointee list item in a target region that use the pointer will access the corresponding pointee. The corresponding pointer remains in this *attached* state until it is removed from the device data environment.

The second scenario, which is only applicable for C/C++, is where the pointer is implicitly privatized inside a **target** construct when it appears as the base pointer to a list item on the construct and does not appear explicitly as a list item in a **map** clause, **is_device_ptr** clause, or data-sharing attribute clause. This scenario can be further split into two cases: the list item is a zero-length array section (e.g., `p[:0]`) or it is not.

If it is a zero-length array section, this will trigger a runtime check on entry to the **target** region for a previously mapped list item where the value of the pointer falls within the range of its base address and ending address. If such a match is found the private pointer is initialized to the device address corresponding to the value of the original pointer, and otherwise it is initialized to **NULL** (or retains its original value if the **unified_address** requirement is specified for that compilation unit).

If the list item (again, call it the *pointee*) is not a zero-length array section, the private pointer will be initialized such that references in the **target** region to the pointee list item that use the pointer will access the corresponding pointee.

The following example shows the basics of mapping pointers with and without associated storage on the host.

Storage for pointers `ptr1` and `ptr2` is created on the host. To map storage that is associated with a pointer on the host, the data can be explicitly mapped as an array section so that the compiler knows the amount of data to be assigned in the device (to the *corresponding* data storage area). On the **target** construct array sections are mapped; however, the pointer `ptr1` is mapped, while `ptr2`

is not. Since *ptr2* is not explicitly mapped, it is firstprivate. This creates a subtle difference in the way these pointers can be used.

As a firstprivate pointer, *ptr2* can be manipulated on the device; however, as an explicitly mapped pointer, *ptr1* becomes an *attached* pointer and cannot be manipulated. In both cases the host pointer is not updated with the device pointer address—as one would expect for distributed memory. The storage data on the host is updated from the corresponding device data at the end of the **target** region.

As a comparison, note that the *aray* array is automatically mapped, since the compiler knows the extent of the array.

The pointer *ptr3* is used inside the **target** construct, but it does not appear in a data-mapping or data-sharing clause. Nor is there a **defaultmap** clause on the construct to indicate what its implicit data-mapping or data-sharing attribute should be. For such a case, *ptr3* will be implicitly privatized within the construct and there will be a runtime check to see if the host memory to which it is pointing has corresponding memory in the device data environment. If this runtime check passes, the private *ptr3* would be initialized to point to the corresponding memory. But in this case the check does not pass and so it is initialized to null. Since *ptr3* is private, the value to which it is assigned in the **target** region is not returned into the original *ptr3* on the host.

C / C++

Example target_ptr_map.1.c (**omp_5.0**)

```
S-1   #include <stdio.h>
S-2   #include <stdlib.h>
S-3   #define N 100
S-4
S-5   int main()
S-6   {
S-7     int *ptr1;
S-8     int *ptr2;
S-9     int *ptr3;
S-10    int aray[N];
S-11
S-12    ptr1 = (int *)malloc(sizeof(int)*N);
S-13    ptr2 = (int *)malloc(sizeof(int)*N);
S-14
S-15    #pragma omp target map(ptr1, ptr1[:N]) map(ptr2[:N] )
S-16    {
S-17       for (int i=0; i<N; i++)
S-18       {
S-19           ptr1[i] = i;
S-20           ptr2[i] = i;
S-21           aray[i] = i;
S-22       }
S-23
```

```
S-24      //*(++ptr1) = 9;  //NOT ALLOWED since ptr1 is an attached pointer
S-25        *(++ptr2) = 9;  //    allowed since ptr2 is firstprivate
S-26
S-27        ptr3=(int *)malloc(sizeof(int)*N); // ptr3 is firstprivate
S-28                                           // ptr3 value not returned
S-29        for (int i=0; i<N; i++) ptr3[i] = 5;
S-30
S-31        for (int i=0; i<N; i++) ptr1[i] += ptr3[i];
S-32
S-33        free(ptr3);     // explicitly free allocated storage on device
S-34      }
S-35
S-36      printf(" %d %d\n",ptr1[1],ptr2[1]);
S-37      //        6  9
S-38
S-39      free(ptr1);
S-40      free(ptr2);
S-41      return 0;
S-42    }
```

C / C++

In the following example the global pointer *p* appears in a declare target directive. Hence, the pointer *p* will persist on the device throughout executions in all **target** regions.

The pointer is also used in an array section of a **map** clause on a **target** construct. When the pointer of storage associated with a declare target directive is mapped, as for the array section *p[:N]* in the **target** construct, the array section on the device is *attached* to the device pointer *p* on entry to the construct, and the value of the device pointer *p* becomes undefined on exit. (Of course, storage allocation for the array section on the device will occur before the pointer on the device is attached.)

C / C++

Example target_ptr_map.2.c (**omp_5.1**)

```
S-1     #include <stdio.h>
S-2     #include <stdlib.h>
S-3     #define N 100
S-4
S-5     #pragma omp begin declare target
S-6       int *p;
S-7       extern void use_arg_p(int *p, int n);
S-8       extern void use_global_p(     int n);
S-9     #pragma omp end declare target
S-10
S-11    int main()
S-12    {
S-13      int i;
```

```
S-14      p = (int *)malloc(sizeof(int)*N);
S-15
S-16      #pragma omp target map(p[:N])  // device p attached to array section
S-17      {
S-18        for (i=0; i<N; i++) p[i] = i;
S-19        use_arg_p(p, N);
S-20        use_global_p(N);
S-21      }                              // value of host p is preserved
S-22
S-23      printf(" %3.3d %3.3d\n", p[1], p[N-1]);
S-24             // 003   297   <- output
S-25
S-26      free(p);
S-27      return 0;
S-28    }
S-29
S-30    // A #pragma omp begin declare target is optional here
S-31    // because of prototype spec
S-32    void use_arg_p(int *q, int n)
S-33    {
S-34      int i;
S-35      for (i=0; i<n; i++)
S-36        q[i] *= 2;
S-37    }
S-38
S-39    void use_global_p(int n)
S-40    {
S-41      int i;
S-42      for (i=0; i<n; i++)
S-43        p[i] += i;   // valid since p is in declare target and called from
S-44                     // inside target region where p was attached to
S-45                     // valid memory
S-46    }
S-47    // A #pragma omp end declare target is optional here
S-48    // because of prototype spec
```

C / C++

The following two examples illustrate subtle differences in pointer attachment to device address because of the order of data mapping.

In example *target_ptr_map.3a* the global pointer *p1* points to array *x* and *p2* points to array *y* on the host. The array section *x[:N]* is mapped by the **target enter data** directive while array *y* is mapped on the **target** construct. Since the **begin declare target** directive is applied to the declaration of *p1*, *p1* is a treated like a mapped variable on the **target** construct and references to *p1* inside the construct will be to the corresponding *p1* that exists on the device. However, the corresponding *p1* will be undefined since there is no pointer attachment for it. Pointer attachment for *p1* would require that (1) *p1* (or an lvalue expression that refers to the same storage

as *p1*) appears as a base pointer to a list item in a **map** clause, and (2) the construct that has the **map** clause causes the list item to transition from *not mapped* to *mapped*. The conditions are clearly not satisfied for this example.

The problem for *p2* in this example is also subtle. It will be privatized inside the **target** construct, with a runtime check for whether the memory to which it is pointing has corresponding memory that is accessible on the device. If this check is successful, then the *p2* inside the construct would be appropriately initialized to point to that corresponding memory. Unfortunately, despite there being an implicit map of the array *y* (to which *p2* is pointing) on the construct, the order of this map relative to the initialization of *p2* is unspecified. Therefore, the initial value of *p2* will also be undefined.

Thus, referencing values via either *p1* or *p2* inside the **target** region would be invalid.

C / C++

Example target_ptr_map.3a.c (**omp_5.1**)

```
S-1   #define N 100
S-2
S-3   int x[N], y[N];
S-4   #pragma omp begin declare target
S-5   int *p1;
S-6   #pragma omp end declare target
S-7   int *p2;
S-8
S-9   int foo()
S-10  {
S-11    p1 = &x[0];
S-12    p2 = &y[0];
S-13
S-14    // Explicitly map array section x[:N]
S-15    #pragma omp target enter data map(x[:N])
S-16
S-17    #pragma omp target  // as if .. map(p1) map(p1[:0]) map(p2[:0]) map(y)
S-18    {
S-19      // Accessing the mapped arrays x,y is OK here.
S-20      x[0] = 1;
S-21      y[1] = 2;
S-22
S-23      // Pointer attachment for p1 does not occur here
S-24      //   because p1[:0] does not allocate a new array section and
S-25      //   array x is present on the target construct as it was mapped
S-26      //   before by the target enter data directive.
S-27      p1[0] = 3;      // accessing p1 is undefined
S-28
S-29      // The initial value of p2 in the target region is undefined
S-30      //   because map(y) may occur after map(p2[:0]).
```

```
S-31        p2[1] = 4;       // accessing p2 is undefined
S-32
S-33      }
S-34      return 0;
S-35    }
```

C / C++

In example *target_ptr_map.3b* the mapping orders for arrays *x* and *y* were rearranged to allow proper pointer attachments. On the **target** construct, the **map(***x***)** clause triggers pointer attachment for *p1* to the device address of *x*. Pointer *p2* is assigned the device address of the previously mapped array *y*. Referencing values via either *p1* or *p2* inside the **target** region is now valid.

C / C++

Example target_ptr_map.3b.c (**omp_5.1**)

```
S-1     #define N 100
S-2
S-3     int x[N], y[N];
S-4     #pragma omp begin declare target
S-5     int *p1;
S-6     #pragma omp end declare target
S-7     int *p2;
S-8
S-9     int foo()
S-10    {
S-11      p1 = &x[0];
S-12      p2 = &y[0];
S-13
S-14      // Explicitly map array section y[:N]
S-15      #pragma omp target enter data map(y[:N])
S-16
S-17      #pragma omp target map(x[:N]) map(p1[:N]) map(p2[:0])
S-18      {
S-19        // Accessing the mapped arrays x,y is OK here.
S-20        x[0] = 1;
S-21        y[1] = 2;
S-22
S-23        // Pointer attachment for p1 occurs here when array x is mapped
S-24        //   on the target construct (as p1 = &x[0] on the device)
S-25        p1[0] = 3;       // accessing p1 is OK
S-26
S-27        // p2 in the target region is initialized to &y[0]
S-28        p2[1] = 4;       // accessing p2 is OK
S-29      }
S-30
```

```
  return 0;
}
```

C / C++

In the following example, storage allocated on the host is not mapped in a **target** region if it is determined that the host memory is accessible from the device. On platforms that support host memory access from a target device, it may be more efficient to omit map clauses and avoid the potential memory allocation and data transfers that may result from the map. The **omp_target_is_accessible** API routine is used to determine if the host storage of size *buf_size* is accessible on the device, and a metadirective is used to select the directive variant (a **target** with/without a **map** clause).

The **omp_target_is_accessible** routine will return true if the storage indicated by the first and second arguments is accessible on the target device. In this case, the host pointer *ptr* may be directly dereferenced in the subsequent **target** region to access this storage, rather than mapping an array section based off the pointer. By explicitly specifying the host pointer in a **firstprivate** clause on the construct, its original value will be used directly in the **target** region. In OpenMP 5.1, removing the **firstprivate** clause will result in an implicit presence check of the storage to which *ptr* points, and since this storage is not mapped by the program, *ptr* will be **NULL**-initialized in the **target** region. In the OpenMP 5.2 Specification, a false presence check without the **firstprivate** clause will cause the pointer to retain its original value.

C / C++

Example target_ptr_map.4.c (**omp_5.2**)

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

void do_work(int *ptr, const int size);

int main()
{
   const int n = 1000;
   const int buf_size = sizeof(int) * n;
   const int dev = omp_get_default_device();

   int *ptr = (int *) malloc(buf_size); // possibly compiled on
                                        // Unified Shared Memory system
   const int accessible = omp_target_is_accessible(ptr, buf_size, dev);

   #pragma omp metadirective \
      when(user={condition(accessible)}: target firstprivate(ptr) ) \
      otherwise(                         target map(ptr[:n])      )
   {
      do_work(ptr, n);
```

```
S-22        }
S-23
S-24        free(ptr);
S-25        return 0;
S-26     }
```

C / C++

Similar to the previous example, the **omp_target_is_accessible** routine is used to discover if a deep copy is required for the platform. Here, the *deep_copy* map, defined in the **declare mapper** directive, is used if the host storage referenced by *s.ptr* (or *s%ptr* in Fortran) is not accessible from the device.

C / C++

Example target_ptr_map.5.c (**omp_5.2**)

```
S-1     #include <stdio.h>
S-2     #include <stdlib.h>
S-3     #include <omp.h>
S-4
S-5     typedef struct {
S-6        int *ptr;
S-7        int buf_size;
S-8     } T;
S-9
S-10    #pragma omp declare mapper(deep_copy: T s) map(s, s.ptr[:s.buf_size])
S-11
S-12    void do_work(int *ptr, const int size);
S-13
S-14    int main()
S-15    {
S-16       const int n = 1000;
S-17       const int buf_size = sizeof(int) * n;
S-18       T s = { 0, buf_size };
S-19       const int dev = omp_get_default_device();
S-20       s.ptr = (int *)malloc(buf_size);
S-21       const int accessible =
S-22          omp_target_is_accessible(s.ptr, s.buf_size, dev);
S-23
S-24       #pragma omp metadirective \
S-25          when(user={condition(accessible)}: target) \
S-26          otherwise(target map(mapper(deep_copy),tofrom:s) )
S-27       {
S-28          do_work(s.ptr, n);
S-29       }
S-30
S-31       free(s.ptr);
```

```
   return 0;
}
```

C / C++

Fortran

Example target_ptr_map.5.f90 (`omp_5.2`)

```
program main
   use omp_lib

   use, intrinsic :: iso_c_binding, only : c_loc, c_size_t, c_sizeof, c_int
   implicit none
   external :: do_work

   type T
     integer,pointer :: ptr(:)
     integer         :: buf_size
   end type

   !$omp declare mapper(deep_copy: T :: s) map(s, s%ptr(:s%buf_size))

   integer,parameter :: n = 1000
   integer(c_int)    :: dev, accessible
   integer(c_size_t) :: buf_size

   type(T) s

   allocate(s%ptr(n))

   buf_size = c_sizeof(s%ptr(1))*n
   dev = omp_get_default_device()

   accessible = omp_target_is_accessible(c_loc(s%ptr(1)), buf_size, dev)

   !$omp  begin metadirective                                  &
   !$omp&       when(user={condition(accessible)}: target) &
   !$omp&       otherwise( target map(mapper(deep_copy),tofrom:s) )

      call do_work(s, n)

   !$omp  end   metadirective

   deallocate(s%ptr)

end program
```

Fortran

6.4 Structure Mapping

In the example below, only structure elements *S.a*, *S.b* and *S.p* of the *S* structure appear in **map** clauses of a **target** construct. Only these components have corresponding variables and storage on the device. Hence, the large arrays, *S.buffera* and *S.bufferb*, and the *S.x* component have no storage on the device and cannot be accessed.

Also, since the pointer member *S.p* is used in an array section of a **map** clause, the array storage of the array section on the device, *S.p[:N]*, is *attached* to the pointer member *S.p* on the device. Explicitly mapping the pointer member *S.p* is optional in this case.

Note: The buffer arrays and the *x* variable have been grouped together, so that the components that will reside on the device are all together (without gaps). This allows the runtime to optimize the transfer and the storage footprint on the device.

C / C++

Example target_struct_map.1.c (**omp_5.1**)

```
S-1   #include <stdio.h>
S-2   #include <stdlib.h>
S-3   #define N 100
S-4   #define BAZILLION 2000000
S-5
S-6   struct foo {
S-7     char buffera[BAZILLION];
S-8     char bufferb[BAZILLION];
S-9     float x;
S-10    float a, b;
S-11    float *p;
S-12  };
S-13
S-14  #pragma omp begin declare target
S-15  void saxpyfun(struct foo *S)
S-16  {
S-17    int i;
S-18    for(i=0; i<N; i++)
S-19      S->p[i] = S->p[i]*S->a + S->b;
S-20  }
S-21  #pragma omp end declare target
S-22
S-23  int main()
S-24  {
S-25    struct foo S;
S-26    int i;
S-27
S-28    S.a = 2.0;
S-29    S.b = 4.0;
```

```
S-30      S.p = (float *)malloc(sizeof(float)*N);
S-31      for(i=0; i<N; i++) S.p[i] = i;
S-32
S-33      #pragma omp target map(alloc:S.p) map(S.p[:N]) map(to:S.a, S.b)
S-34      saxpyfun(&S);
S-35
S-36      printf(" %4.0f %4.0f\n", S.p[0], S.p[N-1]);
S-37            //     4  202  <- output
S-38
S-39      free(S.p);
S-40      return 0;
S-41    }
```

C / C++

The following example is a slight modification of the above example for a C++ class. In the member function *SAXPY::driver* the array section *p[:N]* is attached to the pointer member *p* on the device.

C++

Example target_struct_map.2.cpp (**omp_5.1**)

```
S-1     #include <cstdio>
S-2     #include <cstdlib>
S-3     #define N 100
S-4
S-5     class SAXPY {
S-6       private:
S-7        float a, b, *p;
S-8       public:
S-9        float buffer[N];
S-10
S-11       SAXPY(float arg_a, float arg_b){ a=arg_a; b=arg_b; }
S-12       void driver();
S-13       void saxpyfun(float *p);
S-14    };
S-15
S-16    #pragma omp begin declare target
S-17    void SAXPY::saxpyfun(float *q)
S-18    {
S-19      for(int i=0; i<N; i++)
S-20        buffer[i] = q[i]*a + b;
S-21    }
S-22    #pragma omp end declare target
S-23
S-24    void SAXPY::driver()
S-25    {
S-26      p = (float *) malloc(N*sizeof(float));
S-27      for(int i=0; i<N; i++) p[i]=i;
```

```
S-28
S-29      #pragma omp target map(alloc:p) map(to:p[:N]) map(to:a,b) \
S-30                  map(from:buffer[:N])   // attach(p) to device_malloc()
S-31      {
S-32        saxpyfun(p);
S-33      }
S-34
S-35      free(p);
S-36    }
S-37
S-38    int main()
S-39    {
S-40      SAXPY my_saxpy(2.0,4.0);
S-41
S-42      my_saxpy.driver();
S-43
S-44      printf(" %4.0f %4.0f\n", my_saxpy.buffer[0], my_saxpy.buffer[N-1]);
S-45             //   4   202     <- output
S-46
S-47      return 0;
S-48    }
```

C++

The next example shows two ways in which the structure may be *incorrectly* mapped.

In Case 1, the array section *S1.p[:N]* is first mapped in an enclosing **target data** construct, and the **target** construct then implicitly maps the structure *S1*. The initial map of the array section does not map the base pointer *S1.p* – it only maps the elements of the array section. Furthermore, the implicit map is not sufficient to ensure pointer attachment for the structure member *S1.p* (refer to the conditions for pointer attachment described in Section 6.3). Consequentially, the dereference operation *S1.p[i]* in the call to *saxpyfun* will probably fail because *S1.p* contains a host address.

In Case 2, again an array section is mapped on an enclosing **target data** construct. This time, the nested **target** construct explicitly maps *S2.p*, *S2.a*, and *S2.b*. But as in Case 1, this does not satisfy the conditions for pointer attachment since the construct must map a list item for which *S2.p* is a base pointer, and it must do so when the *S2.p* is already present on the device or will be created on the device as a result of the same construct.

C / C++

Example target_struct_map.3.c (**omp_5.1**)

```
S-1     #include <stdio.h>
S-2     #include <stdlib.h>
S-3     #define N 100
S-4     #define BAZILLION 2000000
S-5
```

```
S-6     struct foo {
S-7       char buffera[BAZILLION];
S-8       char bufferb[BAZILLION];
S-9       float x;
S-10      float a, b;
S-11      float *p;
S-12    };
S-13
S-14    #pragma omp begin declare target
S-15    void saxpyfun(struct foo *S)
S-16    {
S-17      int i;
S-18      for(i=0; i<N; i++)
S-19        S->p[i] = S->p[i] * S->a + S->b; // S->p[i] invalid
S-20    }
S-21    #pragma omp end declare target
S-22
S-23    int main()
S-24    {
S-25      struct foo S1, S2;
S-26      int i;
S-27
S-28      // Case 1
S-29
S-30      S1.a = 2.0;
S-31      S1.b = 4.0;
S-32      S1.p = (float *)malloc(sizeof(float)*N);
S-33      for(i=0; i<N; i++) S1.p[i] = i;
S-34
S-35      // No pointer attachment for S1.p here
S-36      #pragma omp target data map(S1.p[:N])
S-37      #pragma omp target // implicit map of S1
S-38      saxpyfun(&S1);
S-39
S-40      // Case 2
S-41
S-42      S2.a = 2.0;
S-43      S2.b = 4.0;
S-44      S2.p = (float *)malloc(sizeof(float)*N);
S-45      for(i=0; i<N; i++) S2.p[i] = i;
S-46
S-47      // No pointer attachment for S2.p here either
S-48      #pragma omp target data map(S2.p[:N])
S-49      #pragma omp target map(S2.p, S2.a, S2.b) // implicit map of S2
S-50      saxpyfun(&S2);
S-51
S-52      // These print statement may not execute because the
```

```
S-53      // above code is invalid
S-54      printf(" %4.0f %4.0f\n", S1.p[0], S1.p[N-1]);
S-55      printf(" %4.0f %4.0f\n", S2.p[0], S2.p[N-1]);
S-56
S-57      free(S1.p);
S-58      free(S2.p);
S-59      return 0;
S-60    }
```

C / C++

The following example correctly implements pointer attachment cases that involve implicit structure maps.

In Case 1, members *p*, *a*, and *b* of the structure *S1* are explicitly mapped by the **target data** construct, to avoid mapping parts of *S1* that aren't required on the device. The mapped *S1.p* is attached to the array section *S1.p[:N]*, and remains attached while it exists on the device (for the duration of **target data** region). Due to the *S1* reference inside the nested **target** construct, the construct implicitly maps *S1* so that the reference refers to the corresponding storage created by the enclosing **target data** region. Note that only the members *a*, *b*, and *p* may be accessed from this storage.

In Case 2, only the storage for the array section *S2.p[:N]* is mapped by the **target data** construct. The nested **target** construct explicitly maps *S2.a* and *S2.b* and explicitly maps an array section for which *S2.p* is a base pointer. This satisfies the conditions for *S2.p* becoming an attached pointer. The array section in this case is zero-length, but the effect would be the same if the length was a positive integer less than or equal to *N*. There is also an implicit map of the containing structure *S2*, again due to the reference to *S2* inside the construct. The effect of this implicit map permits access only to members *a*, *b*, and *p*, as for Case 1.

In Case 3, there is no **target data** construct. The **target** construct explicitly maps *S3.a* and *S3.b* and explicitly maps an array section for which *S3.p* is a base pointer. Again, there is an implicit map of the structure referenced in the construct, *S3*. This implicit map also causes *S3.p* to be implicitly mapped, because no other part of *S3* is present prior to the construct being encountered. The result is an attached pointer *S3.p* on the device. As for Cases 1 and 2, this implicit map only ensures that storage for the members *a*, *b*, and *p* are accessible within the corresponding *S3* that is created on the device.

C / C++

Example target_struct_map.4.c (**omp_5.1**)

```
S-1     #include <stdio.h>
S-2     #include <stdlib.h>
S-3     #define N 100
S-4     #define BAZILLION 2000000
S-5
S-6     struct foo {
S-7       char buffera[BAZILLION];
```

```
  char bufferb[BAZILLION];
  float x;
  float a, b;
  float *p;
};

#pragma omp begin declare target
void saxpyfun(struct foo *S)
{
  int i;
  for(i=0; i<N; i++)
    S->p[i] = S->p[i]*S->a + S->b;
}
#pragma omp end declare target

int main()
{
  struct foo S1, S2, S3;
  int i;

  // Case 1

  S1.a = 2.0;
  S1.b = 4.0;
  S1.p = (float *)malloc(sizeof(float)*N);
  for(i=0; i<N; i++) S1.p[i] = i;

  // The target data construct results in pointer attachment for S1.p.
  // Explicitly mapping S1.p, S1.a, and S1.b rather than S1 avoids
  // mapping the entire structure (including members buffera, bufferb,
  // and x).
  #pragma omp target data map(S1.p[:N],S1.p,S1.a,S1.b)
  #pragma omp target //implicit map of S1
  saxpyfun(&S1);

  // Case 2

  S2.a = 2.0;
  S2.b = 4.0;
  S2.p = (float *)malloc(sizeof(float)*N);
  for(i=0; i<N; i++) S2.p[i] = i;

  // The target construct results in pointer attachment for S2.p.
  #pragma omp target data map(S2.p[:N])
  #pragma omp target map(S2.p[:0], S2.a, S2.b) // implicit map of S2
  saxpyfun(&S2);

```

```
S-55      // Case 3
S-56
S-57      S3.a = 2.0;
S-58      S3.b = 4.0;
S-59      S3.p = (float *)malloc(sizeof(float)*N);
S-60      for(i=0; i<N; i++) S3.p[i] = i;
S-61
S-62      // The target construct results in pointer attachment for S3.p.
S-63      // Note that S3.p is implicitly mapped due to the implicit map of S3
S-64      // (but corresponding storage is NOT created for members buffera,
S-65      // bufferb, and x).
S-66      #pragma omp target map(S3.p[:N], S3.a, S3.b)  // implicit map of S3
S-67      saxpyfun(&S3);
S-68
S-69      printf(" %4.0f %4.0f\n", S1.p[0], S1.p[N-1]);  //OUT1 4 202
S-70      printf(" %4.0f %4.0f\n", S2.p[0], S2.p[N-1]);  //OUT2 4 202
S-71      printf(" %4.0f %4.0f\n", S3.p[0], S3.p[N-1]);  //OUT3 4 202
S-72
S-73      free(S1.p);
S-74      free(S2.p);
S-75      free(S3.p);
S-76      return 0;
S-77   }
```

C / C++

Fortran

6.5 Fortran Allocatable Array Mapping

The following examples illustrate the use of Fortran allocatable arrays in **target** regions.

In the first example, allocatable variables (*a* and *b*) are first allocated on the host, and then mapped onto a device in the Target 1 and 2 sections, respectively. For *a* the map is implicit and for *b* an explicit map is used. Both are mapped with the default **tofrom** map type. The user-level behavior is similar to non-allocatable arrays. However, the mapping operations include creation of the allocatable variable, creation of the allocated storage, setting the allocation status to allocated, and making sure the allocatable variable references the storage.

In Target 3 and 4 sections, allocatable variables are mapped in two different ways before they are allocated on the host and subsequently used on the device. In one case, a **target data** construct creates an enclosing region for the allocatable variable to persist, and in the other case a **declare target** directive maps the allocation variable for all device executions. In both cases the new array storage is mapped **tofrom** with the **always** modifier. An explicit map is used here with an **always** modifier to ensure that the allocatable variable status is updated on the device.

Note: OpenMP 5.1 specifies that an **always** map modifier guarantees the allocation status update for an existing allocatable variable on the device.

In Target 3 and 4 sections, the behavior of an allocatable variable is very much like a Fortran pointer, in which a pointer can be mapped to a device with an associated or disassociated status, and associated storage can be mapped and attached as needed. For allocatable variables, the update of the allocation status to allocated (allowing reference to allocated storage) on the device, is similar to pointer attachment.

Example target_fort_allocatable_map.1.f90 (**omp_5.1**)

```
S-1   program main
S-2     implicit none
S-3     integer :: i
S-4
S-5     integer, save, allocatable :: d(:)
S-6     !$omp    declare target(d)
S-7
S-8     integer, allocatable :: a(:)
S-9     integer, allocatable :: b(:)
S-10    integer, allocatable :: c(:)
S-11
S-12    allocate(a(4))
S-13    !$omp target                      ! Target 1
S-14      a(:) = 4
S-15    !$omp end target
S-16    print *, a ! prints 4*4
S-17
S-18    allocate(b(4))
S-19    !$omp target map(b)               ! Target 2
S-20      b(:) = 4
S-21    !$omp end target
S-22    print *, b ! prints 4*4
S-23
S-24    !$omp target data map(c)
S-25
S-26      allocate(c(4), source=[1,2,3,4])
S-27      !$omp target map(always,tofrom:c) ! Target 3
S-28         c(:) = 4
S-29      !$omp end target
S-30      print *, c ! prints 4*4
S-31
S-32      deallocate(c)
S-33
S-34    !$omp end target data
```

```
S-35
S-36      allocate(d(4), source=[1,2,3,4])
S-37      !$omp target map(always,tofrom:d) ! Target 4
S-38         d(:) = d(:) + [ ( i,i=size(d),1,-1) ]
S-39      !$omp end target
S-40      print *, d ! prints 4*5
S-41
S-42      deallocate(a, b, d)
S-43
S-44    end program
```

Once an allocatable variable has been allocated on the host, its allocation status may not be changed in a **target** region, either explicitly or implicitly. The following example illustrates typical operations on allocatable variables that violate this restriction. Note, an assignment that reshapes or reassigns (causing a deallocation and allocation) in a **target** region is not conforming. Also, an initial intrinsic assignment of an allocatable variable requires deallocation before the **target** region ends.

Example target_fort_allocatable_map.2.f90 (**omp_5.1**)

```
S-1     program main
S-2       implicit none
S-3
S-4       integer, allocatable :: a(:,:), b(:), c(:)
S-5       integer              :: x(10,2)
S-6
S-7       allocate(a(2,10))
S-8
S-9       !$omp target
S-10         a = x             ! Reshape (or resize) NOT ALLOWED (implicit change)
S-11
S-12         deallocate(a)     ! Allocation status change of "a" NOT ALLOWED.
S-13
S-14         allocate(b(20))   ! Allocation of  b *
S-15
S-16         c = 10            ! Intrinsic assignment allocates c *
S-17
S-18         ! * Since an explicit deallocation for b and c does not occur before
S-19         ! the end of the target region, the PROGRAM BEHAVIOR IS UNSPECIFIED.
S-20      !$omp end target
S-21
S-22    end program
```

The next example illustrates a corner case of this restriction (allocatable status change in a **target** region). Two allocatable arrays are passed to a subroutine within a **target** region. The dummy-variable arrays are declared **allocatable**. Also, the *ain* variable has the **intent(in)** attribute, and *bout* has the **intent(out)** attribute. For the dummy argument with the attributes **allocatable** and **intent(out)**, the compiler will deallocate the associated actual argument when the subroutine is invoked. (However, the allocation on procedure entry can be avoided by specifying the intent as **intent(inout)**, making the intended use conforming.)

Example target_fort_allocatable_map.3.f90 (**omp_5.1**)

```
module corfu
contains
  subroutine foo(ain,bout)
    implicit none
    integer, allocatable, intent( in) :: ain(:)
    integer, allocatable, intent(out) :: bout(:) !"out" causes de/realloc
    !$omp declare target
    bout = ain
  end subroutine
end module

program  main
  use corfu
  implicit none

  integer, allocatable :: a(:)
  integer, allocatable :: b(:)
  allocate(a(10),b(10))
  a(:)=10
  b(:)=10

  !$omp target

  call foo(a,b) !ERROR: b deallocation/reallocation not allowed
                !  in target region

  !$omp end target

end program
```

Fortran

6.6 Array Sections in Device Constructs

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

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

C / C++

Example array_sections.1.c (**omp_4.0**)

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

C / C++

Fortran

Example array_sections.1.f90 (**omp_4.0**)

```
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
```

Fortran

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

C / C++

Example array_sections.2.c (**omp_4.0**)

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

C / C++

Fortran

Example array_sections.2.f90 (**omp_4.0**)

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

Fortran

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

C / C++

Example array_sections.3.c (**omp_4.0**)

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

C / C++

Fortran

Example array_sections.3.f90 (**omp_4.0**)

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

Fortran

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

C / C++

Example array_sections.4.c (**omp_4.0**)

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

C / C++

Fortran

Example array_sections.4.f90 (**omp_4.0**)

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

Fortran

6.7 Unified Shared Memory

The following examples show behavior of scalars, pointers, references (C++) and associate names (Fortran) in **target** constructs when unified shared memory (USM) is required throughout the scope of the program by the **unified_shared_memory** clause in a **requires** directive. USM assumes a unified address space.

C++

In the C++ code of the first example, a scalar (*x*), a pointer (*ptr*), and a reference (*ref*) are used in a **target** construct in Cases 1, 2 and 3, respectively. For the scalar variable *x*, the predetermined data-sharing attribute is still firstprivate under the USM requirement and, hence, any manipulation of the local variable on the device is never reflected on the host. With the USM requirement, pointers always refer to the same location in memory on the host and devices. Hence, the value of *x* (in the host data environment) can be modified with *ptr* in the **target** construct, as seen in Case 2. For the reference *ref*, the object to which it refers is mapped for the **target** construct, as seen in Case 3.

Example usm_scalar_ptr_ref_asc.1.cpp (**omp_5.2**)

```
S-1   #include <stdio.h>
S-2
S-3   #pragma omp requires unified_shared_memory
S-4
S-5   int main(){
S-6     int x = 0;     // scalar
S-7     int *ptr = &x; // pointer to a scalar
S-8     int &ref = x;  // reference to a scalar
S-9
S-10    bool pass = true;
S-11
S-12    // Case 1: x is firstprivate
S-13    #pragma omp target
S-14    {
S-15       x++;
S-16    }
S-17    if( x != 0 ) pass = false;
S-18
S-19    x = 0;
S-20    // Case 2: ptr is firstprivate
S-21    //         (uses address assigned in host data environment)
S-22    #pragma omp target
S-23    {
S-24       (*ptr)++;
S-25    }
S-26    if( x != 1 ) pass = false;
S-27
S-28    x = 0;
```

```
  // Case 3: ref and its object are mapped
  #pragma omp target
  {
    ref++;
  }
  if( x != 1 ) pass = false;

  // Verification
  if( pass ) { printf("PASSED\n"); return 0; }
  else       { printf("FAILED\n"); return 1; }

}
```

C++

Fortran

In Case 1 of the Fortran example, the scalar *x* is firstprivate under the USM requirement in the **target** construct, and modification of the local variable on the device is never updated to the host data environment. Also, in Case 2, the use of *ax*, which is associated with *x*, will update the *x* value in the host data environment. In Case 3, the Fortran pointer *ptr* and its target *y* are not firstprivate, but implicitly mapped. Hence, updates to the value of *y* appear in the host data environment.

Example usm_scalar_ptr_ref_asc.1.f90 (**omp_5.2**)

```
program main
   !$omp requires unified_shared_memory

   logical           :: pass=.TRUE.

   integer           :: x
   integer, target   :: y
   integer, pointer  :: ptr

   x = 0
   ! Case 1 : x is firstprivate
   !$omp target
      x = x + 1
   !$omp end target
   if(x /= 0 ) pass = .FALSE.

   x = 0
   ASSOCIATE( ax => x)

   ! Case 2 :
   !$omp target
      ax = ax + 1
   !$omp end target
```

```
S-24        if(x /= 1 ) pass = .FALSE.
S-25
S-26        end ASSOCIATE
S-27
S-28        y = 0
S-29        ptr => y
S-30
S-31        ! Case 3a : ptr is mapped
S-32        !$omp target
S-33           ptr = ptr + 1
S-34        !$omp end target
S-35        if(y /= 1 ) pass = .FALSE.
S-36
S-37        y = 0
S-38
S-39        ! Case 3b : y is mapped
S-40        !$omp target
S-41           y = y + 1
S-42        !$omp end target
S-43        if(y /= 1 ) pass = .FALSE.
S-44
S-45
S-46      if(pass) then
S-47        print*, "PASSED"
S-48      else
S-49        print*, "FAILED"; stop 1
S-50      endif
S-51
S-52    end program
```

Fortran

C++

6.8 C++ Virtual Functions

The 5.2 OpenMP Specification clarified restrictions on the use of polymorphic classes and virtual functions when used within **target** regions. The following examples illustrate use cases and the limitations imposed by restrictions for references and the use of Unified Shared Memory.

The first example illustrates two simple cases of using a virtual function through a pointer and reference without Unified Shared Memory.

A class, *D*, is derived from class *A*. Function *vf* in class *A* is declared virtual, and the function *vf* in class *D* is declared with override. An object, *d* of type *D*, is created and mapped through a **target data** directive.

In the first case, a pointer of type *A*, *ap*, is assigned to point to the derived object *d*, and in the first **target** region the pointer is used to call the *vf* function of the derived class, *D*.

In the second case, the reference variable *ar* of type *A* references the derived object *d*). The use of the reference variable *ar* in the **target** region is illegal due to the restriction that the static and dynamic types must match when mapping an object for the first time. That is, the behavior of the implicit map of *ar* is non-conforming – its static type *A* doesn't match its dynamic type *D*. Hence the behavior of the access to the virtual functions is unspecified.

Example virtual_functions.1.cpp (**omp_5.2**)

```
#include <iostream>

#pragma omp begin declare target
class A {
  public:
    virtual void vf()  { std::cout << "In A\n"; }
};

class D: public A {
  public:
    void vf() override { std::cout << "In D\n"; }
};
#pragma omp end declare target

int main(){

   D d;                // D derives from A, and vf() is virtual

 #pragma omp target data map(d)
 {
    // Case 1
    A *ap = &d;         // pointer to derived object d
    #pragma omp target // ap is firstprivate
    {
       ap->vf();        // calls D::vf()
    }

    // Case 2
    A &ar = d;          // reference to Derived object d
    #pragma omp target // ar is implicitly mapped
    {
       ar.vf();         // unspecified behavior
    }
 }
```

```
S-35
S-36        return 0;
S-37    }
```

The second example illustrates the restriction:

"Invoking a virtual member function of an object on a device other than the device on which the object was constructed results in unspecified behavior, unless the object is accessible and was constructed on the host device."

In the first case, an instantiation `ap` of a polymorphic class (`A`) occurs in the **`target`** region, and access of its virtual function is incorrectly attempted on the host (another device).

In the second case, the object `ap` is instantiated on the host; access of `ap` within the next **`target`** region is permitted. (Unified Shared Memory is used here to minimize mapping concerns.)

Example virtual_functions.2.cpp (**`omp_5.2`**)

```
S-1     #include <iostream>
S-2     #pragma omp requires unified_shared_memory
S-3
S-4     #pragma omp begin declare target
S-5     class A {
S-6       public:
S-7        virtual void vf()  { std::cout << "In A\n"; }
S-8     };
S-9
S-10    class D: public A {
S-11      public:
S-12       void vf() override { std::cout << "In D\n"; }
S-13    };
S-14    #pragma omp end declare target
S-15
S-16    int main(){
S-17
S-18       A *ap = nullptr;
S-19       // Case 1
S-20       #pragma omp target
S-21       {
S-22            ap = new D();
S-23       }
S-24       ap->vf();     // illegal
S-25       #pragma omp target
S-26       {
S-27          delete ap;
S-28       }
```

```
   // Case 2
   ap = new D();
   #pragma omp target  // No need for mapping with Unified Share Memory
   {
      ap->vf();  // ok
   }

   return 0;
}
```

C++

6.9 Array Shaping

C / C++

A pointer variable can be shaped to a multi-dimensional array to facilitate data access. This is achieved by a *shape-operator* casted in front of a pointer (lvalue expression):

([s_1]**[**s_2**]...[**s_n**])***pointer*

where each s_i is an integral-type expression of positive value. The shape-operator can appear in either the *motion-clause* of the **target update** directive or the **depend** clause.

The following example shows the use of the shape-operator in the **target update** directive. The shape-operator `([nx][ny+2])` casts pointer variable `a` to a 2-dimensional array of size `nx*(ny+2)`. The resulting array is then accessed as array sections (such as `[0:nx][1]` and `[0:nx][ny]`) in the **from** or **to** clause for transferring two columns of noncontiguous boundary data from or to the device. Note the use of additional parentheses around the shape-operator and `a` to ensure the correct precedence over array-section operations.

Example array_shaping.1.c (**omp_5.1**)

```
#pragma omp begin declare target
  int do_work(double *a, int nx, int ny);
  int other_work(double *a, int nx, int ny);
#pragma omp end declare target

void exch_data(double *a, int nx, int ny);

void array_shaping(double *a, int nx, int ny)
{
   // map data to device and do work
   #pragma omp target data map(a[0:nx*(ny+2)])
   {
```

```
S-13          // do work on the device
S-14          #pragma omp target  // map(a[0:nx*(ny+2)]) is optional here
S-15          do_work(a, nx, ny);
S-16
S-17          // update boundary points (two columns of 2D array) on the host
S-18          // pointer is shaped to 2D array using the shape-operator
S-19          #pragma omp target update from( (([nx][ny+2])a)[0:nx][1], \
S-20                                          (([nx][ny+2])a)[0:nx][ny] )
S-21
S-22          // exchange ghost points with neighbors
S-23          exch_data(a, nx, ny);
S-24
S-25          // update ghost points (two columns of 2D array) on the device
S-26          // pointer is shaped to 2D array using the shape-operator
S-27          #pragma omp target update to( (([nx][ny+2])a)[0:nx][0], \
S-28                                        (([nx][ny+2])a)[0:nx][ny+1] )
S-29
S-30          // perform other work on the device
S-31          #pragma omp target  // map(a[0:nx*(ny+2)]) is optional here
S-32          other_work(a, nx, ny);
S-33       }
S-34    }
```

C / C++

Fortran

The shape operator is not defined for Fortran. Explicit array shaping of procedure arguments can be used instead to achieve a similar goal. Below is the Fortran equivalent of the above example that illustrates the support of transferring two rows of noncontiguous boundary data in the **target update** directive.

Example array_shaping.1.f90 (**omp_5.2**)

```
S-1     module m
S-2        interface
S-3           subroutine do_work(a, nx, ny)
S-4              !$omp declare target enter(do_work)
S-5              integer, intent(in) :: nx, ny
S-6              double precision a(0:nx+1,ny)
S-7           end subroutine do_work
S-8
S-9           subroutine other_work(a, nx, ny)
S-10             !$omp declare target enter(other_work)
S-11             integer, intent(in) :: nx, ny
S-12             double precision a(0:nx+1,ny)
S-13          end subroutine other_work
S-14
S-15          subroutine exch_data(a, nx, ny)
```

```
S-16            integer, intent(in) :: nx, ny
S-17            double precision a(0:nx+1,ny)
S-18          end subroutine exch_data
S-19       end interface
S-20    end module m
S-21
S-22    subroutine array_shaping(a, nx, ny)
S-23       use m
S-24       implicit none
S-25       integer, intent(in) :: nx, ny
S-26       double precision a(0:nx+1,ny)
S-27
S-28       ! map data to device and do work
S-29       !$omp target data map(a)
S-30
S-31          ! do work on the device
S-32          !$omp target      ! map(a) is optional here
S-33          call do_work(a, nx, ny)
S-34          !$omp end target
S-35
S-36          ! update boundary points (two rows of 2D array) on the host.
S-37          ! data transferred are noncontiguous
S-38          !$omp target update from( a(1,1:ny), a(nx,1:ny) )
S-39
S-40          ! exchange ghost points with neighbors
S-41          call exch_data(a, nx, ny)
S-42
S-43          ! update ghost points (two rows of 2D array) on the device.
S-44          ! data transferred are noncontiguous
S-45          !$omp target update to( a(0,1:ny), a(nx+1,1:ny) )
S-46
S-47          ! perform other work on the device
S-48          !$omp target      ! map(a) is optional here
S-49          call other_work(a, nx, ny)
S-50          !$omp end target
S-51
S-52       !$omp end target data
S-53
S-54    end subroutine
```

Fortran

6.10 `declare mapper` Directive

The following examples show how to use the **`declare mapper`** directive to prescribe a map for later use. It is also quite useful for pre-defining partitioned and nested structure elements.

In the first example the **`declare mapper`** directive specifies that any structure of type *myvec_t* for which implicit data-mapping rules apply will be mapped according to its **`map`** clause. The variable *v* is used for referencing the structure and its elements within the **`map`** clause. Within the **`map`** clause the *v* variable specifies that all elements of the structure are to be mapped. Additionally, the array section *v.data[0:v.len]* specifies that the dynamic storage for data is to be mapped.

Within the main program the *s* variable is typed as *myvec_t*. Since the variable is found within the **`target`** region and the type has a mapping prescribed by a **`declare mapper`** directive, it will be automatically mapped according to its prescription: full structure, plus the dynamic storage of the *data* element.

C / C++

Example target_mapper.1.c (**`omp_5.0`**)

```
S-1   #include  <stdlib.h>
S-2   #include   <stdio.h>
S-3   #define N 100
S-4
S-5   typedef struct myvec{
S-6       size_t len;
S-7       double *data;
S-8   } myvec_t;
S-9
S-10  #pragma omp declare mapper(myvec_t v) \
S-11                      map(v, v.data[0:v.len])
S-12  void init(myvec_t *s);
S-13
S-14  int main(){
S-15     myvec_t s;
S-16
S-17     s.data = (double *)calloc(N,sizeof(double));
S-18     s.len  = N;
S-19
S-20     #pragma omp target
S-21     init(&s);
S-22
S-23     printf("s.data[%d]=%lf\n",N-1,s.data[N-1]);  //s.data[99]=99.000000
S-24  }
S-25
S-26  void init(myvec_t *s)
S-27  { for(size_t i=0; i<s->len; i++) s->data[i]=i; }
```

C / C++

Fortran

Example target_mapper.1.f90 (**omp_5.0**)

```
module my_structures
  type myvec_t
    integer                       :: len
    double precision, pointer     :: data(:)
  end type
end module

program main
  use my_structures
  integer, parameter :: N=100

  !$omp  declare mapper(myvec_t :: v) &
  !$omp&          map(v, v%data(1:v%len))

  type(myvec_t) :: s

  allocate(s%data(N))
  s%data(1:N) = 0.0d0
  s%len = N

  !$omp target
  call init(s)
  !$omp end target

  print*,"s%data(",N,")=",s%data(N)  !! s%data( 100 )=100.000000000000
end program

subroutine init(s)
  use my_structures
  type(myvec_t) :: s
  !$omp declare target
  s%data = [ (i, i=1,s%len) ]
end subroutine
```

Fortran

The next example illustrates the use of the *mapper-identifier* and deep copy within a structure. The structure, *dzmat_t*, represents a complex matrix, with separate real (*r_m*) and imaginary (*i_m*) elements. Two map identifiers are created for partitioning the *dzmat_t* structure.

For the C/C++ code the first identifier is named *top_id* and maps the top half of two matrices of type *dzmat_t*; while the second identifier, *bottom_id*, maps the lower half of two matrices. Each identifier is applied to a different **target** construct, as **map(mapper(***top_id***), tofrom:** *a,b***)** and **map(mapper(***bottom_id***), tofrom:** *a,b***)**. Each target offload is allowed to execute concurrently on two different devices (*0* and *1*) through the **nowait** clause.

The Fortran code uses the *left_id* and *right_id* map identifiers in the **map(mapper(***left_id***),tofrom:** *a,b***)** and **map(mapper(***right_id***),tofrom:** *a,b***)** map clauses. The array sections for these left and right contiguous portions of the matrices were defined previously in the **declare mapper** directive.

Note, the *is* and *ie* scalars are firstprivate by default for a **target** region, but are declared firstprivate anyway to remind the user of important firstprivate data-sharing properties required here.

C / C++

Example target_mapper.2.c (**omp_5.0**)

```
S-1   #include <stdio.h>
S-2   //                    N MUST BE EVEN
S-3   #define N  100
S-4
S-5     typedef struct dzmat
S-6     {
S-7        double r_m[N][N];
S-8        double i_m[N][N];
S-9     } dzmat_t;
S-10
S-11    #pragma omp declare mapper( top_id: dzmat_t v) \
S-12                        map(v.r_m[0:N/2][0:N],     \
S-13                            v.i_m[0:N/2][0:N]      )
S-14
S-15    #pragma omp declare mapper(bottom_id: dzmat_t v) \
S-16                        map(v.r_m[N/2:N/2][0:N],     \
S-17                            v.i_m[N/2:N/2][0:N]      )
S-18  //initialization
S-19  void dzmat_init(dzmat_t *z, int is, int ie, int n);
S-20  //matrix add: c=a+b
S-21  void host_add(  dzmat_t *a, dzmat_t *b, dzmat_t *c, int n);
S-22
S-23
S-24  int main()
S-25  {
S-26    dzmat_t a,b,c;
S-27    int     is,ie;
S-28
S-29    is=0; ie=N/2-1;       //top N/2 rows on device 0
S-30    #pragma omp target map(mapper(top_id), tofrom: a,b) device(0) \
S-31                       firstprivate(is,ie) nowait
S-32    {
S-33      dzmat_init(&a,is,ie,N);
S-34      dzmat_init(&b,is,ie,N);
S-35    }
S-36
```

```
S-37      is=N/2; ie=N-1;       //bottom N/2 rows on device 1
S-38      #pragma omp target map(mapper(bottom_id), tofrom: a,b) device(1) \
S-39                          firstprivate(is,ie) nowait
S-40      {
S-41        dzmat_init(&a,is,ie,N);
S-42        dzmat_init(&b,is,ie,N);
S-43      }
S-44
S-45      #pragma omp taskwait
S-46
S-47      host_add(&a,&b,&c,N);
S-48    }
```

C / C++

Fortran

1 *Example target_mapper.2.f90* (`omp_5.0`)

```
 S-1    module complex_mats
 S-2
 S-3       integer, parameter :: N=100    !N must be even
 S-4       type dzmat_t
 S-5         double precision ::  r_m(N,N), i_m(N,N)
 S-6       end type
 S-7
 S-8       !$omp  declare mapper( left_id: dzmat_t :: v) map( v%r_m(N,  1:N/2), &
 S-9       !$omp&                                             v%i_m(N,  1:N/2))
S-10
S-11       !$omp  declare mapper(right_id: dzmat_t :: v) map( v%r_m(N,N/2+1:N), &
S-12       !$omp&                                             v%i_m(N,N/2+1:N))
S-13
S-14    end module
S-15
S-16
S-17    program main
S-18      use  complex_mats
S-19      type(dzmat_t) :: a,b,c
S-20      external dzmat_init, host_add  !initialization and matrix add: a=b+c
S-21
S-22      integer :: is,ie
S-23
S-24
S-25      is=1; ie=N/2            !left N/2 columns on device 0
S-26      !$omp target map(mapper( left_id), tofrom: a,b) device(0) &
S-27      !$omp&       firstprivate(is,ie) nowait
S-28        call dzmat_init(a,is,ie)
S-29        call dzmat_init(b,is,ie)
S-30      !$omp end target
```

```
  is=N/2+1; ie=N          !right N/2 columns on device 1
  !$omp target map(mapper(right_id), tofrom: a,b) device(1) &
  !$omp&       firstprivate(is,ie) nowait
    call dzmat_init(a,is,ie)
    call dzmat_init(b,is,ie)
  !$omp end target

  !$omp taskwait

  call host_add(a,b,c)

end program main
```

Fortran

In the third example *myvec* structures are nested within a *mypoints* structure. The *myvec_t* type is mapped as in the first example. Following the *mypoints* structure declaration, the *mypoints_t* type is mapped by a **declare mapper** directive. For this structure the *hostonly_data* element will not be mapped; also the array section of *x* (*v.x[:1]*) and *x* will be mapped; and *scratch* will be allocated and used as scratch storage on the device. The default map-type mapping, **tofrom**, applies to the *x* array section, but not to *scratch* which is explicitly mapped with the **alloc** map-type. Note: the variable *v* is not included in the map list (otherwise the *hostonly_data* would be mapped)– just the elements to be mapped are listed.

The two mappers are combined when a *mypoints_t* structure type is mapped, because the mapper *myvec_t* structure type is used within a *mypoints_t* type structure.

C / C++

Example target_mapper.3.c (**omp_5.0**)

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

#define N 100

typedef struct myvec {
    size_t len;
    double *data;
} myvec_t;

#pragma omp declare mapper(myvec_t v) \
                    map(v, v.data[0:v.len])

typedef struct mypoints {
    struct myvec scratch;
    struct myvec *x;
    double hostonly_data[500000];
```

```
} mypoints_t;

#pragma omp declare mapper(mypoints_t v)  \
                    map(v.x, v.x[0] ) map(alloc:v.scratch)

void init_mypts_array(mypoints_t *P, int n);
void eval_mypts_array(mypoints_t *P, int n);

int main(){

   mypoints_t P;

   init_mypts_array(&P, N);

   #pragma omp target map(P)
   eval_mypts_array(&P, N);

}
```

C / C++

Fortran

Example target_mapper.3.f90 (`omp_5.0`)

```
module my_structures
  type myvec_t
    integer                    :: len
    double precision, pointer :: data(:)
  end type
  !$omp  declare mapper(myvec_t :: v) &
  !$omp&         map(v, v%data(:))

  type mypoints_t
     type(myvec_t)             :: scratch
     type(myvec_t), pointer    :: x(:)
     double precision          :: hostonly_data(500000)
  end  type
 !$omp  declare mapper(mypoints_t :: v)  &
 !$omp&         map(v%x, v%x(1)) map(alloc:v%scratch)

end module


program main
  use my_structures
  external  init_mypts_array, eval_mypts_array

  type(mypoints_t) :: P
```

```
S-25
S-26        call init_mypts_array(P)
S-27
S-28        !$omp target map(P)
S-29        call eval_mypts_array(P)
S-30        !$omp end target
S-31
S-32    end program
```

Fortran

6.11 `target data` Construct

6.11.1 Simple `target data` Construct

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

C / C++

Example target_data.1.c (**`omp_4.0`**)

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

C / C++

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

Fortran

Example target_data.1.f90 (**omp_4.0**)

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

Fortran

6.11.2 `target data` Region Enclosing Multiple `target` Regions

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

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

C / C++

Example target_data.2.c (**omp_4.0**)

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

C / C++

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

Fortran

Example target_data.2.f90 (**omp_4.0**)

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

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

Fortran

In the following example, the array *Q* is mapped once at the enclosing **target data** region instead of at each **target** construct. In OpenMP 4.0, a scalar variable is implicitly mapped with the **tofrom** map-type. But since OpenMP 4.5, a scalar variable, such as the *tmp* variable, has to be explicitly mapped with the **tofrom** map-type at the first **target** construct in order to return its reduced value from the parallel loop construct to the host. The variable defaults to firstprivate at the second **target** construct.

C / C++

Example target_data.3.c (**omp_4.0**)

```
#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
```

```
S-27              execution with 4.5 (and pre-4.5) compliant compilers.
S-28              See Devices Intro.
S-29     */
```

C / C++

Fortran

Example target_data.3.f90 (`omp_4.0`)

```
S-1     subroutine gramSchmidt(Q,rows,cols)
S-2     integer             ::   rows,cols,  i,k
S-3     double precision    :: Q(rows,cols), tmp
S-4           !$omp target data map(Q)
S-5           do k=1,cols
S-6              tmp = 0.0d0
S-7             !$omp target map(tofrom: tmp)
S-8                !$omp parallel do reduction(+:tmp)
S-9                do i=1,rows
S-10                  tmp = tmp + (Q(i,k) * Q(i,k))
S-11               end do
S-12            !$omp end target
S-13
S-14              tmp = 1.0d0/sqrt(tmp)
S-15
S-16            !$omp target
S-17               !$omp parallel do
S-18               do i=1,rows
S-19                   Q(i,k) = Q(i,k)*tmp
S-20               enddo
S-21            !$omp end target
S-22          end do
S-23          !$omp end target data
S-24    end subroutine
S-25
S-26    ! Note:  The variable tmp is now mapped with tofrom, for correct
S-27    ! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
```

Fortran

6.11.3 `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 (**`omp_4.0`**)

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

```
S-21  {
S-22     int i;
S-23     #pragma omp target map(to: v3[0:N], v4[:N]) map(from: p1[0:N])
S-24     #pragma omp parallel for
S-25     for (i=0; i<N; i++)
S-26     {
S-27       p1[i] = v3[i] * v4[i];
S-28     }
S-29  }
```

C / C++

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

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

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

Fortran

Example target_data.4.f90 (**omp_4.0**)

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

```
S-22        !$omp parallel do
S-23        do i=1,N
S-24           p1(i) = v3(i) * v4(i)
S-25        end do
S-26        !$omp end target
S-27
S-28     end subroutine
S-29     end module
```

Fortran

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

C++

Example target_data.5.cpp (**omp_4.0**)

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

C++

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

Fortran

Example target_data.5.f90 (**omp_4.0**)

```
S-1   module my_mult
S-2   contains
S-3   subroutine foo(p0,v1,v2,N)
S-4   real,dimension(:) :: p0, v1, v2
S-5   integer           :: N,i
S-6      call init(v1, v2, N)
S-7      !$omp target data map(to: v1, v2) map(from: p0)
S-8       call vec_mult(p0,v1,v2,N)
S-9      !$omp end target data
S-10     call output(p0, N)
S-11  end subroutine
S-12  subroutine vec_mult(p1,v3,v4,N)
S-13  real,dimension(:) :: p1, v3, v4
S-14  integer           :: N,i
S-15     !$omp target map(to: v3, v4) map(from: p1)
S-16     !$omp parallel do
S-17     do i=1,N
S-18        p1(i) = v3(i) * v4(i)
S-19     end do
S-20     !$omp end target
S-21  end subroutine
S-22  end module
S-23  program main
S-24  use my_mult
S-25  integer, parameter :: N=1024
S-26  real,allocatable, dimension(:) :: p, v1, v2
S-27     allocate( p(N), v1(N), v2(N) )
S-28     call foo(p,v1,v2,N)
S-29     deallocate( p, v1, v2 )
S-30  end program
```

Fortran

6.11.4 `target data` Construct with `if` Clause

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

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

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

C / C++

Example target_data.6.c (**omp_4.0**)

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

C / C++

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

Fortran

Example target_data.6.f90 (**omp_4.0**)

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

Fortran

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

C / C++

Example target_data.7.c (**omp_4.0**)

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

C / C++

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 (**omp_4.0**)

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

```
      !$omp end target
   !$omp end target data
   call output(p, N)  !*** UNDEFINED behavior if N<=THRESHOLD
end subroutine
```

Fortran

6.12 `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.

C++

Example target_unstructured_data.1.cpp (**`omp_4.5`**)

```
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 destructor is called.

    #pragma omp target exit data map(delete:v[0:len])
    delete[] v;
  }

  private:
```

```
  double* v;
  int len;

};
```

C++

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

C / C++

Example target_unstructured_data.1.c (**omp_4.5**)

```
#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;
}
```

C / C++

The following Fortran code allocates and deallocates a module array, *A*. The *initialize* subroutine allocates the module array and uses the **target enter data** directive to map it to the target device. The *finalize* subroutine removes the mapped array from the target device and then deallocates the array on the host. Note, the stand-alone **target enter data** occurs after the host memory is allocated, and the **target exit data** construct occurs before the host data is deallocated.

Fortran

Example target_unstructured_data.1.f90 (**omp_4.5**)

```
S-1   module example
S-2     real(8), allocatable :: A(:)
S-3
S-4     contains
S-5       subroutine initialize(N)
S-6         integer :: N
S-7
S-8         allocate(A(N))
S-9         !$omp target enter data map(alloc:A)
S-10
S-11      end subroutine initialize
S-12
S-13      subroutine finalize()
S-14
S-15        !$omp target exit data map(delete:A)
S-16        deallocate(A)
S-17
S-18      end subroutine finalize
S-19  end module example
```

Fortran

6.13 `target update` Construct

6.13.1 Simple `target data` and `target update` Constructs

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

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

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

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

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

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

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

C / C++

Example target_update.1.c (**omp_4.0**)

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

C / C++

Fortran

Example target_update.1.f90 (**omp_4.0**)

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

```
S-14          !$omp target
S-15          !$omp parallel do
S-16             do i=1,N
S-17                p(i) = p(i) + v1(i) * v2(i)
S-18             end do
S-19          !$omp end target
S-20       !$omp end target data
S-21       call output(p, N)
S-22    end subroutine
```

Fortran

6.13.2 `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 (**`omp_4.0`**)

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

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

C / C++

Fortran

Example target_update.2.f90 (`omp_4.0`)

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

Fortran

6.13.3 `target update` Construct with Mapper

The following example shows how the **`target update`** construct can be used with a **`mapper`** (*`custom`*). The *`custom`* mapper maps members of structure *`T`* with different map-type modifiers. Inside the **`target data`** region the **`target update`** with the **`to`** *motion-clause* is equivalent to an update of *`x`* on the device. After the **`target`** region the **`target update`** with the **`from`** motion-clause is equivalent to an update of *`y`* on the host.

C / C++

Example target_update.3.c (**`omp_5.1`**)

```
S-1   #include <stdio.h>
S-2   #include <stdlib.h>
S-3
S-4   typedef struct{
S-5      int x;
S-6      int y;
S-7      int z;
S-8   }T;
S-9
S-10  #pragma omp declare mapper(custom: T S) map(to:S.x) \
S-11                      map(from:S.y) map(alloc: S.z)
S-12
S-13  int main()
S-14  {
S-15      T s;
S-16
S-17      s.x = 5;
S-18      s.y = 5;
S-19      s.z = 5;
S-20      #pragma omp target data map(mapper(custom),tofrom: s)
S-21      {
S-22          int a,b,c;
S-23          s.x += 5;
S-24          s.y += 5;
S-25          s.z += 5;
S-26
S-27          #pragma omp target update to(mapper(custom): s)
S-28          // becomes #pragma omp target update to(s.x)
S-29
S-30          #pragma omp target map(from: a,b,c)
S-31          {
S-32              a = s.x;
S-33              b = s.y;            //s.y is undefined here
S-34              c = s.z;            //s.z is undefined here
S-35
S-36              s.y = 5;
```

```
S-37
S-38                printf("s.x:%d, s.y:%d \n", s.x, s.y);
S-39                     // s.x:10, s.y:5 (value of s.z is undefined)
S-40            }
S-41            #pragma omp target update from(mapper(custom): s)
S-42            // becomes #pragma omp target update from(s.y)
S-43
S-44            printf("s.y:%d \n", s.y);
S-45                 // s.y:5
S-46            printf("a:%d \n", a);
S-47                 // a:10 (values of b and c are undefined)
S-48        }
S-49        printf("s.x:%d, s.y:%d, s.z:%d\n", s.x, s.y, s.z);
S-50             // s.x:10, s.y:5, s.z:10
S-51
S-52        return 0;
S-53    }
```

C / C++

Fortran

1 *Example target_update.3.f90* (`omp_5.1`)

```
S-1         module my_struct
S-2          type T
S-3           integer :: x,y,z
S-4          end type
S-5         end module
S-6
S-7         program main
S-8          use my_struct
S-9          integer, parameter :: N=100
S-10         integer  :: a,b,c
S-11
S-12        !$omp declare mapper(custom: T :: v) &
S-13        !$omp& map(to:v%x) map(from:v%y) map(alloc: v%z)
S-14
S-15         type(T) :: s
S-16
S-17         s%x = 5
S-18         s%y = 5
S-19         s%z = 5
S-20
S-21         !$omp target data map(mapper(custom),tofrom: s)
S-22
S-23         s%x = s%x + 5
S-24         s%y = s%y + 5
S-25         s%z = s%z + 5
```

```
S-26
S-27          !$omp target update to(mapper(custom) : s)
S-28
S-29          !$omp target map(from: a,b,c)
S-30            a = s%x
S-31            b = s%y
S-32            c = s%z
S-33
S-34            s%y = 5
S-35            print*,"s%x:", s%x, " s%y:", s%y
S-36                 !! s%x:10, s%y:5 (value of s%z is undefined)
S-37          !$omp end target
S-38
S-39          !$omp target update from(mapper(custom) : s)
S-40          print*, "s%y:", s%y !! s%y:5
S-41          print*, "a:", a     !! a:10 (values of b and c are undefined)
S-42
S-43          !$omp end target data
S-44
S-45          print*, "s%x:", s%x, " s%y:", s%y, " s%z:", s%z
S-46               !! s%x:10, s%y:5, s%z:10
S-47        end program
```

Fortran

6.14 Declare Target Directive

6.14.1 Declare Target Directive for a Procedure

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* procedure 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 the following C/C++ code the declaration of the function *fib* appears between the **begin declare target** and **end declare target** directives. In the corresponding Fortran code, the **declare target** directive appears at the end of the specification part of the subroutine.

C / C++

Example declare_target.1.c (**omp_5.1**)

```
S-1   #pragma omp begin declare target
S-2   extern void fib(int N);
S-3   #pragma omp end declare target
S-4
S-5   #define THRESHOLD 1000000
S-6   void fib_wrapper(int n)
S-7   {
S-8      #pragma omp target if(n > THRESHOLD)
S-9      {
S-10        fib(n);
S-11     }
S-12  }
```

C / C++

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

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

Fortran

Example declare_target.1.f90 (**omp_4.0**)

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

Fortran

The next Fortran example shows the use of an external subroutine. As the subroutine is neither use associated nor an internal procedure, the **declare target** declarations within a external subroutine are unknown to the main program unit; therefore, a **declare target** must be provided within the program scope for the compiler to determine that a target binary should be available.

Fortran

Example declare_target.2.f90 (**omp_4.0**)

```
S-1   program my_fib
S-2   integer :: N = 8
S-3   interface
S-4     subroutine fib(N)
S-5       !$omp declare target
S-6       integer :: N
S-7     end subroutine fib
S-8   end interface
S-9      !$omp target
S-10        call fib(N)
S-11     !$omp end target
S-12  end program
S-13  subroutine fib(N)
S-14  integer :: N
S-15  !$omp declare target
S-16       print*,"hello from fib"
S-17       !...
S-18  end subroutine
```

Fortran

6.14.2 Declare Target Directive for Indirect Procedure Call

In the OpenMP 5.1 Specification the **indirect** clause was added to allow indirect procedure calls, via function pointers, in a **target** region. The functions to be allowed indirect invocation are specified in an **enter** clause of a declare target directive, along with the **indirect** clause. The clause has an optional enabling/disabling argument (default enabled). In the absence of the indirect clause the function pointer would be mapped as a scalar (firstprivate) that would point to the host versions of the functions. Indirect clause informs the compiler that the function can potentially be used via function pointers and to use device versions of the same within the target region.

Only with an enabled **indirect** clause and a function specification in an **enter** clause of a declare target directive may a function be called with an indirect invocation in a **target** region. (Note: this feature limits the number of functions that can be used by function pointers in the **target** region to a restricted list for the compiler.)

In the following example, the **declare target enter(***fun1***,** *fun2***) indirect** directive specifies that the *fun1* and *fun2* functions may be invoked with a function pointer in the **target** region. Either the *fun1* or *fun2* function is invoked by the *fptr* function pointer in the **target** construct, as determined by the value of *count*.

C / C++

Example declare_target_indirect_call.1.c (**omp_5.2**)

```
S-1   #include <stdio.h>
S-2   #include <stdlib.h>
S-3
S-4   typedef int(*funcptr)();
S-5
S-6   int fun1() {return 1;}
S-7   int fun2() {return 2;}
S-8   #pragma omp declare target enter(fun1, fun2) indirect
S-9                                              // indirect defaults to true
S-10  int main()
S-11  {
S-12    int ret_val=0;
S-13    const int choice = rand()%2 + 1;         // create runtime number 1 or 2
S-14
S-15    funcptr fptr = (choice == 1) ? &fun1 : &fun2; //select fun1/fun2 for 1/2
S-16
S-17    #pragma omp target map(from: ret_val)
S-18      ret_val = fptr();                      // ret_val = 1/2 from fun1/fun2
S-19
S-20    if (ret_val != choice) { printf("FAILED\n"); exit(1); }
S-21
S-22    return 0;
S-23  }
```

C / C++

Fortran

Example declare_target_indirect_call.1.f90 (**omp_5.2**)

```
S-1   module funcs
S-2     implicit none
S-3
S-4     interface
S-5       function func() result(i)
S-6         integer :: i
S-7       end function
S-8     end interface
S-9
S-10   contains
S-11    function fun1() result(i)
```

```
S-12      !$omp declare target enter(fun1) indirect !! indirect defaults to true
S-13        integer :: i
S-14        i=1
S-15        return
S-16      end function
S-17
S-18      function fun2() result(i)
S-19      !$omp declare target enter(fun2) indirect !! indirect defaults to true
S-20        integer :: i
S-21        i=2
S-22        return
S-23      end function
S-24
S-25    end module
S-26
S-27    program main
S-28      use funcs
S-29      implicit none
S-30      procedure (func), pointer :: fptr=>null()
S-31      integer :: ret_val=0, choice=0
S-32      real    :: rand_no
S-33
S-34      call  random_number(rand_no)  !! create random ( [0.0 - 1.0) )
S-35      choice = nint(rand_no)+1      !! runtime number 1 or 2
S-36
S-37      if (choice == 1 ) fptr => fun1
S-38      if (choice == 2 ) fptr => fun2
S-39
S-40      !$omp target map(from: ret_val)
S-41         ret_val = fptr()  !! ret_val = 1/2 from fun1/fun2
S-42      !$omp end target
S-43
S-44      if (ret_val /= choice) then
S-45         print*, "FAILED"; error stop 1
S-46      endif
S-47
S-48    end program
```

Fortran

6.14.3 Declare Target Directive for Class Type

The following example shows the use of the **begin declare target** and **end declare target** pair to designate the beginning and end of the affected declarations, as introduced in OpenMP 5.1. The **begin declare target** directive was defined to symmetrically complement the terminating ("end") directive.

The example also shows 3 different ways to use a **declare target** directive for a class and an external member-function definition (for the *XOR1*, *XOR2*, and *XOR3* classes and definitions for their corresponding *foo()* member functions).

For *XOR1*, a **begin declare target** and **end declare target** directive enclose both the class and its member function definition. The compiler immediately knows to create a device version of the function for execution in a **target** region.

For *XOR2*, the class member function definition is not specified with a **declare target** directive. An implicit declare target is created for the member function definition. The same applies if this declaration arrangement for the class and function are included through a header file.

For *XOR3*, the class and its member function are not enclosed by **begin declare target** and **end declare target** directives, but there is an implicit declare target since the class, its function and the **target** construct are in the same file scope. That is, the class and its function are treated as if delimited by a **declare target** directive. The same applies if the class and function are included through a header file.

Example declare_target.2a.cpp (**omp_5.1**)

```
S-1   #include <iostream>
S-2   using namespace std;
S-3
S-4     #pragma omp begin declare target // declare target--class and function
S-5     class XOR1
S-6     {
S-7         int a;
S-8       public:
S-9         XOR1(int arg): a(arg) {};
S-10        int foo();
S-11    };
S-12    int XOR1::foo() { return a^0x01;}
S-13    #pragma omp end declare target
S-14
S-15    #pragma omp begin declare target // declare target--class, not function
S-16    class XOR2
S-17    {
S-18        int a;
S-19      public:
S-20        XOR2(int arg): a(arg) {};
S-21        int foo();
S-22    };
S-23    #pragma omp end declare target
S-24
S-25    int XOR2::foo() { return a^0x01;}
S-26
```

```
S-27    class XOR3                // declare target--neither class nor function
S-28    {
S-29        int a;
S-30      public:
S-31        XOR3(int arg): a(arg) {};
S-32        int foo();
S-33    };
S-34    int XOR3::foo() { return a^0x01;}
S-35
S-36  int main (){
S-37
S-38      XOR1 my_XOR1(3);
S-39      XOR2 my_XOR2(3);
S-40      XOR3 my_XOR3(3);
S-41      int res1, res2, res3;
S-42
S-43      #pragma omp target map(tofrom:res1)
S-44      res1=my_XOR1.foo();
S-45
S-46      #pragma omp target map(tofrom:res2)
S-47      res2=my_XOR2.foo();
S-48
S-49      #pragma omp target map(tofrom:res3)
S-50      res3=my_XOR3.foo();
S-51
S-52      cout << res1 << endl;  // OUT1: 2
S-53      cout << res2 << endl;  // OUT2: 2
S-54      cout << res3 << endl;  // OUT3: 2
S-55  }
```

Often class definitions and their function definitions are included in separate files, as shown in *declare_target.2b_classes.hpp* and *declare_target.2b_functions.cpp* example code files below. In this case, it is necessary to specify a declare target directive for the classes. However, as long as the *2b_functions.cpp* file includes the corresponding declare target classes, there is no need to specify the functions with a declare target directive. The functions are treated as if they are specified with a declare target directive. Compiling the *declare_target.2b_functions.cpp* and *declare_target.2b_main.cpp* files separately and linking them, will create appropriate executable device functions for the target device.

Example declare_target.2b_classes.hpp (`omp_5.1`)

```
S-1   #pragma omp begin declare target
S-2   class XOR1
S-3   {
S-4       int a;
```

```
   public:
     XOR1(int arg): a(arg) {};
     int foo();
};
#pragma omp end declare target
```

Example declare_target.2b_functions.cpp (**omp_5.1**)

```
#include "declare_target.2b_classes.hpp"
int XOR1::foo() { return a^0x01;}
```

Example declare_target.2b_main.cpp (**omp_5.1**)

```
#include <iostream>
using namespace std;

#include "declare_target.2b_classes.hpp"

int main (){

   XOR1 my_XOR1(3);
   int res1;

   #pragma omp target map(from: res1)
   res1=my_XOR1.foo();

   cout << res1 << endl;  // OUT1: 2
}
```

The following example shows how the **begin declare target** and **end declare target** directives are used to enclose the declaration of a variable *varY* with a class type *typeY*.

This example shows pre-OpenMP 5.0 behavior for the *varY.foo()* function call (an error). The member function *typeY::foo()* cannot be accessed on a target device because its declaration does not appear between **begin declare target** and **end declare target** directives. As of OpenMP 5.0, the function is implicitly declared with a declare target directive and will successfully execute the function on the device. See previous examples.

Example declare_target.2c.cpp (**omp_5.1**)

```
struct typeX
{
   int a;
};
class typeY
{
   int a;
```

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

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

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

C++

6.14.4 Declare Target Directive for Variables

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

C / C++

Example declare_target.3.c (**omp_5.1**)

```
#define N 1000

#pragma omp begin declare target
float p[N], v1[N], v2[N];
#pragma omp end declare target

extern void init(float *, float *, int);
extern void output(float *, int);

void vec_mult()
```

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

C / C++

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

Fortran

Example declare_target.3.f90 (**omp_4.0**)

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

Fortran

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

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

C / C++

Example declare_target.4.c (**omp_5.1**)

```
S-1   #define N 10000
S-2
S-3   #pragma omp begin declare target
S-4     float Q[N][N];
S-5     float Pfun(const int i, const int k) { return Q[i][k] * Q[k][i]; }
S-6   #pragma omp end declare target
S-7
S-8   float accum(int k)
S-9   {
S-10      float tmp = 0.0;
S-11      #pragma omp target update to(Q)
S-12      #pragma omp target map(tofrom: tmp)
S-13      #pragma omp parallel for reduction(+:tmp)
S-14      for(int i=0; i < N; i++)
S-15          tmp += Pfun(i,k);
S-16      return tmp;
S-17  }
S-18
S-19  /* Note:  The variable tmp is now mapped with tofrom, for correct
S-20            execution with 4.5 (and pre-4.5) compliant compilers.
S-21            See Devices Intro.
S-22   */
```

C / C++

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

Fortran

Example declare_target.4.f90 (**omp_4.0**)

```
S-1   module my_global_array
S-2   !$omp declare target (N,Q)
S-3   integer, parameter :: N=10
S-4   real               :: Q(N,N)
S-5   contains
S-6   function Pfun(i,k)
S-7   !$omp declare target
S-8   real               :: Pfun
S-9   integer,intent(in) :: i,k
S-10     Pfun=(Q(i,k) * Q(k,i))
S-11  end function
S-12  end module
S-13
S-14  function accum(k) result(tmp)
S-15  use my_global_array
S-16  real    :: tmp
S-17  integer :: i, k
S-18     tmp = 0.0e0
S-19     !$omp target map(tofrom: tmp)
S-20     !$omp parallel do reduction(+:tmp)
S-21     do i=1,N
S-22        tmp = tmp + Pfun(k,i)
S-23     end do
S-24     !$omp end target
S-25  end function
S-26
S-27  ! Note:  The variable tmp is now mapped with tofrom, for correct
S-28  ! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
```

Fortran

6.14.5 Declare Target Directive with `declare simd`

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

C / C++

Example declare_target.5.c (**omp_5.1**)

```
S-1   #define N 10000
S-2   #define M 1024
S-3
S-4   #pragma omp begin declare target
S-5   float Q[N][N];
S-6
S-7   #pragma omp declare simd uniform(i) linear(k) notinbranch
S-8   float P(const int i, const int k)
S-9   {
S-10    return Q[i][k] * Q[k][i];
S-11  }
S-12  #pragma omp end declare target
S-13
S-14  float accum(void)
S-15  {
S-16    float tmp = 0.0;
S-17    int i, k;
S-18  #pragma omp target map(tofrom: tmp)
S-19  #pragma omp parallel for reduction(+:tmp)
S-20    for (i=0; i < N; i++) {
S-21      float tmp1 = 0.0;
S-22  #pragma omp simd reduction(+:tmp1)
S-23      for (k=0; k < M; k++) {
S-24        tmp1 += P(i,k);
S-25      }
S-26      tmp += tmp1;
S-27    }
S-28    return tmp;
S-29  }
S-30
S-31  /* Note:  The variable tmp is now mapped with tofrom, for correct
S-32            execution with 4.5 (and pre-4.5) compliant compilers.
S-33            See Devices Intro.
S-34   */
```

C / C++

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

Fortran

Example declare_target.5.f90 (**omp_4.0**)

```
S-1   module my_global_array
S-2   !$omp declare target (N,Q)
S-3   integer, parameter :: N=10000, M=1024
S-4   real               :: Q(N,N)
S-5   contains
S-6   function P(k,i)
S-7   !$omp declare simd uniform(i) linear(k) notinbranch
S-8   !$omp declare target
S-9   real               :: P
S-10  integer,intent(in) :: k,i
S-11     P=(Q(k,i) * Q(i,k))
S-12  end function
S-13  end module
S-14
S-15  function accum() result(tmp)
S-16  use my_global_array
S-17  real    :: tmp, tmp1
S-18  integer :: i
S-19     tmp = 0.0e0
S-20     !$omp target map(tofrom: tmp)
S-21     !$omp parallel do private(tmp1) reduction(+:tmp)
S-22     do i=1,N
S-23        tmp1 = 0.0e0
S-24        !$omp simd reduction(+:tmp1)
S-25        do k = 1,M
S-26           tmp1 = tmp1 + P(k,i)
S-27        end do
S-28        tmp = tmp + tmp1
S-29     end do
S-30     !$omp end target
S-31  end function
S-32
S-33  ! Note:  The variable tmp is now mapped with tofrom, for correct
S-34  ! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
```

Fortran

6.14.6 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 (**omp_5.1**)

```
S-1   #define N 100000000
S-2
S-3   float  sp[N], sv1[N], sv2[N];
S-4   double dp[N], dv1[N], dv2[N];
S-5   #pragma omp declare target link(sp,sv1,sv2) \
S-6                              link(dp,dv1,dv2)
S-7
S-8   void s_init(float *, float *, int);
S-9   void d_init(double *, double *, int);
S-10  void s_output(float *, int);
S-11  void d_output(double *, int);
S-12
S-13  #pragma omp begin declare target
S-14
S-15  void s_vec_mult_accum()
S-16  {
S-17     int i;
S-18
S-19     #pragma omp parallel for
S-20     for (i=0; i<N; i++)
S-21       sp[i] = sv1[i] * sv2[i];
S-22  }
S-23
S-24  void d_vec_mult_accum()
S-25  {
S-26     int i;
S-27
```

```
S-28        #pragma omp parallel for
S-29        for (i=0; i<N; i++)
S-30          dp[i] = dv1[i] * dv2[i];
S-31     }
S-32     #pragma omp end declare target
S-33
S-34     int main()
S-35     {
S-36        s_init(sv1, sv2, N);
S-37        #pragma omp target map(to:sv1,sv2) map(from:sp)
S-38           s_vec_mult_accum();
S-39        s_output(sp, N);
S-40
S-41        d_init(dv1, dv2, N);
S-42        #pragma omp target map(to:dv1,dv2) map(from:dp)
S-43           d_vec_mult_accum();
S-44        d_output(dp, N);
S-45
S-46      return 0;
S-47     }
```

C / C++

Fortran

1 *Example declare_target.6.f90* (`omp_4.5`)

```
 S-1     module m_dat
 S-2        integer, parameter :: N=100000000
 S-3        !$omp declare target link(sp,sv1,sv2)
 S-4        real :: sp(N), sv1(N), sv2(N)
 S-5
 S-6        !$omp declare target link(dp,dv1,dv2)
 S-7        double precision :: dp(N), dv1(N), dv2(N)
 S-8
 S-9     contains
S-10        subroutine s_vec_mult_accum()
S-11        !$omp declare target
S-12           integer :: i
S-13
S-14           !$omp parallel do
S-15           do i = 1,N
S-16             sp(i) = sv1(i) * sv2(i)
S-17           end do
S-18
S-19        end subroutine s_vec_mult_accum
S-20
S-21        subroutine d_vec_mult_accum()
S-22        !$omp declare target
```

```
S-23          integer :: i
S-24
S-25          !$omp parallel do
S-26          do i = 1,N
S-27            dp(i) = dv1(i) * dv2(i)
S-28          end do
S-29
S-30       end subroutine
S-31    end module m_dat
S-32
S-33    program prec_vec_mult
S-34       use m_dat
S-35
S-36       call s_init(sv1, sv2, N)
S-37       !$omp target map(to:sv1,sv2) map(from:sp)
S-38         call s_vec_mult_accum()
S-39       !$omp end target
S-40       call s_output(sp, N)
S-41
S-42       call d_init(dv1, dv2, N)
S-43       !$omp target map(to:dv1,dv2) map(from:dp)
S-44         call d_vec_mult_accum()
S-45       !$omp end target
S-46       call d_output(dp, N)
S-47
S-48    end program
```

Fortran

6.14.7 Declare Target Directive with `device_type` Clause

The **`declare target`** directives apply to procedures to ensure that they can be executed or accessed on a device. The **`device_type`** clause specifies whether a version of the procedure or variable should be made available on the host, device or both. This example uses **`nohost`** for a procedure *`foo()`*. Only a device version of the procedure *`foo()`* is made available. If the variant function *`foo_onhost()`* is not specified for the host fallback execution, the call to *`foo()`* from the **`target`** region will result in a link time error due to the code generated for host execution of the target region. This is because host symbol for the device routine *`foo()`* marked as **`nohost`** is not required to be present in the host environment.

C / C++

Example declare_target.7.c (**omp_5.2**)

```
S-1   #include <stdio.h>
S-2
S-3   void foo();
S-4   void foo_onhost();
S-5
S-6   #pragma omp declare target enter(foo) device_type(nohost)
S-7
S-8   #pragma omp declare variant(foo_onhost) match(device={kind(host)})
S-9   void foo(){
S-10    //device specific computation
S-11  }
S-12
S-13  void foo_onhost(){
S-14    printf("On host\n");
S-15  }
S-16
S-17  int main(){
S-18    #pragma omp target teams
S-19    {
S-20      foo(); //calls foo() on target device or
S-21             //foo_onhost() in case of host fallback
S-22    }
S-23    return 0;
S-24
S-25  }
```

C / C++

Fortran

Example declare_target.7.f90 (**omp_5.2**)

```
S-1   module subs
S-2
S-3   contains
S-4      subroutine foo()
S-5      !$omp declare target enter(foo) device_type(nohost)
S-6      !$omp declare variant(foo_onhost) match(device={kind(host)})
S-7        ! device specific computation
S-8      end subroutine
S-9
S-10     subroutine foo_onhost()
S-11       print *,' On host.'
S-12     end subroutine
S-13
S-14  end module
```

```
S-15
S-16    program main
S-17
S-18       use subs
S-19       !$omp target
S-20       call foo    !calls foo() on device or
S-21                   !foo_onhost() in case of host fallback
S-22       !$omp end target
S-23
S-24    end program
```

Fortran

C++

6.15 Lambda Expressions

The following example illustrates the usage of lambda expressions and their corresponding closure objects within a **target** region.

In Case 1, a lambda expression is defined inside a **target** construct that implicitly maps the structure *s*. Inside the construct, the lambda captures (by reference) the corresponding *s*, and the resulting closure object is assigned to *lambda1*. When the call operator is invoked on *lambda1*, the captured reference to *s* is used in the call. The modified *s* is then copied back to the host device on exit from the **target** construct.

In Case 2, a lambda expression is instead defined before the **target** construct and captures (by copy) the pointer *sp*. A **target data** construct is used to first map the structure, and then the **target** construct implicitly maps the closure object referenced by *lambda2*, a zero-length array section based on the structure pointer *sp*, and a zero-length array section based on the captured pointer in the closure object. The implicit maps result in attached pointers to the corresponding structure. The call for *lambda2* inside the **target** construct will access *sp->a* and *sp->b* from the corresponding structure.

Case 3 is similar to Case 2, except *s* is instead captured by reference by the lambda expression. As for Case 2, the structure is first mapped by an enclosing **target data** construct, and then the **target** construct implicitly maps *s* and the closure object referenced by *lambda3*. The effect of the map is to make the the call for *lambda3* refer to the corresponding *s* inside the **target** construct rather than the original *s*.

In Case 4, the program defines a static variable *ss* of the same structure type as *s*. While the body of the lambda expression refers to *ss*, it is not captured. In order for *lambda4* to be callable in the **target** region, the reference to *ss* should be to a device copy of *ss* that also has static storage. This is achieved with the use of the **declare target** directive. Inside the **target** construct, all references to *ss*, including in the *lambda4* call, will refer to the corresponding *ss* that results

from the **declare target** directive. The **always** modifier is used on the **map** clause to transfer the updated values for the structure back to the host device.

Example lambda_expressions.1.cpp (**omp_5.0**)

```
S-1   #include <iostream>
S-2   using namespace std;
S-3
S-4   struct S { int a; int b; };
S-5
S-6   int main()
S-7   {
S-8
S-9   // CASE 1 Lambda defined in target region
S-10
S-11     S s = S {0,1};
S-12
S-13     #pragma omp target
S-14     {
S-15        auto lambda1 = [&s]() { s.a = s.b * 2; };
S-16        s.b += 2;
S-17        lambda1(); // s.a = 3 * 2
S-18     }
S-19     cout << s.a << " " << s.b << endl; //OUT 6 3
S-20
S-21  // CASE 2 Host defined lambda, Capture pointer to s
S-22
S-23     s = {0,1};
S-24     S *sp = &s;
S-25     auto lambda2 = [sp]() {sp->a = sp->b * 2; };
S-26
S-27     // closure object's sp attaches to corresponding s on target
S-28     // construct
S-29     #pragma omp target data map(sp[0])
S-30     #pragma omp target
S-31     {
S-32        sp->b += 2;
S-33        lambda2();
S-34     }
S-35     cout << s.a << " " << s.b << endl; //OUT 6 3
S-36
S-37  // CASE 3 Host defined lambda, Capture s by reference
S-38
S-39     s = {0,1};
S-40     auto lambda3 = [&s]() {s.a = s.b * 2; };
```

```
S-41
S-42        // closure object's s refers to corresponding s in target
S-43        // construct
S-44        #pragma omp target data map(s)
S-45        #pragma omp target
S-46        {
S-47           s.b += 2;
S-48           lambda3();
S-49        }
S-50        cout << s.a << " " << s.b << endl; //OUT 6 3
S-51
S-52     // CASE 4 Host defined lambda, references static variable
S-53
S-54        static S ss = {0,1};
S-55        #pragma omp declare target enter(ss)
S-56        auto lambda4 = [&]() {ss.a = ss.b * 2; };
S-57
S-58        #pragma omp target map(always,from:ss)
S-59        {
S-60           ss.b += 2;
S-61           lambda4();
S-62        }
S-63        cout << ss.a << " " << ss.b << endl; //OUT 6 3
S-64
S-65        return 0;
S-66     }
```

C++

6.16 `teams` Construct and Related Combined Constructs

6.16.1 `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 primary thread of each team executes the **`teams`** region.

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

C / C++

Example teams.1.c (**omp_4.0**)

```
#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 execution with 4.5 (and pre-4.5) compliant compilers.
          See Devices Intro.
 */
```

C / C++

Fortran

Example teams.1.f90 (`omp_4.0`)

```
S-1   function dotprod(B,C,N) result(sum)
S-2   use omp_lib, ONLY : omp_get_num_teams, omp_get_team_num
S-3       real    :: B(N), C(N), sum,sum0, sum1
S-4       integer :: N, i
S-5       sum0 = 0.0e0
S-6       sum1 = 0.0e0
S-7       !$omp target map(to: B, C) map(tofrom: sum0, sum1)
S-8       !$omp teams num_teams(2)
S-9         if (omp_get_num_teams() /= 2) stop "2 teams required"
S-10        if (omp_get_team_num() == 0) then
S-11           !$omp parallel do reduction(+:sum0)
S-12           do i=1,N/2
S-13              sum0 = sum0 + B(i) * C(i)
S-14           end do
S-15        else if (omp_get_team_num() == 1) then
S-16           !$omp parallel do reduction(+:sum1)
S-17           do i=N/2+1,N
S-18              sum1 = sum1 + B(i) * C(i)
S-19           end do
S-20        end if
S-21      !$omp end teams
S-22      !$omp end target
S-23      sum = sum0 + sum1
S-24  end function
S-25
S-26  ! Note:  The variables sum0,sum1 are now mapped with tofrom, for correct
S-27  ! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
```

Fortran

6.16.2 `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 primary thread of each team executes the **`teams`** region. The **`distribute`** construct schedules the subsequent loop iterations across the primary 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 primary threads of each team.

When a team's primary 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 primary 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 primary 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 primary thread's private copy of *sum* created by the **teams** construct. At the end of the **teams** region, each primary thread's private copy of *sum* is reduced into the final *sum* that is implicitly mapped into the **target** region.

C / C++

Example teams.2.c (**omp_4.0**)

```
S-1   #define min(x, y) (((x) < (y)) ? (x) : (y))
S-2
S-3   float dotprod(float B[], float C[], int N, int block_size,
S-4     int num_teams, int block_threads)
S-5   {
S-6       float sum = 0.0;
S-7       int i, i0;
S-8       #pragma omp target map(to: B[0:N], C[0:N]) map(tofrom: sum)
S-9       #pragma omp teams num_teams(num_teams) thread_limit(block_threads) \
S-10        reduction(+:sum)
S-11      #pragma omp distribute
S-12      for (i0=0; i0<N; i0 += block_size)
S-13         #pragma omp parallel for reduction(+:sum)
S-14         for (i=i0; i< min(i0+block_size,N); i++)
S-15             sum += B[i] * C[i];
S-16      return sum;
S-17  }
S-18  /* Note:  The variable sum is now mapped with tofrom, for correct
S-19     execution with 4.5 (and pre-4.5) compliant compilers. See
S-20     Devices Intro.
S-21   */
```

C / C++

Fortran

Example teams.2.f90 (`omp_4.0`)

```
S-1   function dotprod(B,C,N, block_size, num_teams, block_threads) result(sum)
S-2   implicit none
S-3       real    :: B(N), C(N), sum
S-4       integer :: N, block_size, num_teams, block_threads, i, i0
S-5       sum = 0.0e0
S-6       !$omp target map(to: B, C) map(tofrom: sum)
S-7       !$omp teams num_teams(num_teams) thread_limit(block_threads) &
S-8       !$omp&  reduction(+:sum)
S-9       !$omp distribute
S-10         do i0=1,N, block_size
S-11            !$omp parallel do reduction(+:sum)
S-12            do i = i0, min(i0+block_size,N)
S-13               sum = sum + B(i) * C(i)
S-14            end do
S-15         end do
S-16      !$omp end teams
S-17      !$omp end target
S-18  end function
S-19
S-20  ! Note:  The variable sum is now mapped with tofrom, for correct
S-21  ! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
```

Fortran

6.16.3 `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 primary thread of each team executes the **`teams`** region.

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

C / C++

1 *Example teams.3.c* (**omp_4.5**)

```
S-1   float dotprod(float B[], float C[], int N)
S-2   {
S-3      float sum = 0;
S-4      int i;
S-5      #pragma omp target teams map(to: B[0:N], C[0:N]) \
S-6                               defaultmap(tofrom:scalar) reduction(+:sum)
S-7      #pragma omp distribute parallel for reduction(+:sum)
S-8      for (i=0; i<N; i++)
S-9         sum += B[i] * C[i];
S-10     return sum;
S-11  }
S-12
S-13  /* Note:  The variable sum is now mapped with tofrom from the defaultmap
S-14           clause on the combined target teams construct, for correct
S-15           execution with 4.5 (and pre-4.5) compliant compilers.
S-16           See Devices Intro.
S-17   */
```

C / C++

Fortran

2 *Example teams.3.f90* (**omp_4.5**)

```
S-1   function dotprod(B,C,N) result(sum)
S-2      real    :: B(N), C(N), sum
S-3      integer :: N, i
S-4      sum = 0.0e0
S-5      !$omp target teams map(to: B, C)  &
S-6      !$omp&             defaultmap(tofrom:scalar) reduction(+:sum)
S-7      !$omp distribute parallel do reduction(+:sum)
S-8         do i = 1,N
S-9            sum = sum + B(i) * C(i)
S-10        end do
S-11     !$omp end target teams
S-12  end function
S-13
S-14  ! Note:  The variable sum is now mapped with tofrom from the defaultmap
S-15  !  clause on the combined target teams construct, for correct
S-16  !  execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
```

Fortran

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

The following example shows how the **`target teams`** and **`distribute parallel`** constructs are used to execute a **`target`** region. The **`teams`** construct creates a league of at most eight teams where the primary 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 primary 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 primary thread of each team in chunks of *1024* iterations.

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

C / C++

Example teams.4.c (**`omp_4.0`**)

```
S-1   #define N 1024*1024
S-2   float dotprod(float B[], float C[])
S-3   {
S-4       float sum = 0.0;
S-5       int i;
S-6       #pragma omp target map(to: B[0:N], C[0:N]) map(tofrom: sum)
S-7       #pragma omp teams num_teams(8) thread_limit(16) reduction(+:sum)
S-8       #pragma omp distribute parallel for reduction(+:sum) \
S-9                   dist_schedule(static, 1024) schedule(static, 64)
S-10      for (i=0; i<N; i++)
S-11          sum += B[i] * C[i];
S-12      return sum;
S-13  }
S-14
S-15  /* Note:  The variable sum is now mapped with tofrom, for correct
S-16            execution with 4.5 (and pre-4.5) compliant compilers.
S-17            See Devices Intro.
S-18   */
```

C / C++

Fortran

Example teams.4.f90 (`omp_4.0`)

```
S-1   module arrays
S-2   integer,parameter :: N=1024*1024
S-3   real :: B(N), C(N)
S-4   end module
S-5   function dotprod() result(sum)
S-6   use arrays
S-7      real    :: sum
S-8      integer :: i
S-9      sum = 0.0e0
S-10     !$omp target map(to: B, C) map(tofrom: sum)
S-11     !$omp teams num_teams(8) thread_limit(16) reduction(+:sum)
S-12     !$omp distribute parallel do reduction(+:sum) &
S-13     !$omp&  dist_schedule(static, 1024) schedule(static, 64)
S-14        do i = 1,N
S-15           sum = sum + B(i) * C(i)
S-16        end do
S-17     !$omp end teams
S-18     !$omp end target
S-19  end function
S-20
S-21  ! Note:  The variable sum is now mapped with tofrom, for correct
S-22  ! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
```

Fortran

6.16.5 `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 primary thread of each team executes the **`teams`** region.

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

C / C++

1 *Example teams.5.c* (`omp_4.0`)

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

C / C++

Fortran

2 *Example teams.5.f90* (`omp_4.0`)

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

Fortran

6.16.6 `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 primary thread of each team executes the **`teams`** region.

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

C / C++

Example teams.6.c (**`omp_4.0`**)

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

C / C++

Fortran

Example teams.6.f90 (**`omp_4.0`**)

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

Fortran

6.16.7 Evaluation of `num_teams` Clause that Appears inside `target` Region

The following example shows the evaluation of the **num_teams** clause when the **teams** construct is closely nested inside **target** construct. The code is non-conforming since value of x for the clause may be different from different devices. As of OpenMP 6.0, it is the user's responsibility to ensure identical values for the clause expression for nested as well as combined directive cases for **target** and **teams** constructs. This permits implementations to evaluate the **num_teams** argument on the host rather than the target device. For the program to be conforming, the program must update the host value so that x will have the same value when evaluated on the host or target device.

C / C++

Example teams.7.c (**omp_6.0**)

```
S-1   #include<stdio.h>
S-2   #include<omp.h>
S-3
S-4   int x;
S-5   #pragma omp declare target local(x)
S-6
S-7   int main() {
S-8     x = 128;
S-9     #pragma omp target
S-10    x = 256;
S-11
S-12    #pragma omp target
S-13    #pragma omp teams num_teams(x) // Undefined behavior due to value of "x"
S-14    if (omp_get_team_num() == 0){
S-15      printf("%d\n", omp_get_num_teams());
S-16    }
S-17
S-18    return 0;
S-19  }
S-20
```

C / C++

Fortran

Example teams.7.f90 (`omp_6.0`)

```
S-1   PROGRAM main
S-2     USE omp_lib
S-3     INTEGER :: x
S-4     !$OMP DECLARE TARGET LOCAL(x)
S-5
S-6      x = 128
S-7
S-8     !$OMP TARGET
S-9      x = 256
S-10    !$OMP END TARGET
S-11
S-12    !$OMP TARGET
S-13    !$OMP TEAMS NUM_TEAMS(x)        ! Undefined behavior due to value of 'x'
S-14    IF (omp_get_team_num() == 0) THEN
S-15      PRINT *, omp_get_num_teams()
S-16    END IF
S-17    !$OMP END TEAMS
S-18    !$OMP END TARGET
S-19
S-20  END PROGRAM main
```

Fortran

6.17 Asynchronous `target` Execution and Dependences

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 examples with explicit tasks.

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

6.17.1 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 / C++

Example async_target.1.c (**omp_5.1**)

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

C / C++

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

Fortran

Example async_target.1.f90 (**omp_4.0**)

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

Fortran

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

C / C++

Example async_target.2.c (**omp_5.1**)

```
S-1   #include <stdlib.h>
S-2   #include <omp.h>
S-3
S-4   #pragma omp begin declare target
S-5   extern void init(float *, float *, int);
S-6   #pragma omp end declare target
S-7
S-8   extern void foo();
S-9   extern void output(float *, int);
S-10  void vec_mult(float *p, int N, int dev)
S-11  {
S-12     float *v1, *v2;
S-13     int i;
S-14     #pragma omp task shared(v1, v2) depend(out: v1, v2)
S-15     #pragma omp target device(dev) map(v1, v2)
S-16     {
S-17         // check whether on device dev
S-18         if (omp_is_initial_device())
S-19            abort();
S-20         v1 = (float *)malloc(N*sizeof(float));
S-21         v2 = (float *)malloc(N*sizeof(float));
S-22         init(v1, v2, N);
S-23     }
S-24     foo(); // execute other work asychronously
S-25     #pragma omp task shared(v1, v2, p) depend(in: v1, v2)
S-26     #pragma omp target device(dev) map(to: v1, v2) map(from: p[0:N])
S-27     {
S-28         // check whether on device dev
S-29         if (omp_is_initial_device())
S-30            abort();
S-31         #pragma omp parallel for
S-32         for (i=0; i<N; i++)
S-33           p[i] = v1[i] * v2[i];
S-34         free(v1);
S-35         free(v2);
S-36     }
S-37     #pragma omp taskwait
S-38     output(p, N);
S-39  }
```

C / C++

The Fortran example below is similar to the C version above. Instead of pointers, though, it uses the convenience of Fortran allocatable arrays on the device. 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

Example async_target.2.f90 (**omp_4.0**)

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

```
S-23            if( omp_is_initial_device() ) &
S-24               stop "not executing on target device"
S-25            !$omp parallel do
S-26               do i = 1,N
S-27                  p(i) = v1(i) * v2(i)
S-28               end do
S-29            deallocate(v1,v2)
S-30
S-31         !$omp end target
S-32      !$omp end task
S-33
S-34      !$omp taskwait
S-35
S-36      !$omp end target data
S-37
S-38      call output(p, N)
S-39
S-40   end subroutine
```

Fortran

6.17.2 `nowait` Clause on `target` Construct

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 primary 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 target execution, **`static`** scheduling may be desired.

C / C++

Example async_target.3.c (`omp_5.1`)

```
S-1   #include <stdio.h>
S-2
S-3   #define N 1000000      //N must be even
S-4   void init(int n, float *v1, float *v2);
S-5
S-6   int main(){
S-7      int   i, n=N;
S-8      int   chunk=1000;
S-9      float v1[N],v2[N],vxv[N];
S-10
S-11     init(n, v1,v2);
S-12
S-13     #pragma omp parallel
S-14     {
S-15
S-16        #pragma omp masked
S-17        #pragma omp target teams distribute parallel for nowait \
S-18                                map(to: v1[0:n/2]) \
S-19                                map(to: v2[0:n/2]) \
S-20                                map(from: vxv[0:n/2])
S-21        for(i=0; i<n/2; i++){ vxv[i] = v1[i]*v2[i]; }
S-22
S-23        #pragma omp for schedule(dynamic,chunk)
S-24        for(i=n/2; i<n; i++){ vxv[i] = v1[i]*v2[i]; }
S-25
S-26     }
S-27     printf(" vxv[0] vxv[n-1] %f %f\n", vxv[0], vxv[n-1]);
S-28     return 0;
S-29  }
```

C / C++

Fortran

Example async_target.3.f90 (`omp_5.1`)

```
S-1   program concurrent_async
S-2      use omp_lib
S-3      integer,parameter :: n=1000000  !!n must be even
S-4      integer           :: i, chunk=1000
S-5      real              :: v1(n),v2(n),vxv(n)
S-6
S-7      call init(n, v1,v2)
S-8
S-9      !$omp parallel
S-10
```

```
        !$omp masked
        !$omp target teams distribute parallel do nowait &
        !$omp&                    map(to: v1(1:n/2))   &
        !$omp&                    map(to: v2(1:n/2))   &
        !$omp&                    map(from: vxv(1:n/2))
        do i = 1,n/2;    vxv(i) = v1(i)*v2(i); end do
        !$omp end masked

        !$omp do schedule(dynamic,chunk)
        do i = n/2+1,n;  vxv(i) = v1(i)*v2(i); end do

    !$omp end parallel

    print*, " vxv(1) vxv(n) :", vxv(1), vxv(n)

end program
```

Fortran

6.17.3 Asynchronous `target` with `nowait` and `depend` Clauses

More details on dependences can be found in Section 5.3 on page 103, 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.

C / C++

Example async_target.4.c (**omp_4.5**)

```
extern void init(  float*, int);
extern void output(float*, int);

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)
          #pragma omp parallel for private(i)
          for (i=0; i<N; i++)
             p[i] = v1[i] * v2[i];

          #pragma omp task depend(in:p)
          output(p, N);
       }
    }
}
```

C / C++

Fortran

Example async_target.4.f90 (`omp_4.5`)

```
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

         !$omp task depend(out:v2)
         call init(v2, N)
         !$omp end task

         !$omp target nowait depend(in:v1,v2) depend(out:p) &
         !$omp&                 map(to:v1,v2)  map(from: p)
         !$omp parallel do
         do i=1,N
```

```
S-23                p(i) = v1(i) * v2(i)
S-24              end do
S-25              !$omp end target
S-26
S-27
S-28              !$omp task depend(in:p)
S-29              call output(p, N)
S-30              !$omp end task
S-31
S-32            !$omp end single
S-33          !$omp end parallel
S-34
S-35          deallocate( p, v1, v2 )
S-36
S-37    end subroutine
```

Fortran

6.17.4 Conditionally Asynchronous `target` Using the `nowait` Clause

In OpenMP 6.0, **`nowait`** takes an OpenMP logical type argument to specify if the generated *task* is an included task or a deferred task. In the following example, the **`nowait`** clause is used with an argument on the **`target`** directive. In a practical situation, the value of *`is_deferred`* can be chosen based on the time taken for some work on host or device that can be performed asynchronously after the target task is scheduled. If the target task is deferred, it must be synchronized by a **`taskwait`** before the value of *x* is used. Prior to 6.0, the same effect would require the use of a *metadirective* or an **`if-else`** statement that duplicates the **`target`** construct.

C / C++

Example async_target.5.c (**`omp_6.0`**)

```
S-1     #include<stdio.h>
S-2     #include<stdlib.h>
S-3     #include<time.h>
S-4
S-5     #pragma omp begin declare target
S-6     void update(int* num) {
S-7
S-8         *num = (*num) * 3;
S-9     }
S-10    #pragma omp end declare target
S-11
S-12    int main(int argc, char*argv[]){
S-13      int x = 2 ;
```

```
  int is_deferred = time(NULL) % 2;

  #pragma omp target nowait(is_deferred) map(tofrom: x)
  {
    update(&x);
  }

  // Perform other tasks in parallel while the
  // target region is executing

  if(is_deferred){
    #pragma omp taskwait
  }

  if( x == 6){
    printf("Passed\n");
    return 0;
  } else {
    printf("Failed\n");
    return 1;
  }
}
```

C / C++

Fortran

Example async_target.5.f90 (`omp_6.0`)

```
program async_target_nowait_arg
  implicit none
  integer :: x
  logical :: is_deferred
  real :: rand_no

  x = 2
  ! Determine if computation is deferred
  call  random_number(rand_no)
  is_deferred=mod(int(rand_no*10), 2) == 1

  !$omp target map(tofrom: x) nowait(is_deferred)
  call update(x)
  !$omp end target

  ! Perform other tasks in parallel while the target region is executing

  if (is_deferred) then
    !$omp taskwait
  endif
```

```
S-21
S-22      if (x == 6) then
S-23        stop  "Passed"
S-24      else
S-25        error stop  "Failed"
S-26      endif
S-27
S-28    contains
S-29
S-30      subroutine update(num)
S-31        integer, intent(inout) :: num
S-32        !$omp declare target
S-33        num = num * 3
S-34      end subroutine update
S-35
S-36    end program async_target_nowait_arg
```

Fortran

6.18 Device Routines

6.18.1 `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 (**`omp_5.1`**)

```
S-1     #include <stdio.h>
S-2     #include <omp.h>
S-3
S-4     #pragma omp begin declare target
S-5        void vec_mult(float *p, float *v1, float *v2, int N);
S-6        extern float *p, *v1, *v2;
S-7        extern int N;
S-8     #pragma omp end declare target
S-9
S-10    extern void init_vars(float *, float *, int);
S-11    extern void output(float *, int);
S-12
S-13    void foo()
S-14    {
S-15       init_vars(v1, v2, N);
```

```
S-16        #pragma omp target device(42) map(p[:N], v1[:N], v2[:N])
S-17        {
S-18           vec_mult(p, v1, v2, N);
S-19        }
S-20        output(p, N);
S-21     }
S-22
S-23     void vec_mult(float *p, float *v1, float *v2, int N)
S-24     {
S-25        int i;
S-26        int nthreads;
S-27        if (!omp_is_initial_device())
S-28        {
S-29           printf("1024 threads on target device\n");
S-30           nthreads = 1024;
S-31        }
S-32        else
S-33        {
S-34           printf("8 threads on initial device\n");
S-35           nthreads = 8;
S-36        }
S-37        #pragma omp parallel for private(i) num_threads(nthreads)
S-38        for (i=0; i<N; i++)
S-39          p[i] = v1[i] * v2[i];
S-40     }
```

C / C++

Fortran

1 *Example device.1.f90* (**omp_4.0**)

```
S-1      module params
S-2         integer,parameter :: N=1024
S-3      end module params
S-4      module vmult
S-5      contains
S-6         subroutine vec_mult(p, v1, v2, N)
S-7         use omp_lib, ONLY : omp_is_initial_device
S-8         !$omp declare target
S-9         real    :: p(N), v1(N), v2(N)
S-10        integer :: i, nthreads, N
S-11           if (.not. omp_is_initial_device()) then
S-12              print*, "1024 threads on target device"
S-13              nthreads = 1024
S-14           else
S-15              print*, "8 threads on initial device"
S-16              nthreads = 8
S-17           endif
```

```
S-18          !$omp parallel do private(i) num_threads(nthreads)
S-19          do i = 1,N
S-20            p(i) = v1(i) * v2(i)
S-21          end do
S-22       end subroutine vec_mult
S-23    end module vmult
S-24    program prog_vec_mult
S-25    use params
S-26    use vmult
S-27    real :: p(N), v1(N), v2(N)
S-28       call init(v1,v2,N)
S-29       !$omp target device(42) map(p, v1, v2)
S-30          call vec_mult(p, v1, v2, N)
S-31       !$omp end target
S-32       call output(p, N)
S-33    end program
```

Fortran

6.18.2 omp_get_num_devices Routine

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

C / C++

Example device.2.c (**omp_4.0**)

```
S-1     #include <omp.h>
S-2     extern void init(float *, float *, int);
S-3     extern void output(float *, int);
S-4     void vec_mult(float *p, float *v1, float *v2, int N)
S-5     {
S-6        int i;
S-7        init(v1, v2, N);
S-8        int ndev = omp_get_num_devices();
S-9        int do_offload = (ndev>0 && N>1000000);
S-10       #pragma omp target if(do_offload) \
S-11                          map(to: v1[0:N], v2[:N]) \
S-12                          map(from: p[0:N])
S-13       #pragma omp parallel for if(N>1000) private(i)
S-14       for (i=0; i<N; i++)
S-15         p[i] = v1[i] * v2[i];
S-16       output(p, N);
S-17    }
```

C / C++

Fortran

Example device.2.f90 (`omp_4.0`)

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

Fortran

6.18.3 `omp_set_default_device` and `omp_get_default_device` Routines

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

C / C++

Example device.3.c (`omp_4.0`)

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

C / C++

Fortran

Example device.3.f90 (**omp_4.0**)

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

Fortran

6.18.4 Device and Host Memory Association

The association of device memory with host memory can be established by calling the **omp_target_associate_ptr** API routine as part of the mapping. The following example shows the use of this routine to associate device memory of size *CS*, allocated by the **omp_target_alloc** routine and pointed to by the device pointer *dev_ptr*, with a chunk of the host array *arr* starting at index *ioff*. In Fortran, the intrinsic function c_loc is called to obtain the corresponding C pointer (*h_ptr*) of *arr(ioff)* for use in the call to the API routine.

Since the reference count of the resulting mapping is infinite, it is necessary to use the **target update** directive (or the **always** modifier in a **map** clause) to accomplish a data transfer between host and device. The explicit mapping of the array section *arr[ioff:CS]* (or *arr(ioff:ioff+CS-1)* in Fortran) on the **target** construct ensures that the allocated and associated device memory is used when referencing the array *arr* in the **target** region. The device pointer *dev_ptr* cannot be accessed directly after a call to the **omp_target_associate_ptr** routine.

After the **target** region, the device pointer is disassociated from the current chunk of the host memory by calling the **omp_target_disassociate_ptr** routine before working on the next chunk. The device memory is freed by calling the **omp_target_free** routine at the end.

C / C++

Example target_associate_ptr.1.c (**omp_4.5**)

```
S-1   #include <stdio.h>
S-2   #include <omp.h>
S-3
S-4   #define CS 50
S-5   #define N  (CS*2)
S-6
S-7   int main() {
S-8     int arr[N];
S-9     int *dev_ptr;
S-10    int dev;
S-11
S-12    for (int i = 0; i < N; i++)
S-13      arr[i] = i;
S-14
S-15    dev = omp_get_default_device();
S-16
S-17    // Allocate device memory
S-18    dev_ptr = (int *)omp_target_alloc(sizeof(int) * CS, dev);
S-19
S-20    // Loop over chunks
S-21    for (int ioff = 0; ioff < N; ioff += CS) {
S-22
S-23      // Associate device memory with one chunk of host memory
S-24      omp_target_associate_ptr(&arr[ioff], dev_ptr,
S-25                               sizeof(int) * CS, 0, dev);
S-26
S-27      printf("before: arr[%d]=%d\n", ioff, arr[ioff]);
S-28
S-29      // Update the device data
S-30      #pragma omp target update to(arr[ioff:CS]) device(dev)
S-31
S-32      // Explicit mapping of arr to make sure that we use the allocated
S-33      // and associated memory.  No host-device data update here.
S-34      #pragma omp target map(tofrom : arr[ioff:CS]) device(dev)
S-35        for (int i = 0; i < CS; i++) {
S-36          arr[i+ioff]++;
S-37        }
S-38
S-39      // Update the host data
S-40      #pragma omp target update from(arr[ioff:CS]) device(dev)
S-41
S-42      printf("after: arr[%d]=%d\n", ioff, arr[ioff]);
S-43
S-44      // Disassociate device pointer from the current chunk of host memory
```

```
S-45        // before next use
S-46        omp_target_disassociate_ptr(&arr[ioff], dev);
S-47      }
S-48
S-49      // Free device memory
S-50      omp_target_free(dev_ptr, dev);
S-51
S-52      return 0;
S-53    }
S-54    /* Outputs:
S-55      before: arr[0]=0
S-56      after: arr[0]=1
S-57      before: arr[50]=50
S-58      after: arr[50]=51
S-59     */
```

C / C++

Fortran

1 *Example target_associate_ptr.1.f90* (**omp_5.1**)

```
S-1     program target_associate
S-2       use omp_lib
S-3       use, intrinsic :: iso_c_binding
S-4       implicit none
S-5
S-6       integer, parameter :: CS = 50
S-7       integer, parameter :: N  = CS*2
S-8       integer, target :: arr(N)
S-9       type(c_ptr) :: h_ptr, dev_ptr
S-10      integer(c_size_t) :: csize, dev_off
S-11      integer(c_int) :: dev
S-12      integer :: i, ioff, s
S-13
S-14      do i = 1, N
S-15        arr(i) = i
S-16      end do
S-17
S-18      dev = omp_get_default_device()
S-19      csize = c_sizeof(arr(1)) * CS
S-20
S-21      ! Allocate device memory
S-22      dev_ptr = omp_target_alloc(csize, dev)
S-23      dev_off = 0
S-24
S-25      ! Loop over chunks
S-26      do ioff = 1, N, CS
S-27
```

```
      ! Associate device memory with one chunk of host memory
      h_ptr = c_loc(arr(ioff))
      s = omp_target_associate_ptr(h_ptr, dev_ptr, csize, dev_off, dev)

      print *, "before: arr(", ioff, ")=", arr(ioff)

      ! Update the device data
      !$omp target update to(arr(ioff:ioff+CS-1)) device(dev)

      ! Explicit mapping of arr to make sure that we use the allocated
      ! and associated memory.  No host-device data update here.
      !$omp target map(tofrom: arr(ioff:ioff+CS-1)) device(dev)
        do i = 0, CS-1
          arr(i+ioff) = arr(i+ioff) + 1
        end do
      !$omp end target

      ! Update the host data
      !$omp target update from(arr(ioff:ioff+CS-1)) device(dev)

      print *, "after: arr(", ioff, ")=", arr(ioff)

      ! Disassociate device pointer from the current chunk of host memory
      ! before next use
      s = omp_target_disassociate_ptr(h_ptr, dev)
    end do

    ! Free device memory
    call omp_target_free(dev_ptr, dev)

end
! Outputs:
!  before: arr( 1 )= 1
!  after: arr( 1 )= 2
!  before: arr( 51 )= 51
!  after: arr( 51 )= 52
```

Fortran

6.18.5 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.

C / C++

Example device.4.c (**omp_4.5**)

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

void get_dev_cos(double *mem, int 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 = (double *)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);
```

```
S-37
S-38        omp_target_free(mem_dev_cpy, t);
S-39    }
```

C / C++

The following Fortran example illustrates how to use the **omp_target_alloc** and **omp_target_memcpy** functions to directly allocate device storage and transfer data to and from a device. It also shows how to check for the presence of device data with the **omp_target_is_present** function and to associate host and device storage with the **omp_target_associate_ptr** function.

In Section 1 of the code, 40 bytes of storage are allocated on the default device with the **omp_target_alloc** function, which returns a value (of type **C_PTR**) that contains the device address of the storage. In the subsequent **target** construct, *cp* is specified on the **is_device_ptr** clause to instruct the compiler that *cp* is a device pointer. The device pointer (*cp*) is then associated with the Fortran pointer (*fp*) via the **c_f_pointer** routine inside the **target** construct. As a result, *fp* points to the storage on the device that is allocated by the **omp_target_alloc** routine. In the **target** region, the value 4 is assigned to the storage on the device, using the Fortran pointer. A trivial test checks that all values were correctly assigned. The Fortran pointer (*fp*) is nullified before the end of the **target** region. After the **target** construct, the space on the device is freed with the **omp_target_free** function, using the device *cp* pointer which is set to null after the call.

In Section 2, the content of the storage allocated on the host is directly copied to the OpenMP allocated storage on the device. First, storage is allocated for the device and host using **omp_target_alloc**. Next, on the host the device pointer, returned from the allocation **omp_target_alloc** function, is associated with a Fortran pointer, and values are assigned to the storage. Similarly, values are assigned on the device to the device storage, after associating a Fortran pointer (*fp_dst*) with the device's storage pointer (*cp_dst*).

Next the **omp_target_memcpy** function directly copies the host data to the device storage, specified by the respective host and device pointers. This copy will overwrite -1 values in the device storage, and is checked in the next **target** construct. Keyword arguments are used here for clarity. (A positional argument list is used in the next Section.)

In Section 3, space is allocated (with a Fortran **ALLOCATE** statement) and initialized using a host Fortran pointer (*h_fp*), and the address of the storage is directly assigned to a host C pointer (*h_cp*). The following **omp_target_is_present** function returns *0* (*false*, of **integer(C_INT)** type) to indicate that *h_cp* does not have any corresponding storage on the default device.

Next, the same amount of space is allocated on the default device with the **omp_target_alloc** function, which returns a device pointer (*d_cp*). The device pointer *d_cp* and host pointer *h_cp* are then associated using the **omp_target_associate_ptr** function. The device storage to which *d_cp* points becomes the corresponding storage of the host storage to which *h_cp* points.

The following **omp_target_is_present** call confirms this, by returning a non-zero value of **integer(C_INT)** type for true.

After the association, the content of the host storage is copied to the device using the **omp_target_memcpy** function. In the final **target** construct an array section of *h_fp* is mapped to the device, and evaluated for correctness. The mapping establishes a connection of *h_fp* with the corresponding device data in the **target** construct, but does not produce an update on the device because the previous **omp_target_associate_ptr** routine sets the reference count of the mapped object to infinity, meaning a mapping without the **always** modifier will not update the device object.

Fortran

Example device.4.f90 (**omp_5.0**)

```
S-1   program device_mem
S-2     use omp_lib
S-3     use, intrinsic            :: iso_c_binding
S-4
S-5     integer(kind=4),parameter :: N = 10
S-6     type(c_ptr)               :: cp
S-7     integer(c_int), pointer   :: fp(:)
S-8     integer(c_int)            :: rc, host_dev, targ_dev
S-9     integer(c_size_t)         :: int_bytes
S-10
S-11    integer, pointer  :: fp_src(:), fp_dst(:)     ! Section 2 vars
S-12    type(c_ptr)       :: cp_src,    cp_dst        ! Section 2 vars
S-13
S-14    integer, pointer  :: h_fp(:)                  ! Section 3 vars
S-15    type(c_ptr)       :: h_cp,    d_cp            ! Section 3 vars
S-16
S-17    integer :: i
S-18
S-19    host_dev  = omp_get_initial_device()
S-20    targ_dev  = omp_get_default_device()
S-21    int_bytes = C_SIZEOF(rc)
S-22
S-23  !-----------------------------------------------Section 1 vv-----------
S-24    cp = omp_target_alloc(N*int_bytes, targ_dev)
S-25
S-26    !$omp target is_device_ptr(cp) device(targ_dev) !fp implicit map
S-27       call c_f_pointer(cp, fp, [ N ])              !fp becomes associated
S-28       fp(:) = 4
S-29       if( all(fp == 4) ) print*,"PASSED 1 of 5"
S-30       nullify(fp)                  !fp must be returned as disassociated
S-31    !$omp end target
S-32
S-33    call omp_target_free(cp, targ_dev)
```

```
S-34      cp = c_null_ptr
S-35
S-36    !-----------------------------------------------Section 2 vv-----------
S-37
S-38       cp_src = omp_target_alloc((N+1)*int_bytes, host_dev)
S-39       cp_dst = omp_target_alloc(  N  *int_bytes, targ_dev)
S-40
S-41    !          Initialize host array (src)
S-42       call c_f_pointer(cp_src, fp_src, [N+1])
S-43       fp_src = [(i,i=1,N+1)]
S-44
S-45       !$omp target device(targ_dev) is_device_ptr(cp_dst)
S-46         call c_f_pointer(cp_dst, fp_dst, [N])   ! fp_dst becomes associated
S-47         fp_dst(:) = -1                          ! Initial device storage
S-48         nullify(fp_dst)                         ! return as disassociated
S-49       !$omp end target
S-50
S-51    !  Copy subset of host (src) array to device (dst) array
S-52       rc = omp_target_memcpy(                                          &
S-53                dst=cp_dst,             src=cp_src,    length=N*int_bytes, &
S-54                dst_offset=0_c_size_t,  src_offset=int_bytes,           &
S-55                dst_device_num=targ_dev,src_device_num=host_dev)
S-56
S-57    !  Check dst array on device
S-58
S-59       !$omp target device(targ_dev) is_device_ptr(cp_dst)
S-60         call c_f_pointer(cp_dst, fp_dst, [N])
S-61         if ( all(fp_dst == [(i,i=1,N)]) ) print*,"PASSED 2 of 5"
S-62         nullify(fp_dst)
S-63       !$omp end target
S-64
S-65    !-----------------------------------------------Section 3 vv-----------
S-66
S-67       !allocate host memory and initialize.
S-68       allocate(h_fp(N), source=[(i,i=1,N)])
S-69
S-70       h_cp = c_loc(h_fp)
S-71                ! Device is not aware of allocation on host
S-72       if(omp_target_is_present(h_cp, targ_dev) == 0) &
S-73          print*, "PASSED 3 of 5"
S-74
S-75                ! Allocate device memory
S-76       d_cp = omp_target_alloc(c_sizeof(h_fp(1))*size(h_fp), targ_dev)
S-77
S-78                ! now associate host and device storage
S-79       rc=omp_target_associate_ptr(h_cp,d_cp,c_sizeof(h_fp(1))*size(h_fp), &
S-80                                   0_c_size_t,targ_dev)
```

```
S-81
S-82                ! check presence of device data, associated w. host pointer
S-83        if(omp_target_is_present(h_cp, targ_dev) /= 0) &
S-84           print*,"PASSED 4 of 5"
S-85
S-86                ! copy from host to device via C pointers
S-87        rc=omp_target_memcpy(d_cp,       h_cp,c_sizeof(h_fp(1))*size(h_fp), &
S-88                             0_c_size_t, 0_c_size_t,                        &
S-89                             targ_dev,   host_dev)
S-90
S-91                ! validate the device data in the target region
S-92                ! no data copy here since the reference count is infinity
S-93        !$omp target device(targ_dev) map(h_fp)
S-94          if ( all(h_fp == [(i,i=1,N)]) ) print*, "PASSED 5 of 5"
S-95        !$omp end target
S-96
S-97        call omp_target_free(d_cp,targ_dev)
S-98        deallocate(h_fp)
S-99    end program
```

Fortran

The following example illustrates the use of the **omp_target_memcpy_async** routine to perform asynchronous memory copies. The routine acts as if it is a deferrable task so that a **taskwait** construct can be used to wait for the completion of the deferrable task. In the example the **omp_target_memcpy_async** routine copies host data (*h_buf*) to device (*d_buf*). The Fortran code uses the intrinsic **c_loc** function to get the corresponding C pointer (*c_hbuf*) for passing to the **omp_target_memcpy_async** routine. The last two arguments (*0* and *NULL*) to the routine indicate that there is no specified dependence associated with the call. The Fortran code omits the unused last argument.

C / C++

Example device.5.c (**omp_5.2**)

```
S-1     #include <stdio.h>
S-2     #include <stdlib.h>
S-3     #include <omp.h>
S-4
S-5     #define N 128
S-6     extern void do_work();
S-7
S-8     void async_memcpy() {
S-9       int h_dev = omp_get_initial_device();
S-10      int d_dev = omp_get_default_device();
S-11      size_t dsize;
S-12      float h_buf[N];
S-13      void *d_buf;
S-14      int i;
```

```
  /* allocate device memory */
  dsize = N * sizeof(float);
  d_buf = omp_target_alloc(dsize, d_dev);
  if (!d_buf)
    abort();

  /* set up host data */
  for (i = 0; i < N; i++) {
    h_buf[i] = i*0.1f;
  }

  /* copy data from host to device asynchronously */
  if (omp_target_memcpy_async(d_buf, h_buf, dsize, 0, 0,
                              d_dev, h_dev, 0, NULL))
    abort();

  /* do some work here at the same time */
  do_work();

  /* wait for task completion */
  #pragma omp taskwait

  omp_target_free(d_buf, d_dev);
}
```

C / C++

Fortran

Example device.5.f90 (`omp_5.2`)

```
subroutine async_memcpy
  use omp_lib
  use, intrinsic     :: iso_c_binding

  implicit none

  integer, parameter :: N = 128
  real, target       :: h_buf(N)
  type(c_ptr)        :: c_dbuf, c_hbuf
  integer(c_int)     :: d_dev, h_dev
  integer(c_size_t)  :: dsize

  integer :: i

  h_dev = omp_get_initial_device()
  d_dev = omp_get_default_device()
  dsize = N * c_sizeof(h_buf(1))
```

```
S-18
S-19      ! allocate device memory
S-20      c_dbuf = omp_target_alloc(dsize, d_dev)
S-21      if (.not.c_associated(c_dbuf)) stop
S-22      c_hbuf = c_loc(h_buf)
S-23
S-24      ! set up host data
S-25      h_buf = [(i*0.1, i = 1, N)]
S-26
S-27      ! copy data from host to device asynchronously
S-28      if (omp_target_memcpy_async(c_dbuf, c_hbuf, dsize, 0, 0, &
S-29                                  d_dev, h_dev, 0) /= 0) then
S-30        stop
S-31      endif
S-32
S-33      ! do some work here at the same time
S-34      call do_work
S-35
S-36      ! wait for task completion
S-37      !$omp taskwait
S-38
S-39      call omp_target_free(c_dbuf, d_dev)
S-40
S-41    end subroutine
```

Fortran

The following is a more complicated example that shows the use of the **omp_target_memcpy_async** routine with a depend object *obj* to overlap the memory copy with computation performed by *do_work*. The depend object *obj* was created by the **depobj** directive and initialized to an **out** dependence on the data *d_buf[0:N]* (or *d_buf(1:N)* for Fortran) in advance. The **depend(depobj:** *obj***)** (or alternatively **depend(in:** *d_buf[0:N]***)**) clause on the **target** construct ensures the asynchronous memory copy is complete before the data *d_buf* can be used in the **target** region.

C / C++

Example device.6.c (**omp_5.2**)

```
S-1     #include <stdlib.h>
S-2     #include <omp.h>
S-3     extern void do_work(int, float *);
S-4     extern void do_more_work(int, float *);
S-5     #pragma omp declare target enter(do_more_work)
S-6
S-7     void async_work(int N, float *d_buf, float *h_buf)
S-8     {
S-9        omp_depend_t obj;
S-10       int d_dev, h_dev;
```

```
   size_t dsize;

   h_dev = omp_get_initial_device();
   d_dev = omp_get_default_device();
   dsize = N * sizeof(float);

   // initialize a depend object 'obj'
   #pragma omp depobj(obj) depend(out: d_buf[0:N])

   // start the async memcpy of s_buf to d_buf on device
   if (omp_target_memcpy_async(d_buf, h_buf, dsize, 0, 0,
                               d_dev, h_dev, 1, &obj))
      abort();

   // do some useful work at the same time on host
   do_work(N, h_buf);

   // wait until memcpy finishes before using d_buf in the target region
   #pragma omp target is_device_ptr(d_buf) depend(depobj: obj)
   do_more_work(N, d_buf);
}
```

C / C++

Fortran

Example device.6.f90 (`omp_5.2`)

```
subroutine async_work(N, d_buf, h_buf)
   use omp_lib
   use, intrinsic    :: iso_c_binding

   implicit none
   integer           :: N
   real, pointer     :: d_buf(:), h_buf(:)

   type(c_ptr)       :: c_dp, c_hp
   integer(c_int)    :: d_dev, h_dev
   integer(c_size_t) :: dsize
   integer(omp_depend_kind) :: obj(1)

   external :: do_work
   external :: do_more_work
   !$omp declare target enter(do_more_work)
   integer :: i

   h_dev = omp_get_initial_device()
   d_dev = omp_get_default_device()
   dsize = N * c_sizeof(d_buf(1))
```

```
S-22
S-23        c_dp = c_loc(d_buf)
S-24        c_hp = c_loc(h_buf)
S-25
S-26        ! initialize a depend object 'obj'
S-27        !$omp depobj(obj) depend(out: d_buf(1:N))
S-28
S-29        ! start the async memcpy of h_buf to d_buf on device
S-30        if (omp_target_memcpy_async(c_dp, c_hp, dsize, 0, 0, &
S-31                                    d_dev, h_dev, 1, obj) /= 0) then
S-32           stop
S-33        endif
S-34
S-35        ! do some useful work at the same time on host
S-36        call do_work(N, h_buf)
S-37
S-38        ! wait until memcpy finishes before using d_buf in the target region
S-39        !$omp target has_device_addr(d_buf) depend(depobj: obj)
S-40        call do_more_work(N, d_buf)
S-41        !$omp end target
S-42
S-43     end subroutine
```

Fortran

6.19 Traits for Specifying Devices

Environment variables **OMP_AVAILABLE_DEVICES** and **OMP_DEFAULT_DEVICE** can take traits to specify the available devices and the default device, respectively. In addition, **OMP_DEFAULT_DEVICE** can also take an integer as a device number to specify the default device.

The following examples show how traits are used to specify devices for these environment variables.

Only GPU non-host devices are available to program:

```
export OMP_AVAILABLE_DEVICES="kind(gpu)"
```

Order of available devices would be all vendor *A* GPUs, then the rest of the non-host devices as specified by "***":

```
export OMP_AVAILABLE_DEVICES="kind(gpu)&&vendor(A),*"
```

Available devices would be all non-gpu devices from vendor *A*:

```
export OMP_AVAILABLE_DEVICES="!kind(gpu)&&vendor(A)"
```

Available devices start with 1 vendor *A* GPU device, then 2 vendor *B* GPU devices, and then the rest of the non-host devices:

```
export OMP_AVAILABLE_DEVICES="(kind(gpu)&&vendor(A))[0],
                              (kind(gpu)&&vendor(B))[0:2],*"
```

The device number range is specified by the C/C++ array section syntax *[0:2]* where "*0*" is the first index and "*2*" is the length.

Three available devices are re-ordered with "*uid-gpu3*" corresponding to device 0, "*uid-gpu2*" to device 1 and "*uid-gpu1*" to device 2:

```
export OMP_AVAILABLE_DEVICES="uid(uid-gpu3),uid(uid-gpu2),
                              uid(uid-gpu1)"
```

The default device will be some visible vendor *A* GPU device. If not available, then set to initial device:

```
export OMP_DEFAULT_DEVICE="kind(gpu)&&vendor(A),initial"
```

The default device will be some visible vendor *A* GPU device. If not available, then set to invalid device so that upon first use of default device the program will error out:

```
export OMP_DEFAULT_DEVICE="kind(gpu)&&vendor(A),invalid"
```

7 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 dependences (or with dependences 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** directive 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 (**notinbranch/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.

7.1 simd and declare simd Directives

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

C / C++

Example SIMD.1.c (**omp_4.0**)

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

C / C++

Fortran

Example SIMD.1.f90 (**omp_4.0**)

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

Fortran

When a function can be inlined within a loop the compiler has an opportunity to vectorize the loop. By guaranteeing SIMD behavior of a function's operations, characterizing the arguments of the function and privatizing temporary variables of the loop, the compiler can often create faster, vector code for the loop. In the examples below the **declare simd** directive 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** directives 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** directive 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 (**omp_4.0**)

```
S-1   #include <stdio.h>
S-2
S-3   #pragma omp declare simd uniform(fact)
S-4   double add1(double a, double b, double fact)
S-5   {
S-6      double c;
S-7      c = a + b + fact;
S-8      return c;
S-9   }
S-10
S-11  #pragma omp declare simd uniform(a,b,fact) linear(i:1)
S-12  double add2(double *a, double *b, int i, double fact)
S-13  {
S-14     double c;
S-15     c = a[i] + b[i] + fact;
S-16     return c;
S-17  }
S-18
S-19  #pragma omp declare simd uniform(fact) linear(a,b:1)
S-20  double add3(double *a, double *b, double fact)
S-21  {
S-22     double c;
S-23     c = *a + *b + fact;
S-24     return c;
S-25  }
S-26
S-27  void work( double *a, double *b, int n )
S-28  {
S-29     int i;
S-30     double tmp;
S-31     #pragma omp simd private(tmp)
S-32     for ( i = 0; i < n; i++ ) {
S-33        tmp  = add1( a[i],  b[i], 1.0);
S-34        a[i] = add2( a,     b, i, 1.0) + tmp;
S-35        a[i] = add3(&a[i], &b[i], 1.0);
S-36     }
S-37  }
S-38
S-39  int main(){
S-40     int i;
S-41     const int N=32;
```

```
S-42        double a[N], b[N];
S-43
S-44        for ( i=0; i<N; i++ ) {
S-45           a[i] = i; b[i] = N-i;
S-46        }
S-47
S-48        work(a, b, N );
S-49
S-50        for ( i=0; i<N; i++ ) {
S-51           printf("%d %f\n", i, a[i]);
S-52        }
S-53
S-54        return 0;
S-55     }
```

C / C++

Fortran

Example SIMD.2.f90 (`omp_4.0`)

```
 S-1     program main
 S-2        implicit none
 S-3        integer, parameter :: N=32
 S-4        integer :: i
 S-5        double precision   :: a(N), b(N)
 S-6        do i = 1,N
 S-7           a(i) = i-1
 S-8           b(i) = N-(i-1)
 S-9        end do
S-10        call work(a, b, N )
S-11        do i = 1,N
S-12           print*, i,a(i)
S-13        end do
S-14     end program
S-15
S-16     function add1(a,b,fact) result(c)
S-17        implicit none
S-18     !$omp declare simd(add1) uniform(fact)
S-19        double precision :: a,b,fact, c
S-20        c = a + b + fact
S-21     end function
S-22
S-23     function add2(a,b,i, fact) result(c)
S-24        implicit none
S-25     !$omp declare simd(add2) uniform(a,b,fact) linear(i:1)
S-26        integer          :: i
S-27        double precision :: a(*),b(*),fact, c
S-28        c = a(i) + b(i) + fact
```

```
S-29    end function
S-30
S-31    subroutine work(a, b, n )
S-32       implicit none
S-33       double precision           :: a(n),b(n), tmp
S-34       integer                    :: n, i
S-35       double precision, external :: add1, add2
S-36
S-37       !$omp simd private(tmp)
S-38       do i = 1,n
S-39          tmp  = add1(a(i), b(i), 1.0d0)
S-40          a(i) = add2(a,    b, i, 1.0d0) + tmp
S-41          a(i) = a(i) + b(i) + 1.0d0
S-42       end do
S-43    end subroutine
```

Fortran

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

C / C++

Example SIMD.3.c (**`omp_4.0`**)

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

C / C++

Fortran

Example SIMD.3.f90 (**omp_4.0**)

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

Fortran

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

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

C / C++

Example SIMD.4.c (**omp_4.0**)

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

C / C++

Fortran

Example SIMD.4.f90 (`omp_4.0`)

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

Fortran

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.

C / C++

Example SIMD.5.c (`omp_4.0`)

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

C / C++

Fortran

Example SIMD.5.f90 (**omp_4.0**)

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

Fortran

7.2 `inbranch` and `notinbranch` Clauses

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

C / C++

Example SIMD.6.c (**omp_4.0**)

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

```
S-15
S-16    #pragma omp declare simd linear(p:1) inbranch
S-17    float goo(float *p){
S-18      *p = *p + 18.5f;
S-19      return *p;
S-20    }
S-21
S-22    int myaddfloat(float *x, float *y, int n)
S-23    {
S-24    #pragma omp simd
S-25      for (int i=0; i<n; i++){
S-26         x[i] = (x[i] > y[i]) ? goo(&y[i]) : y[i];
S-27           /* goo is called under the condition (or within a branch) */
S-28      }
S-29      return x[n-1];
S-30    }
```

C / C++

Fortran

1 *Example SIMD.6.f90* (`omp_4.0`)

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

```
S-27      p = p + 18.5
S-28      r = p
S-29    end function goo
S-30
S-31    function myaddfloat(x, y, n) result(r)
S-32      implicit none
S-33      real :: x(*), y(*), r
S-34      integer :: n
S-35      integer :: i
S-36      real, external :: goo
S-37
S-38      !$omp simd
S-39      do i=1, n
S-40         if (x(i) > y(i)) then
S-41            x(i) = goo(y(i))
S-42            ! goo is called under the condition (or within a branch)
S-43         else
S-44            x(i) = y(i)
S-45         endif
S-46      end do
S-47
S-48      r = x(n)
S-49    end function myaddfloat
```

Fortran

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

C / C++

Example SIMD.7.c (**omp_4.0**)

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

```
S-15    }
S-16
S-17    int main(void)
S-18    {
S-19       int i;
S-20
S-21       #pragma omp simd
S-22       for (i=0; i < N; i++) b[i] = i;
S-23
S-24       #pragma omp simd
S-25       for (i=0; i < N; i++) {
S-26          a[i] = fib(b[i]);
S-27       }
S-28       printf("Done a[%d] = %d\n", N-1, a[N-1]); //Done a[44] = 701408733
S-29       return 0;
S-30    }
```

C / C++

Fortran

Example SIMD.7.f90 (`omp_4.0`)

```
S-1     program fibonacci
S-2        implicit none
S-3        integer,parameter :: N=45
S-4        integer           :: a(0:N-1), b(0:N-1)
S-5        integer           :: i
S-6        integer, external :: fib
S-7
S-8        !$omp simd
S-9        do i = 0,N-1
S-10          b(i) = i
S-11       end do
S-12
S-13       !$omp simd
S-14       do i=0,N-1
S-15          a(i) = fib(b(i))
S-16       end do
S-17
S-18       write(*,*) "Done a(", N-1, ") = ", a(N-1)
S-19                             ! 44  701408733
S-20    end program
S-21
S-22    recursive function fib(n) result(r)
S-23       implicit none
S-24    !$omp declare simd(fib) inbranch
S-25       integer  :: n, r
S-26
```

```
S-27        if (n <= 1) then
S-28           r = n
S-29        else
S-30           r = fib(n-1) + fib(n-2)
S-31        endif
S-32
S-33     end function fib
```

Fortran

7.3 Loop-Carried Lexical Forward Dependence

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.

C / C++

Example SIMD.8.c (**omp_4.0**)

```
S-1      #include <stdio.h>
S-2      #include <math.h>
S-3
S-4      int   P[1000];
S-5      float A[1000];
S-6
S-7      float do_work(float *arr)
S-8      {
S-9        float pri;
S-10       int i;
S-11     #pragma omp simd lastprivate(pri)
S-12       for (i = 0; i < 999; ++i) {
S-13         int j = P[i];
S-14
S-15         pri = 0.5f;
S-16         if (j % 2 == 0) {
S-17           pri = A[j+1] + arr[i];
S-18         }
S-19         A[j] = pri * 1.5f;
```

```
S-20        pri = pri + A[j];
S-21      }
S-22      return pri;
S-23    }
S-24
S-25    int main(void)
S-26    {
S-27      float pri, arr[1000];
S-28      int i;
S-29
S-30      for (i = 0; i < 1000; ++i) {
S-31         P[i]   = i;
S-32         A[i]   = i * 1.5f;
S-33         arr[i] = i * 1.8f;
S-34      }
S-35      pri = do_work(&arr[0]);
S-36      if (pri == 8237.25) {
S-37        printf("passed: result pri = %7.2f (8237.25) \n", pri);
S-38      }
S-39      else {
S-40        printf("failed: result pri = %7.2f (8237.25) \n", pri);
S-41      }
S-42      return 0;
S-43    }
```

C / C++

Fortran

1 *Example SIMD.8.f90* (`omp_4.0`)

```
S-1     module work
S-2
S-3     integer :: P(1000)
S-4     real    :: A(1000)
S-5
S-6     contains
S-7     function do_work(arr) result(pri)
S-8       implicit none
S-9       real, dimension(*) :: arr
S-10
S-11      real :: pri
S-12      integer :: i, j
S-13
S-14      !$omp simd private(j) lastprivate(pri)
S-15      do i = 1, 999
S-16        j = P(i)
S-17
S-18        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
    print 2, "passed", pri
  else
    print 2, "failed", pri
  endif
2 format(a, ": result pri = ", f7.2, " (8237.25)")

end program
```

Fortran

7.4 `ref`, `val`, `uval` Modifiers for `linear` Clause

When generating vector functions from **`declare simd`** directives, it is important for a compiler to know the proper types of function arguments in order to generate efficient codes. This is especially true for C++ reference types and Fortran arguments.

In the following example, the function *add_one2* has a C++ reference parameter (or Fortran argument) *p*. Variable *p* gets incremented by 1 in the function. The caller loop *i* in the main program passes a variable *k* as a reference to the function *add_one2* call. The **`ref`** modifier for the **`linear`** clause on the **`declare simd`** directive specifies that the reference-type parameter *p*

is to match the property of the variable *k* in the loop. This use of reference type is equivalent to the second call to *add_one2* with a direct passing of the array element *a[i]*. In the example, the preferred vector length 8 is specified for both the caller loop and the callee function.

When **linear(*p*: ref)** is applied to an argument passed by reference, it tells the compiler that the addresses in its vector argument are consecutive, and so the compiler can generate a single vector load or store instead of a gather or scatter. This allows more efficient SIMD code to be generated with less source changes.

C++

Example linear_modifier.1.cpp (**omp_5.2**)

```
S-1   #include <stdio.h>
S-2
S-3   #define NN 1023
S-4   int a[NN];
S-5
S-6   #pragma omp declare simd linear(p: ref) simdlen(8)
S-7   void add_one2(int& p)
S-8   {
S-9      p += 1;
S-10  }
S-11
S-12  int main(void)
S-13  {
S-14     int i;
S-15     int* p = a;
S-16
S-17     for (i = 0; i < NN; i++) {
S-18        a[i] = i;
S-19     }
S-20
S-21  #pragma omp simd linear(p) simdlen(8)
S-22     for (i = 0; i < NN; i++) {
S-23        int& k = *p;
S-24        add_one2(k);
S-25        add_one2(a[i]);
S-26        p++;
S-27     }
S-28
S-29     for (i = 0; i < NN; i++) {
S-30        if (a[i] != i+2) {
S-31           printf("failed\n");
S-32           return 1;
S-33        }
S-34     }
S-35     printf("passed\n");
S-36     return 0;
```

```
}
```

C++

Fortran

Example linear_modifier.1.f90 (`omp_5.2`)

```
module m
   integer, parameter :: NN = 1023
   integer :: a(NN)

 contains
   subroutine add_one2(p)
   implicit none
   !$omp declare simd(add_one2) linear(p: ref) simdlen(8)

   integer :: p

   p = p + 1
   end subroutine
end module

program main
   use m
   implicit none
   integer :: i, p

   do i = 1, NN
      a(i) = i
   end do

   p = 1
   !$omp simd linear(p) simdlen(8)
   do i = 1, NN
      associate(k => a(p))
         call add_one2(k)
      end associate
      call add_one2(a(i))
      p = p + 1
   end do

   do i = 1, NN
      if (a(i) /= i+2) then
         print *, "failed"
         stop
      endif
   end do
```

```
S-41      print *, "passed"
S-42   end program
```

Fortran

The following example is a variant of the above example. The function *add_one2* in the C++ code includes an additional C++ reference parameter *i*. The loop index *i* of the caller loop *i* in the main program is passed as a reference to the function *add_one2* call. The loop index *i* has a uniform address with linear value of step 1 across SIMD lanes. Thus, the **uval** modifier is used for the **linear** clause to specify that the C++ reference-type parameter *i* is to match the property of loop index *i*.

In the corresponding Fortran code the arguments *p* and *i* in the routine *add_on2* are passed by references. Similar modifiers are used for these variables in the **linear** clauses to match with the property at the caller loop in the main program.

When **linear(***i*: **uval)** is applied to an argument passed by reference, it tells the compiler that its addresses in the vector argument are uniform so that the compiler can generate a scalar load or scalar store and create linear values. This allows more efficient SIMD code to be generated with less source changes.

C++

Example linear_modifier.2.cpp (**omp_5.2**)

```
S-1    #include <stdio.h>
S-2
S-3    #define NN 1023
S-4    int a[NN];
S-5
S-6    #pragma omp declare simd linear(p: ref) linear(i: uval)
S-7    void add_one2(int& p, const int& i)
S-8    {
S-9       p += i;
S-10   }
S-11
S-12   int main(void)
S-13   {
S-14      int i;
S-15      int* p = a;
S-16
S-17      for (i = 0; i < NN; i++) {
S-18         a[i] = i;
S-19      }
S-20
S-21      #pragma omp simd linear(p)
S-22      for (i = 0; i < NN; i++) {
S-23         int& k = *p;
S-24         add_one2(k, i);
```

```
S-25        p++;
S-26     }
S-27
S-28     for (i = 0; i < NN; i++) {
S-29        if (a[i] != i*2) {
S-30           printf("failed\n");
S-31           return 1;
S-32        }
S-33     }
S-34     printf("passed\n");
S-35     return 0;
S-36  }
```

C++

Fortran

1 *Example linear_modifier.2.f90* (`omp_5.2`)

```
S-1   module m
S-2      integer, parameter :: NN = 1023
S-3      integer :: a(NN)
S-4
S-5    contains
S-6      subroutine add_one2(p, i)
S-7      implicit none
S-8      !$omp declare simd(add_one2) linear(p: ref) linear(i: uval)
S-9
S-10     integer :: p
S-11     integer, intent(in) :: i
S-12
S-13     p = p + i
S-14     end subroutine
S-15  end module
S-16
S-17  program main
S-18     use m
S-19     implicit none
S-20     integer :: i, p
S-21
S-22     do i = 1, NN
S-23        a(i) = i
S-24     end do
S-25
S-26     p = 1
S-27     !$omp simd linear(p)
S-28     do i = 1, NN
S-29        call add_one2(a(p), i)
```

```
S-30          p = p + 1
S-31       end do
S-32
S-33       do i = 1, NN
S-34          if (a(i) /= i*2) then
S-35             print *, "failed"
S-36             stop
S-37          endif
S-38       end do
S-39       print *, "passed"
S-40    end program
```

Fortran

In the following example, the function *func* takes arrays *x* and *y* as arguments, and accesses the array elements referenced by the index *i*. The caller loop *i* in the main program passes a linear copy of the variable *k* to the function *func*. The **val** modifier is used for the **linear** clause in the **declare simd** directive for the function *func* to specify that the argument *i* is to match the property of the actual argument *k* passed in the SIMD loop. Arrays *x* and *y* have uniform addresses across SIMD lanes.

When **linear(***i*: **val,step(***1***))** is applied to an argument, it tells the compiler that its addresses in the vector argument may not be consecutive, however, their values are linear (with stride 1 here). When the value of *i* is used in subscript of array references (e.g., *x[i]*), the compiler can generate a vector load or store instead of a gather or scatter. This allows more efficient SIMD code to be generated with less source changes.

C / C++

Example linear_modifier.3.c (**omp_5.2**)

```
S-1     #include <stdio.h>
S-2
S-3     #define N 128
S-4
S-5     #pragma omp declare simd simdlen(4) uniform(x, y) linear(i:val,step(1))
S-6     double func(double x[], double y[], int i)
S-7     {
S-8        return (x[i] + y[i]);
S-9     }
S-10
S-11    int main(void)
S-12    {
S-13       double x[N], y[N], z1[N], z2;
S-14       int i, k;
S-15
S-16       for (i = 0; i < N; i++) {
S-17          x[i] = (double)i;
S-18          y[i] = (double)i*2;
```

```
   }

   k = 0;
#pragma omp simd linear(k)
   for (i = 0; i < N; i++) {
      z1[i] = func(x, y, k);
      k++;
   }

   for (i = 0; i < N; i++) {
      z2 = (double)(i + i*2);
      if (z1[i] != z2) {
         printf("failed\n");
         return 1;
      }
   }
   printf("passed\n");
   return 0;
}
```

C / C++

Fortran

Example linear_modifier.3.f90 (`omp_5.2`)

```
module func_mod
contains
   real(8) function func(x, y, i)
   implicit none
!$omp declare simd(func) simdlen(4) uniform(x, y) linear(i:val,step(1))

      real(8), intent(in) :: x(*), y(*)
      integer, intent(in) :: i

      func = x(i) + y(i)

   end function func
end module func_mod

program main
   use func_mod
   implicit none
   integer, parameter :: n = 128
   real(8) :: x(n), y(n), z1(n), z2
   integer :: i, k

   do i=1, n
      x(i) = real(i, kind=8)
```

```
S-24          y(i) = real(i*2, kind=8)
S-25       enddo
S-26
S-27       k = 1
S-28    !$omp simd linear(k)
S-29       do i=1, n
S-30          z1(i) = func(x, y, k)
S-31          k = k + 1
S-32       enddo
S-33
S-34       do i=1, n
S-35          z2 = real(i+i*2, kind=8)
S-36          if (z1(i) /= z2) then
S-37             print *, 'failed'
S-38             stop
S-39          endif
S-40       enddo
S-41       print *, 'passed'
S-42    end program main
```

Fortran

This page intentionally left blank

8 Loop Transformations

To obtain better performance on a platform, code may need to be restructured relative to the way it is written (which is often for best readability). User-directed loop transformations accomplish this goal by providing a means to separate code semantics and its optimization.

A loop transformation construct states that a transformation operation is to be performed on set of nested loops. This directive approach can target specific loops for transformation, rather than applying more time-consuming general compiler heuristics methods with compiler options that may not be able to discover optimal transformations.

Loop transformations can be augmented by preprocessor support or OpenMP **metadirective** directives, to select optimal dimension and size parameters for specific platforms, facilitating a single code base for multiple platforms. Moreover, directive-based transformations make experimenting easier: whereby specific hot spots can be affected by transformation directives.

8.1 `tile` Construct

In the following example a **tile** construct transforms two nested loops within the *func1* function into four nested loops. The tile sizes in the **sizes** clause are applied from outermost to innermost loops (left-to-right). The effective tiling operation is illustrated in the *func2* function. (For easier illustration, tile sizes for all examples in this section evenly divide the iteration counts so that there are no remainders.)

In the following C/C++ code the inner loop traverses columns and the outer loop traverses the rows of a 100x128 (row x column) matrix. The **sizes(***5,16***)** clause of the **tile** construct specifies a 5x16 blocking, applied to the outer (row) and inner (column) loops. The worksharing-loop construct before the **tile** construct is applied after the transform.

C / C++

Example tile.1.c (**omp_5.1**)

```
S-1  void func1(int A[100][128])
S-2  {
S-3     #pragma omp parallel for
S-4     #pragma omp tile sizes(5,16)
S-5     for (int i = 0; i < 100; ++i)
S-6         for (int j = 0; j < 128; ++j)
S-7             A[i][j] = i*1000 + j;
S-8  }
S-9
```

```
S-10  void func2(int A[100][128])
S-11  {
S-12     #pragma omp parallel for
S-13     for (int i1 = 0; i1 < 100; i1+=5)
S-14         for (int j1 = 0; j1 < 128; j1+=16)
S-15             for (int i2 = i1; i2 < i1+5; ++i2)
S-16                 for (int j2 = j1; j2 < j1+16; ++j2)
S-17                    A[i2][j2] = i2*1000 + j2;
S-18  }
```

C / C++

In the following Fortran code the inner loop traverses rows and the outer loop traverses the columns of a 128x100 (row x column) matrix. The **sizes(***5, 16***)** clause of the **tile** construct specifies a 5x16 blocking, applied to the outer (column) and inner (row) loops. The worksharing-loop construct before the **tile** construct is applied after the transform.

Fortran

Example tile.1.f90 (**omp_5.1**)

```
S-1   subroutine func1(A)
S-2       integer :: A(128,100)
S-3       integer :: i, j
S-4       !$omp parallel do
S-5       !$omp tile sizes(5,16)
S-6       do i = 1, 100
S-7       do j = 1, 128
S-8           A(j,i) = j*1000 + i
S-9       end do; end do
S-10  end subroutine
S-11
S-12  subroutine func2(A)
S-13      integer :: A(128,100)
S-14      integer :: i1, j1, i2, j2
S-15      !$omp parallel do
S-16      do i1 = 1, 100,5
S-17      do j1 = 1, 128,16
S-18         do i2 = i1, i1+( 5-1)
S-19         do j2 = j1, j1+(16-1)
S-20            A(j2,i2) = j2*1000 + i2
S-21         end do; end do
S-22      end do; end do
S-23  end subroutine
```

Fortran

This example illustrates transformation nesting. Here, a 4x4 "outer" **tile** construct is applied to the "inner" tile transform shown in the example above. The effect of the inner loop is shown in *func2* (cf. *func2* in *tile.1.c*). The outer **tile** construct's **sizes(***4, 4***)** clause applies a 4x4 tile upon the resulting blocks of the inner transform. The effective looping is shown in *func3*.

C / C++

Example tile.2.c (**omp_5.1**)

```
S-1   void func1(int A[100][128])
S-2   {
S-3      #pragma omp tile sizes(4, 4)
S-4      #pragma omp tile sizes(5,16)
S-5      for (int i = 0; i < 100; ++i)
S-6          for (int j = 0; j < 128; ++j)
S-7              A[i][j] = i*1000 + j;
S-8   }
S-9
S-10  void func2(int A[100][128])
S-11  {
S-12     #pragma omp tile sizes(4,4)
S-13     for (int i1 = 0; i1 < 100; i1+=5)
S-14         for (int j1 = 0; j1 < 128; j1+=16)
S-15             for (int i2 = i1; i2 < i1+5; ++i2)
S-16                 for (int j2 = j1; j2 < j1+16; ++j2)
S-17                    A[i2][j2] = i2*1000 + j2;
S-18  }
S-19
S-20  void func3(int A[100][128])
S-21  {
S-22     for (int i11 = 0; i11 < 100; i11+= 5*4)
S-23     for (int j11 = 0; j11 < 128; j11+=16*4)
S-24
S-25         for (int i12 = i11; i12 < i11+( 5*4); i12+= 5)
S-26         for (int j12 = j11; j12 < j11+(16*4); j12+=16)
S-27
S-28             for (int i2 = i12; i2 < i12+ 5; ++i2)
S-29             for (int j2 = j12; j2 < j12+16; ++j2)
S-30                A[i2][j2] = i2*1000 + j2;
S-31  }
```

C / C++

Fortran

Example tile.2.f90 (`omp_5.1`)

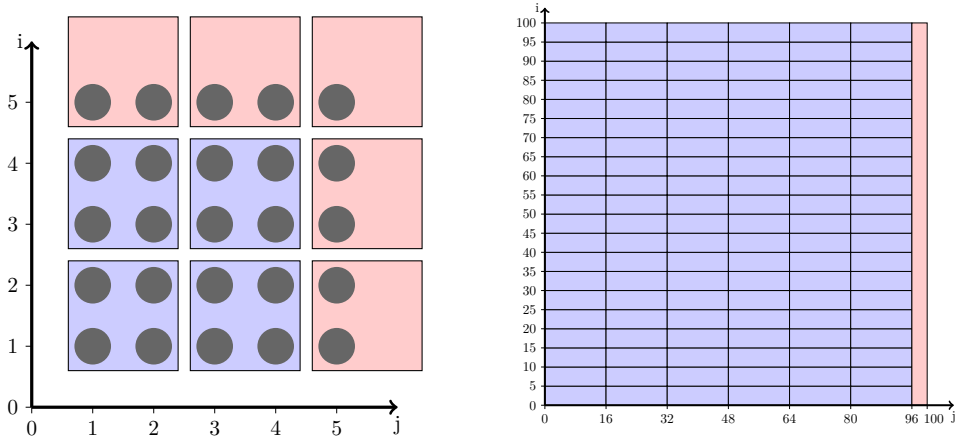
```
S-1   subroutine func1(A)
S-2       integer :: A(128,100)
S-3       integer :: i, j
S-4       !$omp tile sizes(4, 4)
S-5       !$omp tile sizes(5,16)
S-6       do i = 1, 100
S-7       do j = 1, 128
S-8          A(j,i) = j*1000 + i
S-9       end do; end do
S-10  end subroutine
S-11
S-12  subroutine func2(A)
S-13      integer :: A(128,100)
S-14      integer :: i1, j1, i2, j2
S-15      !$omp tile sizes(4,4)
S-16      do i1 = 1, 100,5
S-17      do j1 = 1, 128,16
S-18         do i2 = i1, i1+( 5-1)
S-19         do j2 = j1, j1+(16-1)
S-20            A(j2,i2) = j2*1000 + i2
S-21         end do; end do
S-22      end do; end do
S-23
S-24  end subroutine
S-25
S-26  subroutine func3(A)
S-27      integer :: A(128,100)
S-28      integer :: i11, j11, i12, j12, i2, j2
S-29      do i11 = 1, 100,  5*4
S-30      do j11 = 1, 128, 16*4
S-31         do i12 = i11, i11+( 5*4-1),  5
S-32         do j12 = j11, j11+(16*4-1), 16
S-33            do i2 = i12, i12+ 5-1
S-34            do j2 = j12, j12+16-1
S-35               A(j2,i2) = j2*1000 + i2
S-36            enddo; enddo;
S-37         enddo; enddo;
S-38      enddo; enddo
S-39
S-40  end subroutine
```

Fortran

8.2 Incomplete Tiles

Optimal performance for tiled loops is achieved when the loop iteration count is a multiple of the tile size. When this condition does not exist, the implementation is free to execute the partial loops in a manner that optimizes performance, while preserving the specified order of iterations in the complete-tile loops.

Figure 8.1a shows an example of a 2-by-2 tiling for a 5-by-5 iteration space. There are nine resulting tiles. Four are *complete* 2-by-2 tiles, and the remaining five tiles are *partial* tiles.



(A) 2-dimensional tiling with partial tiles

(B) Partial tiles of Example *partial_tile.1*

FIGURE 8.1: Tiling illustrations

In the following example, function `func1` uses the **`tile`** construct with a **`sizes(`***`4,16`***`)`** tiling clause. Because the second tile dimension of 16 does not evenly divide into the iteration count of the *j*-loop, the iterations corresponding to the remainder for the *j*-loop correspond to partial tiles as shown in Figure 8.1b. Each remaining function illustrates a code implementation that a compiler may generate to implement the **`tile`** construct in `func1`.

The order of tile execution relative to other tiles can be changed, but execution order of iterations within the same tile must be preserved. Implementations must ensure that dependencies that are valid with any tile size need to be preserved (including tile size of 1 and tiles as large as the iteration space).

Functions `func2` through `func6` are valid implementations of `func1`. In `func2` the unrolling is illustrated as a pair of nested loops with a simple adjustment in the size of the final iteration block in the `j2` iteration space for the partial tile.

Performance of the implementation depends on the hardware architecture, the instruction set and compiler optimization goals. Functions `func3`, `func4`, and `func5` have the advantage that the

innermost loop for the complete tile is a constant size and can be replaced with SIMD instructions. If the target platform has masked SIMD instructions with no overhead, then avoiding the construction of a remainder loop, as in *func5*, might be the best option. Another option is to use a remainder loop without tiling, as shown in *func6*, to reduce control-flow overhead.

C / C++

Example partial_tile.1.c (**omp_5.1**)

```
S-1   int min(int a, int b){ return (a < b)? a : b; }
S-2
S-3   void func1(double A[100][100])
S-4   {
S-5      #pragma omp tile sizes(4,16)
S-6      for (int i = 0; i < 100; ++i)
S-7          for (int j = 0; j < 100; ++j)
S-8             A[i][j] = A[i][j] + 1;
S-9   }
S-10
S-11  void func2(double A[100][100])
S-12  {
S-13     for (int i1 = 0; i1 < 100; i1+=4)
S-14         for (int j1 = 0; j1 < 100; j1+=16)
S-15             for (int i2 = i1; i2 < i1+4; ++i2)
S-16                 for (int j2 = j1; j2 < min(j1+16,100); ++j2)
S-17                    A[i2][j2] = A[i2][j2] + 1;
S-18  }
S-19
S-20  void func3(double A[100][100])
S-21  {
S-22     // complete tiles
S-23     for (int i1 = 0; i1 < 100; i1+=4)
S-24         for (int j1 = 0; j1 < 96; j1+=16)
S-25             for (int i2 = i1; i2 < i1+4; ++i2)
S-26                 for (int j2 = j1; j2 < j1+16; ++j2)
S-27                    A[i2][j2] = A[i2][j2] + 1;
S-28     // partial tiles / remainder
S-29     for (int i1 = 0; i1 < 100; i1+=4)
S-30         for (int i2 = i1; i2 < i1+4; ++i2)
S-31             for (int j = 96; j < 100; j+=1)
S-32                    A[i2][j] = A[i2][j] + 1;
S-33  }
S-34
S-35  void func4(double A[100][100])
S-36  {
S-37     for (int i1 = 0; i1 < 100; i1+=4) {
S-38         // complete tiles
S-39         for (int j1 = 0; j1 < 96; j1+=16)
```

```
S-40                for (int i2 = i1; i2 < i1+4; ++i2)
S-41                    for (int j2 = j1; j2 < j1+16; ++j2)
S-42                       A[i2][j2] = A[i2][j2] + 1;
S-43            // partial tiles
S-44            for (int i2 = i1; i2 < i1+4; ++i2)
S-45                for (int j = 96; j < 100; j+=1)
S-46                       A[i2][j] = A[i2][j] + 1;
S-47        }
S-48    }
S-49
S-50    void func5(double A[100][100])
S-51    {
S-52        for (int i1 = 0; i1 < 100; i1+=4)
S-53            for (int j1 = 0; j1 < 100; j1+=16)
S-54                for (int i2 = i1; i2 < i1+4; ++i2)
S-55                    for (int j2 = j1; j2 <j1+16; ++j2)
S-56                       if (j2 < 100)
S-57                           A[i2][j2] = A[i2][j2] + 1;
S-58    }
S-59
S-60    void func6(double A[100][100])
S-61    {
S-62        // complete tiles
S-63        for (int i1 = 0; i1 < 100; i1+=4)
S-64            for (int j1 = 0; j1 < 96; j1+=16)
S-65                for (int i2 = i1; i2 < i1+4; ++i2)
S-66                    for (int j2 = j1; j2 < j1+16; ++j2)
S-67                       A[i2][j2] = A[i2][j2] + 1;
S-68        // partial tiles / remainder (not tiled)
S-69        for (int i = 0; i < 100; ++i)
S-70            for (int j = 96; j < 100; ++j)
S-71                       A[i][j] = A[i][j] + 1;
S-72    }
```

C / C++

Fortran

1 *Example partial_tile.1.f90* (`omp_5.1`)

```
S-1     subroutine func1(A)
S-2        implicit none
S-3        double precision   :: A(100,100)
S-4        integer            :: i,j
S-5
S-6        !$omp tile sizes(4,16)
S-7        do i = 1, 100
S-8        do j = 1, 100
S-9           A(j,i)  = A(j,i) + 1
```

```
S-10        end do; end do
S-11
S-12     end subroutine
S-13
S-14
S-15     subroutine func2(A)
S-16        implicit none
S-17        double precision   :: A(100,100)
S-18        integer            :: i1,i2,j1,j2
S-19
S-20        do i1 =  1, 100,  4
S-21        do j1 =  1, 100, 16
S-22        do i2 = i1,  i1 + 3
S-23        do j2 = j1, min(j1+15,100)
S-24           A(j2,i2) = A(j2,i2) + 1
S-25        end do; end do; end do; end do
S-26
S-27     end subroutine
S-28
S-29
S-30     subroutine func3(A)
S-31        implicit none
S-32        double precision   :: A(100,100)
S-33        integer            :: i1,i2,j1,j2, j
S-34
S-35        !! complete tiles
S-36        do i1 =  1, 100,  4
S-37        do j1 =  1,  96, 16
S-38        do i2 = i1,  i1 + 3
S-39        do j2 = j1,  j1 +15
S-40           A(j2,i2) = A(j2,i2) + 1
S-41        end do; end do; end do; end do
S-42
S-43        !! partial tiles / remainder
S-44        do i1 =  1, 100, 4
S-45        do i2 = i1,  i1 +3
S-46        do  j = 97, 100
S-47           A(j,i2) = A(j,i2) + 1
S-48        end do; end do; end do
S-49
S-50     end subroutine
S-51
S-52
S-53     subroutine func4(A)
S-54        implicit none
S-55        double precision   :: A(100,100)
S-56        integer            :: i1,i2,j1,j2, j
```

```
S-57
S-58        do i1 =  1, 100,  4
S-59
S-60           !! complete tiles
S-61           do j1 =  1,  96, 16
S-62           do i2 = i1,  i1 + 3
S-63           do j2 = j1,  j1 +15
S-64              A(j2,i2) = A(j2,i2) + 1
S-65           end do; end do; end do
S-66
S-67           !! partial tiles
S-68           do i2 = i1,  i1 +3
S-69           do  j = 97, 100
S-70              A(j,i2) = A(j,i2) + 1
S-71           end do; end do
S-72
S-73        end do
S-74
S-75     end subroutine
S-76
S-77
S-78     subroutine func5(A)
S-79        implicit none
S-80        double precision   :: A(100,100)
S-81        integer            :: i1,i2,j1,j2
S-82
S-83        do i1 =  1, 100,  4
S-84        do j1 =  1, 100, 16
S-85        do i2 = i1,  i1 + 3
S-86        do j2 = j1,  j1 +15
S-87           if (j2 < 101) A(j2,i2) = A(j2,i2) + 1
S-88        end do; end do; end do; end do
S-89
S-90     end subroutine
S-91
S-92
S-93     subroutine func6(A)
S-94        implicit none
S-95        double precision   :: A(100,100)
S-96        integer            :: i1,i2,j1,j2, i,j
S-97
S-98        !! complete tiles
S-99        do i1 =  1, 100,  4
S-100       do j1 =  1,  96, 16
S-101       do i2 = i1,  i1 + 3
S-102       do j2 = j1,  j1 +15
S-103          A(j2,i2) = A(j2,i2) + 1
```

```
S-104       end do; end do; end do; end do
S-105
S-106       !! partial tiles / remainder (not tiled)
S-107       do i =  1, 100
S-108       do j = 97, 100
S-109          A(j,i) = A(j,i) + 1
S-110       end do; end do
S-111
S-112    end subroutine
```

Fortran

In the following example, function *func7* tiles nested loops with a size of (*4,16*), resulting in partial tiles that cover the last 4 iterations of the *j*-loop, as in the previous example. However, the outer loop is parallelized with a **parallel** worksharing-loop construct.

Functions *func8* and *func9* illustrate two implementations of the tiling with **parallel** and worksharing-loop directives. Function *func8* uses a single outer loop, with a *min* function to accommodate the partial tiles. Function *func9* uses two sets of nested loops, the first iterates over the complete tiles and the second covers iterations from the partial tiles. When fissioning loops that are in a **parallel** worksharing-loop region, each iteration of each workshared loop must be executed on the same thread as in an un-fissioned loop. The **schedule(static)** clause in *func7* forces the implementation to use static scheduling and allows the fission in function *func8*. When dynamic scheduling is prescribed, fissioning is not allowed. When no scheduling is specified, the compiler implementation will select a scheduling *kind* and adhere to its restrictions.

C / C++

Example partial_tile.2.c (**omp_5.1**)

```
S-1     int min(int a, int b){ return (a < b)? a : b; }
S-2
S-3     void func7(double A[100][100])
S-4     {
S-5        #pragma omp parallel for schedule(static)
S-6        #pragma omp tile sizes(4,16)
S-7        for (int i = 0; i < 100; ++i)
S-8            for (int j = 0; j < 100; ++j)
S-9                A[i][j] = A[i][j] + 1;
S-10    }
S-11
S-12    void func8(double A[100][100])
S-13    {
S-14       #pragma omp parallel for schedule(static)
S-15       for (int i1 = 0; i1 < 100; i1+=4)
S-16           for (int j1 = 0; j1 < 100; j1+=16)
S-17               for (int i2 = i1; i2 < i1+4; ++i2)
S-18                   for (int j2 = j1; j2 < min(j1+16,100); ++j2)
S-19                      A[i2][j2] = A[i2][j2] + 1;
```

```
S-20    }
S-21
S-22    void func9(double A[100][100])
S-23    {
S-24       #pragma omp parallel
S-25       {
S-26           #pragma omp for schedule(static) nowait
S-27           for (int i1 = 0; i1 < 100; i1+=4)
S-28               for (int j1 = 0; j1 < 96; j1+=16)
S-29                   for (int i2 = i1; i2 < i1+4; ++i2)
S-30                       for (int j2 = j1; j2 < j1+16; ++j2)
S-31                          A[i2][j2] = A[i2][j2] + 1;
S-32           #pragma omp for schedule(static)
S-33           for (int i1 = 0; i1 < 100; i1+=4)
S-34               for (int i2 = i1; i2 < i1+4; ++i2)
S-35                   for (int j = 96; j < 100; j+=1)
S-36                       A[i2][j] = A[i2][j] + 1;
S-37        }
S-38    }
```

C / C++

Fortran

1 *Example partial_tile.2.f90* (`omp_5.1`)

```
S-1     subroutine func7(A)
S-2        implicit none
S-3        double precision   :: A(100,100)
S-4        integer            :: i,j
S-5
S-6        !$omp parallel do schedule(static)
S-7        !$omp tile sizes(4,16)
S-8        do i=1,100
S-9        do j = 1, 100
S-10          A(j,i)  = A(j,i) + 1
S-11       end do; end do
S-12
S-13    end subroutine
S-14
S-15    subroutine func8(A)
S-16       implicit none
S-17       double precision   :: A(100,100)
S-18       integer            :: i1,i2,j1,j2
S-19
S-20       do i1 =  1, 100,  4
S-21       do j1 =  1, 100, 16
S-22       do i2 = i1,  i1 + 3
S-23       do j2 = j1, min(j1+15,100)
```

```
S-24          A(j2,i2) = A(j2,i2) + 1
S-25       end do; end do; end do; end do
S-26
S-27    end subroutine
S-28
S-29    subroutine func9(A)
S-30       implicit none
S-31       double precision   :: A(100,100)
S-32       integer            :: i1,i2,j1,j2,j
S-33
S-34       !$omp parallel
S-35
S-36          !$omp do schedule(static)
S-37          do i1 =  1, 100,  4
S-38          do j1 =  1,  96, 16
S-39          do i2 = i1,  i1 + 3
S-40          do j2 = j1,  j1 +15
S-41             A(j2,i2) = A(j2,i2) + 1
S-42          end do; end do; end do; end do
S-43          !$omp end do nowait
S-44
S-45          !$omp do schedule(static)
S-46          do i1 =  1, 100, 4
S-47          do i2 = i1,  i1 +3
S-48          do  j = 97, 100
S-49             A(j,i2) = A(j,i2) + 1
S-50          end do; end do; end do;
S-51
S-52       !$omp end parallel
S-53
S-54    end subroutine
```

Fortran

8.3 `unroll` Construct

The **`unroll`** construct is a loop transformation that increases the number of loop blocks in a loop, while reducing the number of iterations. The **`full`** clause specifies that the loop is to be completely unrolled. That is, a loop block for each iteration is created, and the loop is removed. A **`partial`** clause with an *unroll-factor* specifies that the number of iterations will be reduced multiplicatively by the factor while the number of blocks will be increased by the same factor. Operationally, the loop is tiled by the factor, and the tiled loop is fully expanded, resulting in a single loop with multiple blocks.

Unrolling can reduce control-flow overhead and provide additional optimization opportunities for the compiler and the processor pipeline. Nevertheless, unrolling can increase the code size, and saturate the instruction cache. Hence, the trade-off may need to be assessed. Unrolling a loop does not change the code's semantics. Also, compilers may unroll loops without explicit directives, at various optimization levels.

In the example below, the **`unroll`** construct is used without any clause, and then with a **`full`** clause, in the first two functions, respectively. When no clause is used, it is up to the implementation (compiler) to decide if and how the loop is to be unrolled. The iteration count can have a run time value. In the second function, the **`unroll`** construct uses a **`full`** clause to completely unroll the loop. A compile-time constant is required for the iteration count. The statements in the third function (*`unroll_full_equivalent`*) illustrates equivalent code for the full unrolling in the second function.

C / C++

Example unroll.1.c (**`omp_5.1`**)

```
S-1   void unroll(double A[], int n)
S-2   {
S-3      #pragma omp unroll
S-4      for (int i = 0; i < n; ++i)
S-5         A[i] = 0;
S-6   }
S-7
S-8   void unroll_full(double A[])
S-9   {
S-10     #pragma omp unroll full
S-11     for (int i = 0; i < 4; ++i)
S-12        A[i] = 0;
S-13  }
S-14
S-15  void unroll_full_equivalent(double A[])
S-16  {
S-17     A[0] = 0;
S-18     A[1] = 0;
S-19     A[2] = 0;
```

```
S-20        A[3] = 0;
S-21    }
```

C / C++

Fortran

Example unroll.1.f90 (`omp_5.1`)

```
 S-1    subroutine unroll(A, n)
 S-2       implicit none
 S-3       integer          :: i,n
 S-4       double precision :: A(n)
 S-5
 S-6       !$omp unroll
 S-7       do i = 1,n
 S-8          A(i) = 0.0d0
 S-9       end do
S-10    end subroutine
S-11
S-12    subroutine unroll_full(A)
S-13       implicit none
S-14       integer :: i
S-15       double precision :: A(*)
S-16
S-17       !$omp unroll full
S-18       do i = 1,4
S-19          A(i) = 0.0d0
S-20       end do
S-21    end subroutine
S-22
S-23    subroutine unroll_full_equivalent(A)
S-24       implicit none
S-25       double precision :: A(*)
S-26
S-27       A(1) = 0.0d0
S-28       A(2) = 0.0d0
S-29       A(3) = 0.0d0
S-30       A(4) = 0.0d0
S-31    end subroutine
```

Fortran

The next example shows cases when it is incorrect to use full unrolling.

C / C++

1 *Example unroll.2.c* (`omp_5.1`)

```
S-1   void illegal_2a(double A[])
S-2   {
S-3       #pragma omp for
S-4       #pragma omp unroll full  // ERROR: No loop left after full unrolling.
S-5       for (int i = 0; i < 12; ++i)
S-6           A[i] = 0;
S-7   }
S-8
S-9   void illegal_2b(double A[])
S-10  {
S-11      // Loop might be fully unrolled (or a partially unrolled loop
S-12      // replacement). Hence, no canonical for-loop, resulting in
S-13      // non-compliant code. Implementations may suggest adding a
S-14      // "partial" clause.
S-15
S-16      #pragma omp for          //        Requires a canonical loop
S-17      #pragma omp unroll       // ERROR: may result in non-compliant code
S-18      for (int i = 0; i < 12; ++i)
S-19          A[i] = 0;
S-20  }
S-21
S-22  void illegal_2c(int n, double A[])
S-23  {
S-24      #pragma omp unroll full  // ERROR: Constant iteration count required.
S-25      for (int i = 0; i < n; ++i)
S-26          A[i] = 0;
S-27  }
```

C / C++
Fortran

2 *Example unroll.2.f90* (`omp_5.1`)

```
S-1   subroutine illegal_2a(A)
S-2      implicit none
S-3      double precision :: A(*)
S-4      integer :: i
S-5
S-6      !$omp do
S-7      !$omp unroll full  !! ERROR: No loop left after full unrolling
S-8      do i = 1,12
S-9         A(i) = 0.0d0
S-10     end do
S-11  end subroutine
S-12
```

```
S-13    subroutine illegal_2b(A)
S-14       implicit none
S-15       double precision :: A(*)
S-16       integer :: i
S-17
S-18       !! Loop might be fully unrolled (or a partially unrolled loop
S-19       !! replacement).  Hence, no canonical do-loop will exist,
S-20       !! resulting in non-compliant code.
S-21       !! Implementations may suggest to adding a "partial" clause.
S-22
S-23       !$omp do           !!        Requires a canonical loop
S-24       !$omp unroll       !! ERROR: may result in non-compliant code
S-25       do i = 1,12
S-26          A(i) = 0.0d0
S-27       end do
S-28    end subroutine
S-29
S-30    subroutine illegal_2c(n, A)
S-31       implicit none
S-32       integer          :: i,n
S-33       double precision :: A(*)
S-34
S-35       !$omp unroll full  !! Full unroll requires constant iteration count
S-36       do i = 1,n
S-37          A(i) = 0.0d0
S-38       end do
S-39    end subroutine
```

Fortran

In many cases, when the iteration count is large and/or dynamic, it is reasonable to partially unroll a loop by including a **partial** clause. In the *unroll3_partial* function below, the *unroll-factor* value of 4 is used to create a tile size of 4 that is unrolled to create 4 unrolled statements. The equivalent "hand unrolled" loop code is presented in the *unroll3_partial_equivalent* function. If the *unroll-factor* is omitted, as in the *unroll3_partial_nofactor* function, the implementation may optimally select a factor from 1 (no unrolling) to the iteration count (full unrolling). In the latter case the construct generates a loop with a single iteration.

C / C++

Example unroll.3.c (**omp_5.1**)

```
S-1     void unroll3_partial(double A[])
S-2     {
S-3        #pragma omp unroll partial(4)
S-4        for (int i = 0; i < 128; ++i)
S-5            A[i] = 0;
S-6     }
S-7
S-8     void unroll3_partial_equivalent(double A[])
S-9     {
S-10       for (int i_iv = 0; i_iv < 32; ++i_iv) {
S-11           A[i_iv * 4 + 0] = 0;
S-12           A[i_iv * 4 + 1] = 0;
S-13           A[i_iv * 4 + 2] = 0;
S-14           A[i_iv * 4 + 3] = 0;
S-15       }
S-16    }
S-17
S-18    void unroll3_partial_nofactor(double A[])
S-19    {
S-20       #pragma omp unroll partial
S-21       for (int i = 0; i < 128; ++i)
S-22           A[i] = 0;
S-23    }
```

C / C++

Fortran

1 *Example unroll.3.f90* (`omp_5.1`)

```
S-1     subroutine unroll3_partial(A)
S-2        implicit none
S-3        double precision :: A(*)
S-4        integer :: i
S-5
S-6        !$omp unroll partial(4)
S-7        do i = 1,128
S-8           A(i) = 0
S-9        end do
S-10    end subroutine
S-11
S-12    subroutine unroll3_partial_equivalent(A)
S-13       implicit none
S-14       double precision :: A(*)
S-15       integer :: i_iv
S-16
S-17       do i_iv = 0, 31
S-18          A(i_iv * 4 + 1) = 0
S-19          A(i_iv * 4 + 2) = 0
```

```
S-20          A(i_iv * 4 + 3) = 0
S-21          A(i_iv * 4 + 4) = 0
S-22        end do
S-23    end subroutine
S-24
S-25    subroutine unroll3_partial_nofactor(A)
S-26       implicit none
S-27       double precision :: A(*)
S-28       integer :: i
S-29
S-30       !$omp unroll partial
S-31       do i = 1, 128
S-32          A(i) = 0
S-33       end do
S-34    end subroutine
```

Fortran

When the iteration count is not a multiple of the *unroll-factor*, iterations that should not produce executions must be conditionally protected from execution. In this example, the first function unrolls a loop that has a variable iteration count. Since the **unroll** construct uses a **partial(***4***)** clause, the compiler will need to create code that can account for cases when the iteration count is not a multiple of 4. A brute-force, simple-to-understand approach for implementing the conditionals is shown in the *unroll_partial_remainder_option1* function.

The remaining two functions show more optimal algorithms the compiler may select to implement the transformation. Optimal approaches may reduce the number of conditionals as shown in *unroll_partial_remainder_option2*, and may eliminate conditionals completely by peeling off a "remainder" into a separate loop as in *unroll_partial_remainder_option3*.

Regardless of the optimization, implementations must ensure that the semantics remain the same, especially when additional directives are applied to the unrolled loop. For the case in the *unroll_partial_remainder_option3* function, the fission of the worksharing-loop construct may result in a different distribution of threads to the iterations. Since no reproducible scheduling is specified on the work-sharing construct, the worksharing-loop and unrolling are compliant.

C / C++

Example unroll.4.c (**omp_5.1**)

```
S-1     void unroll_partial_remainder(int n, int A[])
S-2     {
S-3         #pragma omp parallel for
S-4         #pragma omp unroll partial(4)
S-5         for (int i = 0; i < n; ++i)
S-6             A[i] = i;
S-7     }
S-8
```

```
 S-9   void unroll_partial_remainder_option1(int n, int A[])
S-10   {
S-11       #pragma omp parallel for
S-12       for (int i_iv = 0; i_iv < (n+3)/4; ++i_iv) {
S-13                                  A[i_iv * 4 + 0] = i_iv * 4 + 0;
S-14           if (i_iv * 4 + 1 < n) A[i_iv * 4 + 1] = i_iv * 4 + 1;
S-15           if (i_iv * 4 + 2 < n) A[i_iv * 4 + 2] = i_iv * 4 + 2;
S-16           if (i_iv * 4 + 3 < n) A[i_iv * 4 + 3] = i_iv * 4 + 3;
S-17      }
S-18   }
S-19
S-20   void unroll_partial_remainder_option2(int n, int A[])
S-21   {
S-22       #pragma omp parallel for
S-23       for (int i_iv = 0; i_iv < (n+3)/4; ++i_iv) {
S-24           if (i_iv < n/4) {
S-25               A[i_iv * 4 + 0] = i_iv * 4 + 0;
S-26               A[i_iv * 4 + 1] = i_iv * 4 + 1;
S-27               A[i_iv * 4 + 2] = i_iv * 4 + 2;
S-28               A[i_iv * 4 + 3] = i_iv * 4 + 3;
S-29           } else {
S-30               // remainder loop
S-31               for (int i_rem = i_iv*4; i_rem < n; ++i_rem)
S-32                   A[i_rem] = i_rem;
S-33           }
S-34       }
S-35   }
S-36
S-37   void unroll_partial_remainder_option3(int n, int A[])
S-38   {
S-39       // main loop
S-40       #pragma omp parallel for
S-41       for (int i_iv = 0; i_iv < n/4; ++i_iv) {
S-42           A[i_iv * 4 + 0] = i_iv * 4 + 0;
S-43           A[i_iv * 4 + 1] = i_iv * 4 + 1;
S-44           A[i_iv * 4 + 2] = i_iv * 4 + 2;
S-45           A[i_iv * 4 + 3] = i_iv * 4 + 3;
S-46       }
S-47
S-48       // remainder loop
S-49       #pragma omp parallel for
S-50       for (int i_rem = (n/4)*4; i_rem < n; ++i_rem)
S-51           A[i_rem] = i_rem;
S-52   }
S-53
S-54   #include <stdio.h>
S-55   #define NT 12
```

```
S-56
S-57    int main(){
S-58    int error=0, A[NT],C[NT];
S-59    for(int i = 0; i<NT; i++){ A[i]=0; C[i]=i; }
S-60
S-61        for(int i = 0; i<NT; i++) A[i]=0.0;
S-62        unroll_partial_remainder(NT,A);
S-63        for(int i = 0; i<NT; i++) if(A[i] != C[i]) error=1;
S-64
S-65        for(int i = 0; i<NT; i++) A[i]=0.0;
S-66        unroll_partial_remainder_option1(NT,A);
S-67        for(int i = 0; i<NT; i++) if(A[i] != C[i]) error=1;
S-68
S-69        for(int i = 0; i<NT; i++) A[i]=0.0;
S-70        unroll_partial_remainder_option2(NT,A);
S-71        for(int i = 0; i<NT; i++) if(A[i] != C[i]) error=1;
S-72
S-73        for(int i = 0; i<NT; i++) A[i]=0.0;
S-74        unroll_partial_remainder_option3(NT,A);
S-75        for(int i = 0; i<NT; i++) if(A[i] != C[i]) error=1;
S-76
S-77      if(!error) printf("OUT: Passed\n");
S-78      if( error) printf("OUT: Failed\n");
S-79    }
```

C / C++

Fortran

1 *Example unroll.4.f90* (`omp_5.1`)

```
 S-1    subroutine unroll_partial_remainder(n, A)
 S-2       implicit none
 S-3       integer :: n, i
 S-4       integer :: A(*)
 S-5
 S-6        !$omp parallel do
 S-7        !$omp unroll partial(4)
 S-8        do i = 1, n
 S-9           A(i) = i
S-10        end do
S-11
S-12    end subroutine
S-13
S-14    subroutine unroll_partial_remainder_option1(n, A)
S-15       implicit none
S-16       integer :: n, i_iv
S-17       integer :: A(*)
S-18
```

```
S-19        !$omp parallel do
S-20        do i_iv = 0,(n+3)/4 -1
S-21                                    A(i_iv * 4 + 1) = i_iv * 4 + 1
S-22             if (i_iv * 4 + 2 <= n) A(i_iv * 4 + 2) = i_iv * 4 + 2
S-23             if (i_iv * 4 + 3 <= n) A(i_iv * 4 + 3) = i_iv * 4 + 3
S-24             if (i_iv * 4 + 4 <= n) A(i_iv * 4 + 4) = i_iv * 4 + 4
S-25        end do
S-26
S-27     end subroutine
S-28
S-29     subroutine unroll_partial_remainder_option2(n, A)
S-30        implicit none
S-31        integer :: n, i_iv, i_rem
S-32        integer :: A(*)
S-33
S-34         !$omp parallel do
S-35         do i_iv = 0, (n+3)/4 -1
S-36             if (i_iv < n/4) then
S-37                 A(i_iv * 4 + 1) = i_iv * 4 + 1
S-38                 A(i_iv * 4 + 2) = i_iv * 4 + 2
S-39                 A(i_iv * 4 + 3) = i_iv * 4 + 3
S-40                 A(i_iv * 4 + 4) = i_iv * 4 + 4
S-41             else
S-42                !! remainder loop
S-43                do i_rem = i_iv*4 +1, n
S-44                   A(i_rem) = i_rem
S-45                end do
S-46             end if
S-47         end do
S-48
S-49     end subroutine
S-50
S-51     subroutine unroll_partial_remainder_option3(n, A)
S-52        implicit none
S-53        integer :: n, i_iv, i_rem
S-54        integer :: A(*)
S-55
S-56         !$omp parallel do
S-57         do i_iv = 0, (n/4) -1
S-58
S-59            A(i_iv * 4 + 1) = i_iv * 4 + 1
S-60            A(i_iv * 4 + 2) = i_iv * 4 + 2
S-61            A(i_iv * 4 + 3) = i_iv * 4 + 3
S-62            A(i_iv * 4 + 4) = i_iv * 4 + 4
S-63         end do
S-64
S-65         !! remainder loop
```

```
S-66        !$omp parallel do
S-67        do i_rem = (n/4)*4 +1, n
S-68           A(i_rem) = i_rem
S-69        end do
S-70
S-71     end subroutine
S-72
S-73     program main
S-74     implicit none
S-75     integer, parameter :: NT=12
S-76
S-77     integer :: i
S-78     logical :: error=.false.
S-79     integer   :: A(NT), C(NT)=[ (i, i=1,NT) ]
S-80
S-81        A(1:NT)=0
S-82        call unroll_partial_remainder(NT, A)
S-83        if( .not. all(A(1:NT) == C(1:NT)) ) error = .true.
S-84
S-85        A(1:NT)=0
S-86        call unroll_partial_remainder_option1(NT, A)
S-87        if( .not. all(A(1:NT) == C(1:NT)) ) error = .true.
S-88
S-89        A(1:NT)=0
S-90        call unroll_partial_remainder_option2(NT, A)
S-91        if( .not. all(A(1:NT) == C(1:NT)) ) error = .true.
S-92
S-93        A(1:NT)=0
S-94        call unroll_partial_remainder_option3(NT, A)
S-95        if( .not. all(A(1:NT) == C(1:NT)) ) error = .true.
S-96
S-97        if(.not. error) print*, "OUT: Passed."
S-98        if(      error) print*, "OUT: Failed"
S-99     end program
```

Fortran

8.4 `apply` Clause

A loop transformation construct can be applied to another nested loop transformation construct, but the application of the "outer" transformation is limited to the outermost generated loop of the "inner" transformation.

The **`apply`** clause on a loop transformation construct can specify additional loop transformation directives that apply to generated loops other than the outermost one. Clause modifiers are used to specify which generated loop to target. Also, an applied directive within a clause may specify another **`apply`** clause.

Any nested loop transformation constructs including any constructs that result from **`apply`** clauses of nested constructs are replaced before any enclosing loop transformation construct. This is referred to as the *innermost-first order* here.

8.4.1 Syntax and Effect

In the example below, the *`construct_unroll`* and *`apply_unroll`* functions illustrate the syntax for two equivalent means of applying the **`unroll`** loop transformation directive to the outermost generated (grid) loop of the **`tile`** construct transformation. In function *`construct_unroll`*, the tile transformation creates the generated (tiled) loops and then the **`unroll`** construct is applied to outermost loop of the replacement. In the *`apply_unroll`* function, the **`apply`** clause on the **`tile`** construct is used to apply an **`unroll`** transformation on the *grid* loop (the outermost loop of the tile transformation) as specified by the **`grid`** modifier.

C / C++

Example apply_syntax.1.c (**`omp_6.0`**)

```
S-1   void construct_unroll(double A[100])
S-2   {
S-3      #pragma omp unroll
S-4      #pragma omp tile sizes(4)
S-5      for (int i = 0; i < 100; ++i)
S-6         A[i] = A[i] + 1;
S-7   }
S-8
S-9   void apply_unroll(double A[100])
S-10  {
S-11     #pragma omp tile sizes(4) apply(grid: unroll)
S-12     for (int i = 0; i < 100; ++i)
S-13        A[i] = A[i] + 1;
S-14  }
```

C / C++

Fortran

Example apply_syntax.1.f90 (`omp_6.0`)

```
S-1   subroutine construct_unroll(A)
S-2      implicit none
S-3      integer :: i
S-4      double precision :: A(0:99)
S-5
S-6      !$omp unroll
S-7      !$omp tile sizes(4)
S-8      do i = 0, 99
S-9         A(i) = A(i) + 1
S-10     end do
S-11  end subroutine
S-12
S-13  subroutine apply_unroll(A)
S-14     implicit none
S-15     integer :: i
S-16     double precision :: A(0:99)
S-17
S-18     !$omp tile sizes(4) apply(grid: unroll)
S-19     do i = 0, 99
S-20        A(i) = A(i) + 1
S-21     end do
S-22  end subroutine
```

Fortran

For the two functions in the previous example, the *equivalent* function in the next example shows an equivalent code that a user could have written without using the **tile** construct or **apply** clause.

C / C++

Example apply_syntax_equivalent.1.c (`omp_5.1`)

```
S-1   void equivalent(double A[100])
S-2   {
S-3      #pragma omp unroll
S-4      for (int i1 = 0; i1 < 25; ++i1)
S-5      for (int i2 = 0; i2 <  4; ++i2) {
S-6         int i = i1 * 4 + i2;
S-7         A[i] = A[i] + 1;
S-8      }
S-9   }
```

C / C++

Fortran

Example apply_syntax_equivalent.1.f90 (**omp_5.1**)

```
S-1   subroutine equivalent(A)
S-2      implicit none
S-3      double precision :: A(0:99)
S-4      integer          :: i1,i2, i
S-5
S-6      !$omp unroll
S-7      do i1=0,24
S-8      do i2=0, 3
S-9         i = i1 * 4 + i2
S-10        A(i) = A(i) + 1
S-11     enddo; enddo
S-12
S-13  end subroutine
```

Fortran

The following example shows how multiple loop transformation directives can be applied to different generated loops resulting from a loop transformation. For the 4x4 **tile** construct there will be two (outer) *grid* loops and two (inner) *intra-tile* loops. The first **apply** clause specifies that the two *grid* loops are to have an **interchange** directive and a **nothing** directive (just a placeholder to indicate no directive application) applied to the grid (two outermost) loops. Directives, read from left to right, are applied to the *grid* loops, from outermost to innermost, respectively. The second **apply** clause specifies that the two *intratile* loops are to have **nothing** and **interchange** directives applied to the last two *tile* loops, respectively. Note that the `A` array dimensions are `A[100][100][3]` and `A(0:2,0:99,0:99)` in the C/C++ and Fortran codes to illustrate equivalent sequential memory access for the `i`, `j` and `k` loops.

C / C++

Example apply_syntax.2.c (**omp_6.0**)

```
S-1   void apply_assoc(double A[100][100][3])
S-2   {
S-3      #pragma omp tile sizes(4,4) \
S-4                  apply(     grid: interchange,nothing) \
S-5                  apply(intratile: nothing,interchange)
S-6      for (int i = 0; i < 100; ++i)
S-7      for (int j = 0; j < 100; ++j)
S-8
S-9         // k loop not associated with tile, but with interchange
S-10        for (int k = 0; k < 3; ++k)
S-11           A[i][j][k] = A[i][j][k] + 1;
S-12  }
```

C / C++

Fortran

Example apply_syntax.2.f90 (**omp_6.0**)

```
S-1   subroutine apply_assoc(A)
S-2      implicit none
S-3      double precision :: A(0:2, 0:99, 0:99)
S-4      integer          ::    k,   j,    i
S-5
S-6      !$omp  tile sizes(4,4) &
S-7      !$omp&      apply(     grid: interchange, nothing) &
S-8      !$omp&      apply(intratile: nothing, interchange)
S-9      do i = 0, 99
S-10     do j = 0, 99
S-11
S-12        do k = 0, 2 !! k loop not associated with tile, but w. interchange
S-13           A(k,j,i) = A(k,j,i) + 1
S-14        enddo
S-15     enddo; enddo
S-16  end subroutine
```

Fortran

For the function in the previous example, the *equivalent* function in the next example shows a possible equivalent tile replacement code (**tile** generated loops) and the appropriately positioned **interchange** and **nothing** directives.

C / C++

Example apply_syntax_equivalent.2.c (**omp_6.0**)

```
S-1   void equivalent(double A[100][100][3])
S-2   {
S-3     #pragma omp interchange
S-4     for (int i1 = 0; i1 < 25; ++i1)
S-5     #pragma omp nothing
S-6     for (int j1 = 0; j1 < 25; ++j1)
S-7
S-8        #pragma omp nothing
S-9        for (int i2 = 0; i2 < 4; ++i2)
S-10       #pragma omp interchange
S-11       for (int j2 = 0; j2 < 4; ++j2)
S-12
S-13          for (int k = 0; k < 3; ++k) {
S-14             int i = i1 * 4 + i2;
S-15             int j = j1 * 4 + j2;
S-16             A[i][j][k] = A[i][j][k] + 1;
S-17          }
S-18  }
```

C / C++

Fortran

Example apply_syntax_equivalent.2.f90 (**omp_6.0**)

```
S-1   subroutine equivalent(A)
S-2      implicit none
S-3      double precision :: A(0:2,   0:99,   0:99)
S-4      integer          ::   k,    j1,j2,  i1,i2
S-5
S-6      !$omp interchange       !! grid modifier
S-7      do i1 = 0, 24
S-8      !$omp nothing           !! grid modifier
S-9      do j1 = 0, 24
S-10
S-11        !$omp nothing        !! intratile modifier
S-12        do i2 = 0, 3
S-13        !$omp interchange    !! intratile modifier
S-14        do j2 = 0, 3
S-15
S-16           do k = 0, 2
S-17              i = i1 * 4 + i2
S-18              j = j1 * 4 + j2
S-19              A(k,j,i) = A(k,j,i) + 1
S-20           enddo
S-21
S-22        enddo; enddo
S-23     enddo; enddo
S-24  end subroutine
```

Fortran

The following example illustrates the use of **apply** clause modifiers with argument. The index of the generated loop instead of a positional location can be used for the applied-directive. The **grid(1)** modifier indicates the first grid loop generated by the **tile** directive and the **intratile(2)** modifier indicates the second tile loop generated by the **tile** directive.

C / C++

Example apply_syntax.3.c (**omp_6.0**)

```
S-1   void apply_complexarg(double A[100*100])
S-2   {
S-3      #pragma omp tile sizes(4,5) \
S-4         apply(grid(1): reverse)  \
S-5         apply(intratile(2): unroll)
S-6      for (int i = 0; i < 100; ++i)
S-7         for (int j = 0; j < 100; ++j)
S-8            A[i*100+j] += 1;
S-9   }
```

C / C++

Fortran

Example apply_syntax.3.f90 (`omp_6.0`)

```
S-1   subroutine apply_complexarg(A)
S-2      implicit none
S-3      double precision :: A(100,100)
S-4      integer :: i, j
S-5
S-6      !$omp tile sizes(4,5)             &
S-7      !$omp&   apply(grid(1): reverse) &
S-8      !$omp&   apply(intratile(2): unroll)
S-9      do i = 1, 100
S-10        do j = 1, 100
S-11           A(j,i) = A(j,i) + 1
S-12        end do
S-13     end do
S-14  end subroutine
```

Fortran

Without the index arguments, the **`nothing`** argument would be needed as a placeholder, as illustrated by the equivalent codes of the above example as follows.

C / C++

Example apply_syntax_equivalent.3.c (`omp_6.0`)

```
S-1   void apply_complexarg_equivalent1(double A[100*100])
S-2   {
S-3      #pragma omp tile sizes(4,5)     \
S-4         apply(grid: reverse,nothing) \
S-5         apply(intratile: nothing,unroll)
S-6      for (int i = 0; i < 100; ++i)
S-7         for (int j = 0; j < 100; ++j)
S-8            A[i*100+j] += 1;
S-9   }
S-10
S-11  void apply_complexarg_equivalent2(double A[100*100])
S-12  {
S-13     #pragma omp reverse
S-14     for (int i1 = 0; i1 < 100; i1+=4)          // grid loop 1
S-15        for (int j1 = 0; j1 < 100; j1+=5)       // grid loop 2
S-16           for (int i = i1; i < i1+4; i+=1)     // tile loop 1
S-17              #pragma omp unroll
S-18              for (int j = j1; j < j1+5; j+=1)  // tile loop 2
S-19                 A[i*100+j] += 1;
S-20  }
```

C / C++

Fortran

Example apply_syntax_equivalent.3.f90 (`omp_6.0`)

```
S-1   subroutine apply_complexarg_equivalent1(A)
S-2      implicit none
S-3      double precision :: A(100,100)
S-4      integer :: i, j
S-5
S-6      !$omp tile sizes(4,5)                  &
S-7      !$omp&   apply(grid: reverse,nothing) &
S-8      !$omp&   apply(intratile: nothing,unroll)
S-9      do i = 1, 100
S-10        do j = 1, 100
S-11           A(j,i) = A(j,i) + 1
S-12        end do
S-13     end do
S-14  end subroutine
S-15
S-16  subroutine apply_complexarg_equivalent2(A)
S-17     implicit none
S-18     double precision :: A(100,100)
S-19     integer :: i, j, i1, j1
S-20
S-21     !$omp reverse
S-22     do i1 = 1, 100, 4          ! grid loop 1
S-23        do j1 = 1, 100, 5       ! grid loop 2
S-24           do i = i1, i1+3      ! tile loop 1
S-25              !$omp unroll
S-26              do j = j1, j1+4   ! tile loop 2
S-27                 A(j,i) = A(j,i) + 1
S-28              end do
S-29           end do
S-30        end do
S-31     end do
S-32  end subroutine
```

Fortran

8.4.2 Spanning Loop Associations

It is possible for a loop transformation directive to be applied to multiple generated loops, and multiple directives applied to the same generated loop. The latter is illustrated in the this example.

C / C++

Example apply_span.1.c (`omp_6.0`)

```
S-1   void span_apply(double A[128][128])
S-2   {
S-3      #pragma omp for collapse(2)
S-4      #pragma omp tile sizes(16,16) \
S-5                  apply(grid: interchange,reverse)
S-6      for (int i = 0; i < 128; ++i)
S-7         for (int j = 0; j < 128; ++j)
S-8            A[i][j] = A[i][j] + 1;
S-9   }
```

C / C++

Fortran

Example apply_span.1.f90 (`omp_6.0`)

```
S-1   subroutine span_apply( A )
S-2      implicit none
S-3      double precision :: A(0:127,0:127)
S-4      integer          ::   i   , j
S-5
S-6       !$omp for collapse(2)
S-7       !$omp tile sizes(16,16) apply(grid: interchange,reverse)
S-8       do i = 0, 127
S-9       do j = 0, 127
S-10        A(j,i) = A(j,i) + 1
S-11      enddo; enddo
S-12
S-13  end subroutine
```

Fortran

In this example, the functions show successive steps in the application of the previous loop transformation example as equivalent user-written code. First, the tiling is applied in the *step1* function. Next, loop transformations in the generated loop nest are replaced according to the innermost-first order rule. Applying the innermost transformation, loop reversal, results in the loop nest in *step2*. After that, the inner tile directive is applied in the *step3* function.

Example apply_span_equivalent.1.c (`omp_6.0`)

```
S-1   void step1(double A[128][128])
S-2   {
S-3      #pragma omp for collapse(2)
S-4      #pragma omp interchange
S-5      for (int i1 = 0; i1 < 8; ++i1)
S-6      #pragma omp reverse
S-7      for (int j1 = 0; j1 < 8; ++j1)
S-8
S-9         for (int i2 = 0; i2 < 16; ++i2)
S-10        for (int j2 = 0; j2 < 16; ++j2) {
S-11           int i = i1 * 16 + i2;
S-12           int j = j1 * 16 + j2;
S-13           A[i][j] = A[i][j] + 1;
S-14        }
S-15  }
S-16
S-17  void step2(double A[128][128])
S-18  {
S-19     #pragma omp for collapse(2)
S-20     #pragma omp interchange
S-21     for (int i1 =  0; i1 < 8; ++i1)
S-22     for (int j1 = 7; j1 >= 0; --j1)
S-23
S-24        for (int i2 = 0; i2 < 16; ++i2)
S-25        for (int j2 = 0; j2 < 16; ++j2) {
S-26           int i = i1 * 16 + i2;
S-27           int j = j1 * 16 + j2;
S-28           A[i][j] = A[i][j] + 1;
S-29        }
S-30  }
S-31
S-32  void step3(double A[128][128])
S-33  {
S-34     #pragma omp for collapse(2)
S-35     for (int j1 = 7; j1 >= 0; --j1)
S-36     for (int i1 =  0; i1 < 8; ++i1)
S-37
S-38        for (int i2 = 0; i2 < 16; ++i2)
S-39        for (int j2 = 0; j2 < 16; ++j2) {
S-40           int i = i1 * 16 + i2;
S-41           int j = j1 * 16 + j2;
S-42           A[i][j] = A[i][j] + 1;
S-43        }
```

```
}
```

C / C++

Fortran

Example apply_span_equivalent.1.f90 (**omp_6.0**)

```
subroutine step1(A)
   implicit none
   double precision :: A(0:127, 0:127)
   integer          :: i,i1,i2, j,j1,j2

   !$omp do collapse(2)
   !$omp interchange
   do i1 = 0, 7
   !$omp reverse
   do j1 = 0, 7

      do i2 = 0, 15
      do j2 = 0, 15
         i = i1 * 16 + i2
         j = j1 * 16 + j2
         A(j,i) = A(j,i) + 1
      enddo; enddo
   enddo; enddo

end subroutine

subroutine step2(A)
   implicit none
   double precision :: A(0:127, 0:127)
   integer          :: i,i1,i2, j,j1,j2

   !$omp do collapse(2)
   !$omp interchange
   do i1 = 0, 7
   do j1 = 7, 0, -1

      do i2 = 0, 15
      do j2 = 0, 15
         i = i1 * 16 + i2
         j = j1 * 16 + j2
         A(j,i) = A(j,i) + 1
      enddo; enddo
   enddo; enddo

end subroutine
```

```
subroutine step3(A)
   implicit none
   double precision :: A(0:127, 0:127)
   integer          :: i,i1,i2, j,j1,j2

   !$omp do collapse(2)
   do j1 = 7, 0, -1
   do i1 = 0, 7

      do i2 = 0, 15
      do j2 = 0, 15
         i = i1 * 16 + i2
         j = j1 * 16 + j2
         A(j,i) = A(j,i) + 1
      enddo; enddo
   enddo; enddo

end subroutine
```

Fortran

8.4.3 Nested apply

The following example illustrates how multiple loop transformations can be chained by nesting **apply** clauses. In the *nested_apply* function, a loop is first tiled, then the intra-tile loop is unrolled, and finally the iteration order of the unrolled loop is reversed. For C/C++ codes, reversing a loop with an unsigned type index may cause the compiler to ensure that underflow is handled correctly.

C / C++

Example apply_nested.1.c (**omp_6.0**)

```
void nested_apply(double A[100])
{
    #pragma omp tile sizes(10) \
            apply(intratile: unroll partial(2) apply(reverse))
    for (int i = 0; i < 100; ++i)
        A[i] = A[i] + 1;
}
```

C / C++

Fortran

Example apply_nested.1.f90 (`omp_6.0`)

```
S-1   subroutine nested_apply(A)
S-2      implicit none
S-3      double precision :: A(0:99)
S-4      integer          :: i
S-5
S-6      !$omp tile sizes(10) apply(intratile: unroll partial(2) apply(reverse))
S-7      do i = 0, 99
S-8         A(i) = A(i) + 1
S-9      enddo
S-10  end subroutine
```

Fortran

In this example the *step1*, *step2* and *step3* functions are all equivalent to the *nested_apply* function, but illustrate a possible chain of transformations but done manually by a user.

C / C++

Example apply_nested_equivalent.1.c (`omp_6.0`)

```
S-1   void step1(double A[100])
S-2   {
S-3      for (int i1 = 0; i1 < 10; ++i1)
S-4      #pragma omp unroll partial(2) apply(reverse)
S-5      for (int i2 = 0; i2 < 10; ++i2) {
S-6         int i = i1 * 10 + i2;
S-7         A[i] = A[i] + 1;
S-8      }
S-9   }
S-10
S-11  void step2(double A[100])
S-12  {
S-13     for (int i1 = 0; i1 < 10; ++i1)
S-14     #pragma omp reverse
S-15     for (int i2 = 0; i2 < 5; ++i2) {
S-16        int i = i1 * 10 + i2 * 2;
S-17        A[i] = A[i] + 1;
S-18        ++i;
S-19        A[i] = A[i] + 1;
S-20     }
S-21  }
S-22
S-23  void step3(double A[100])
S-24  {
S-25     for (int i1 = 0; i1 < 10; ++i1)
```

```
S-26        for (int i2 = 4; i2 >= 0; --i2) {
S-27           int i = i1 * 10 + i2 * 2;
S-28           A[i] = A[i] + 1;
S-29           ++i;
S-30           A[i] = A[i] + 1;
S-31        }
S-32     }
```

C / C++

Fortran

1 *Example apply_nested_equivalent.1.f90* (`omp_6.0`)

```
S-1      subroutine step1(A)
S-2         implicit none
S-3         double precision :: A(0:99)
S-4         integer          :: i,i1,i2
S-5
S-6          do i1 = 0, 9
S-7          !$omp unroll partial(2) apply(reverse)
S-8          do i2 = 0, 9
S-9             i = i1 * 10 + i2
S-10            A(i) = A(i) + 1
S-11         enddo; enddo
S-12     end subroutine
S-13
S-14     subroutine step2(A)
S-15        implicit none
S-16        double precision :: A(0:99)
S-17        integer          :: i,i1,i2
S-18
S-19         do i1 = 0, 9
S-20         !$omp reverse
S-21         do i2 = 0, 4
S-22            i = i1 * 10 + i2 * 2
S-23            A(i) = A(i) + 1
S-24            i = i + 1
S-25            A(i) = A(i) + 1
S-26         enddo; enddo
S-27     end subroutine
S-28
S-29     subroutine step3(A)
S-30        implicit none
S-31        double precision :: A(0:99)
S-32        integer          :: i,i1,i2
S-33
S-34         do i1 = 0, 9
S-35         do i2 = 4, 0, -1
```

```
S-36            i = i1 * 10 + i2 * 2
S-37            A(i) = A(i) + 1
S-38            i = i + 1
S-39            A(i) = A(i) + 1
S-40         enddo; enddo
S-41    end subroutine
```

Fortran

9 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 is not required for **update** statements. The *memory-order* clause can be used to specify the degree of memory ordering enforced by an **atomic** construct. From weakest to strongest, they are **relaxed** (the default), *acquire* and/or *release* clauses (specified with **acquire**, **release**, or **acq_rel**), and **seq_cst**. 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 an 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 for enforcing consistency between a thread's view of memory and the view of memory for other threads (see the Memory Model chapter of this document for more details). When the construct is used with an explicit variable list, a *strong flush* that forces a thread's temporary view of memory to be consistent with the actual memory is applied to all listed variables. When the construct is used without an explicit variable list and without a *memory-order* clause, a strong flush is applied to all locally thread-visible data as defined by the base language, and additionally the construct provides both acquire and release memory ordering semantics. When an explicit variable list is not present and a *memory-order* clause is present, the construct provides acquire and/or release memory ordering semantics according to the

memory-order clause, but no strong flush is performed. A resulting strong flush that applies to a set of variables effectively ensures that no memory (load or store) operation for the affected variables 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.

9.1 `critical` Construct

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

C / C++

Example critical.1.c (**pre_omp_3.0**)

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

```
S-17      }
S-18
S-19   }
```

C / C++

Fortran

Example critical.1.f (**pre_omp_3.0**)

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

Fortran

The following example extends the previous example by adding the **hint** clause to the **critical** constructs.

C / C++

Example critical.2.c (`omp_5.0`)

```
S-1   #include <omp.h>
S-2
S-3   int dequeue(float *a);
S-4   void work(int i, float *a);
S-5
S-6   void critical_example(float *x, float *y)
S-7   {
S-8     int ix_next, iy_next;
S-9
S-10    #pragma omp parallel shared(x, y) private(ix_next, iy_next)
S-11    {
S-12      #pragma omp critical (xaxis) hint(omp_sync_hint_contended)
S-13        ix_next = dequeue(x);
S-14      work(ix_next, x);
S-15
S-16      #pragma omp critical (yaxis) hint(omp_sync_hint_contended)
S-17        iy_next = dequeue(y);
S-18      work(iy_next, y);
S-19    }
S-20
S-21  }
```

C / C++

Fortran

Example critical.2.f (`omp_5.0`)

```
S-1         SUBROUTINE CRITICAL_EXAMPLE(X, Y)
S-2           USE OMP_LIB        ! or INCLUDE "omp_lib.h"
S-3
S-4           REAL X(*), Y(*)
S-5           INTEGER IX_NEXT, IY_NEXT
S-6
S-7   !$OMP PARALLEL SHARED(X, Y) PRIVATE(IX_NEXT, IY_NEXT)
S-8
S-9   !$OMP CRITICAL(XAXIS) HINT(OMP_SYNC_HINT_CONTENDED)
S-10          CALL DEQUEUE(IX_NEXT, X)
S-11  !$OMP END CRITICAL(XAXIS)
S-12          CALL WORK(IX_NEXT, X)
S-13
S-14  !$OMP CRITICAL(YAXIS) HINT(OMP_SYNC_HINT_CONTENDED)
S-15          CALL DEQUEUE(IY_NEXT,Y)
S-16  !$OMP END CRITICAL(YAXIS)
S-17          CALL WORK(IY_NEXT, Y)
S-18
```

```
!$OMP END PARALLEL

      END SUBROUTINE CRITICAL_EXAMPLE
```

Fortran

9.2 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 primary thread 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 (**`pre_omp_3.0`**)

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

C / C++

Fortran

Example worksharing_critical.1.f (**pre_omp_3.0**)

```
      SUBROUTINE CRITICAL_WORK()

        INTEGER I
        I = 1

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

Fortran

9.3 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*.

C / C++

Example barrier_regions.1.c (`pre_omp_3.0`)

```
S-1   void work(int n) {}
S-2
S-3   void sub3(int n)
S-4   {
S-5     work(n);
S-6     #pragma omp barrier
S-7     work(n);
S-8   }
S-9
S-10  void sub2(int k)
S-11  {
S-12    #pragma omp parallel shared(k)
S-13      sub3(k);
S-14  }
S-15
S-16  void sub1(int n)
S-17  {
S-18    int i;
S-19    #pragma omp parallel private(i) shared(n)
S-20    {
S-21      #pragma omp for
S-22      for (i=0; i<n; i++)
S-23        sub2(i);
S-24    }
S-25  }
S-26
S-27  int main()
S-28  {
S-29    sub1(2);
S-30    sub2(2);
S-31    sub3(2);
S-32    return 0;
S-33  }
```

C / C++

Fortran

Example barrier_regions.1.f (`pre_omp_3.0`)

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

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

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


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

      PROGRAM EXAMPLE
        CALL SUB1(2)
        CALL SUB2(2)
        CALL SUB3(2)
      END PROGRAM EXAMPLE
```

Fortran

9.4 `atomic` Construct

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

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

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

C / C++

Example atomic.1.c (**`omp_3.1`**)

```
S-1   float work1(int i)
S-2   {
S-3     return 1.0 * i;
S-4   }
S-5
S-6   float work2(int i)
S-7   {
S-8      return 2.0 * i;
S-9   }
S-10
S-11  void atomic_example(float *x, float *y, int *index, int n)
S-12  {
S-13    int i;
S-14
S-15    #pragma omp parallel for shared(x, y, index, n)
S-16      for (i=0; i<n; i++) {
S-17        #pragma omp atomic update
S-18        x[index[i]] += work1(i);
S-19        y[i] += work2(i);
S-20       }
S-21  }
S-22
S-23  int main()
S-24  {
S-25    float x[1000];
S-26    float y[10000];
S-27    int index[10000];
S-28    int i;
S-29
S-30    for (i = 0; i < 10000; i++) {
S-31      index[i] = i % 1000;
S-32      y[i]=0.0;
S-33    }
```

```
S-34      for (i = 0; i < 1000; i++)
S-35        x[i] = 0.0;
S-36      atomic_example(x, y, index, 10000);
S-37      return 0;
S-38    }
```

C / C++

Fortran

1 *Example atomic.1.f* (`omp_3.1`)

```
 S-1          REAL FUNCTION WORK1(I)
 S-2            INTEGER I
 S-3            WORK1 = 1.0 * I
 S-4            RETURN
 S-5          END FUNCTION WORK1
 S-6
 S-7          REAL FUNCTION WORK2(I)
 S-8            INTEGER I
 S-9            WORK2 = 2.0 * I
S-10            RETURN
S-11          END FUNCTION WORK2
S-12
S-13          SUBROUTINE SUB(X, Y, INDEX, N)
S-14            REAL X(*), Y(*)
S-15            INTEGER INDEX(*), N
S-16
S-17            INTEGER I
S-18
S-19    !$OMP   PARALLEL DO SHARED(X, Y, INDEX, N)
S-20              DO I=1,N
S-21    !$OMP       ATOMIC UPDATE
S-22                  X(INDEX(I)) = X(INDEX(I)) + WORK1(I)
S-23              Y(I) = Y(I) + WORK2(I)
S-24              ENDDO
S-25
S-26          END SUBROUTINE SUB
S-27
S-28          PROGRAM ATOMIC_EXAMPLE
S-29            REAL X(1000), Y(10000)
S-30            INTEGER INDEX(10000)
S-31            INTEGER I
S-32
S-33            DO I=1,10000
S-34              INDEX(I) = MOD(I, 1000) + 1
S-35              Y(I) = 0.0
S-36            ENDDO
S-37
```

```
S-38            DO I = 1,1000
S-39              X(I) = 0.0
S-40            ENDDO
S-41
S-42            CALL SUB(X, Y, INDEX, 10000)
S-43
S-44          END PROGRAM ATOMIC_EXAMPLE
```

Fortran

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

C / C++

Example atomic.2.c (**omp_3.1**)

```
S-1   int atomic_read(const int *p)
S-2   {
S-3       int value;
S-4   /* Guarantee that the entire value of *p is read atomically. No part of
S-5    * *p can change during the read operation.
S-6    */
S-7   #pragma omp atomic read
S-8        value = *p;
S-9        return value;
S-10  }
S-11
S-12  void atomic_write(int *p, int value)
S-13  {
S-14  /* Guarantee that value is stored atomically into *p. No part of *p can
S-15  change
S-16   * until after the entire write operation is completed.
S-17   */
S-18  #pragma omp atomic write
S-19      *p = value;
S-20  }
```

C / C++

Fortran

Example atomic.2.f (**omp_3.1**)

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

Fortran

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 example could be implemented using the *fetch-and-add* instruction available on many kinds of hardware. The example also shows a way to implement a spin lock using the **capture** and **read** clauses.

C / C++

Example atomic.3.c (**omp_3.1**)

```
S-1   int fetch_and_add(int *p)
S-2   {
S-3      /* Atomically read the value of *p and then increment it. The
S-4         previous value is returned. This can be used to implement a
S-5         simple lock as shown below.
S-6       */
S-7      int old;
S-8   #pragma omp atomic capture
S-9      { old = *p; (*p)++; }
S-10     return old;
S-11  }
S-12
S-13  /*
```

```
 * Use fetch_and_add to implement a lock
 */
struct locktype {
   int ticketnumber;
   int turn;
};
void do_locked_work(struct locktype *lock)
{
   int atomic_read(const int *p);
   void work();

   // Obtain the lock
   int myturn = fetch_and_add(&lock->ticketnumber);
   while (atomic_read(&lock->turn) != myturn)
      ;
   // Do some work. The flush is needed to ensure visibility of
   // variables not involved in atomic directives

   #pragma omp flush
   work();
   #pragma omp flush
   // Release the lock
   fetch_and_add(&lock->turn);
}
```

C / C++

Fortran

Example atomic.3.f (`omp_3.1`)

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

! Atomically read the value of p and then increment it. The previous value
! is returned. This can be used to implement a simple lock as shown below.
!$omp atomic capture
      fetch_and_add = p
      p = p + 1
!$omp end atomic
      end function fetch_and_add
      module m
      interface
        function fetch_and_add(p)
          integer :: fetch_and_add
          integer, intent(inout) :: p
        end function
        function atomic_read(p)
```

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

Fortran

C / C++

9.5 Atomic Compare

In OpenMP 5.1 the **compare** clause was added to the *extended-atomic* clauses. The **compare** clause extends the semantics to perform the **atomic** update conditionally.

In the following C/C++ example, two formats of structured blocks are shown for associated **atomic** constructs with the **compare** clause. In the first **atomic** construct, the format forms a *conditional update* statement. In the second **atomic** construct the format forms a *conditional expression* statement. The "greater than" and "less than" forms are not available with the Fortran **compare** clause. One can use the `max` and `min` functions with the **atomic update** construct to perform the C/C++ example operations.

Example cas.1.c (**omp_5.1**)

```
S-1   #include <stdio.h>
S-2   #define N 10
S-3
S-4   void init(int *);
S-5
S-6   int main(){
S-7     int val_min=2*N, val_max=-2*N;
S-8     int val[N];
S-9
S-10    init(val);
S-11
S-12    #pragma omp parallel for num_threads(2)
S-13    for (int i=1; i<N-1; i++) {
S-14
S-15      // compare and update val_min using one atomic form
S-16      #pragma omp atomic compare
S-17      if (val[i] < val_min) { val_min = val[i]; }
S-18
S-19      // compare and update val_max using another atomic form
S-20      #pragma omp atomic compare
S-21      val_max = val[i] > val_max ? val[i] : val_max;
S-22    }
S-23
S-24    if(val_max != 2*N || val_min != -2*N){ printf("FAILED\n");}
S-25    else                                 { printf("PASSED\n");}
S-26    // OUT: PASSED
S-27  }
S-28
S-29  void init(int *val){
S-30    for (int i=0; i<N; i++) val[i]=i;
S-31    val[N/2  ] =  2*N;
S-32    val[N/2+1] = -2*N;
S-33  }
```

In OpenMP 5.1 the **compare** clause was also added to support *Compare And Swap* (CAS) semantics. In the following example the `enqueue` routine (a naive implementation of a Michael and Scott enqueue function), uses the **compare** clause, with the **capture** clause, to perform and compare (`q->head == node->next`) and swap (*if-else* assignments) of the form:

{ r = x == e; if(r) { x = d; } else { v = x; } }.

The example program concurrently enqueues nodes from an array of nodes (`nodes[N]`). Since the equivalence of Fortran pointers can be determined only with a function (such as associated), no Fortran version is provided here. The use of the associated function in an **atomic compare** syntax is being considered in a future release.

Example cas.2.c (**omp_5.1**)

```
S-1   #include <stdlib.h>
```

```
S-2     #include <stdio.h>
S-3     #include <stdbool.h>
S-4     #include <stddef.h>
S-5
S-6     #define N 10
S-7
S-8     typedef struct Node{ struct Node *next; int id; } Node;
S-9     typedef struct Queue{ Node *head; Node *tail;   } Queue;
S-10
S-11    void enqueue( Queue *,  Node * );
S-12
S-13    int main(){
S-14      Queue  q;
S-15      Node nodes[N];
S-16      int  id_check[N];
S-17
S-18    // Initializing
S-19      for(int i=0; i<N; i++){
S-20         nodes[i].next=NULL; nodes[i].id=i; id_check[i]=-1;
S-21      }
S-22
S-23      q.tail=&nodes[0];   // Fill initial tail
S-24
S-25    // Enqueue
S-26      #pragma omp parallel for num_threads(2)
S-27      for(int i=1; i<N; i++){
S-28         enqueue(&q,&nodes[i]);
S-29      }
S-30
S-31    // Checking Results Below
S-32      Node *node=q.tail;
S-33      do{
S-34         id_check[node->id]=node->id;  //Store found id at position id
S-35         node =node->next;
S-36      }while(node->next != NULL);
S-37      id_check[node->id]=node->id;    //checking also the 1st node here
S-38
S-39      for(int id=0; id<N; id++){     // all ids should be found
S-40         if(id != id_check[id]) {printf("FAILED\n"); exit(1);}
S-41      }
S-42      printf("PASSED\n");
S-43
S-44      return 0;
S-45    }
S-46
S-47    void enqueue(Queue *queue,  Node *node) {
S-48      bool result = false;
```

```
  #pragma omp atomic read
  node->next = queue->tail;
  do{
    #pragma omp atomic compare capture
    {
      result = queue->tail == node->next;
      if(result) {
        queue->tail = node;
      }else{
        node->next = queue->tail;
      }
    }
  }while(!result);
}
```

C / C++

9.6 Restrictions on the `atomic` Construct

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

C / C++

Example atomic_restrict.1.c (**`omp_3.1`**)

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

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

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

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

C / C++

Fortran

Example atomic_restrict.1.f (omp_3.1)

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

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

Fortran

C / C++

Example atomic_restrict.2.c (omp_3.1)

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

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

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

#pragma omp atomic update
    *r += 1.0;

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

  }
}
```

C / C++

Fortran

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

Example atomic_restrict.2.f (`omp_3.1`)

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

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

      SUBROUTINE ATOMIC_WRONG2()
        COMMON /BLK/ I
        INTEGER I

!$OMP   PARALLEL

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

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

Example atomic_restrict.3.f (`omp_3.1`)

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

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

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

```
      END SUBROUTINE ATOMIC_WRONG3
```

Fortran

9.7 Atomic Hint

The atomic **hint** clause can be used to specify the expected access to an atomic operation; thereby providing a hint to be used for optimizing the synchronization of the atomic operation.

In the example below the **omp_sync_hint_uncontended** constant in the **hint** clause specifies that few threads are expected to attempt to perform the atomic operation at the same time. This is justified in this case if *calc_vals* takes considerably more time than the atomic operations, and the subsequent time of arrival to execute the **atomic** region is varied about a mean time and by times (much) greater than the execution time of the atomic operation.

In the case where the execution time for *calc_vals* is short compared to the atomic operation time, the **omp_sync_hint_contended** hint parameter might be used.

C / C++

Example atomic.4.c (**omp_5.0**)

```
#include <omp.h>

void calc_val(float *val);

void boxster(float *box_totals, float *vals, int *box, int N)
{
    #pragma omp parallel for
    for(int idx=0; idx<=N; idx++)
    {
       calc_val(&vals[idx]);
       #pragma omp atomic hint(omp_sync_hint_uncontended)
       box_totals[ box[idx] ] = box_totals[ box[idx] ] + vals[idx];
    }
}
```

C / C++

Fortran

Example atomic.4.f90 (**omp_5.0**)

```
S-1   subroutine boxster(box_totals, vals, box, N)
S-2     use omp_lib
S-3     external calc_val
S-4     real,    intent(inout) :: box_totals(:)
S-5     real,    intent(in)    :: vals(:)
S-6     integer, intent(in)    :: box(:)
S-7     integer                :: N, idx
S-8
S-9       !$omp parallel do
S-10      do idx=1,N
S-11        call calc_val(vals(idx))
S-12        !$omp atomic hint(omp_sync_hint_uncontended)
S-13        box_totals( box(idx) ) = box_totals( box(idx) ) + vals(idx)
S-14      enddo
S-15
S-16  end subroutine
```

Fortran

9.8 Synchronization Based on Acquire/Release Semantics

As explained in the Memory Model chapter of this document, a flush operation may be an *acquire flush* and/or a *release flush*, and OpenMP 5.0 defines acquire/release semantics in terms of these fundamental flush operations. For any synchronization between two threads that is specified by OpenMP, a release flush logically occurs at the source of the synchronization and an acquire flush logically occurs at the sink of the synchronization. OpenMP 5.0 added memory ordering clauses – **acquire**, **release**, and **acq_rel** – to the **flush** and **atomic** constructs for explicitly requesting acquire/release semantics. Furthermore, implicit flushes for all OpenMP constructs and runtime routines that synchronize OpenMP threads in some manner were redefined in terms of synchronizing release and acquire flushes to avoid the requirement of strong memory fences (see the *Flush Synchronization and Happens Before* and *Implicit Flushes* sections of the OpenMP Specifications document).

The examples that follow in this section illustrate how acquire and release flushes may be employed, implicitly or explicitly, for synchronizing threads. A **flush** directive without a list and without any memory ordering clause can also function as both an acquire and release flush for facilitating thread synchronization. Flushes implied on entry to, or exit from, an atomic operation (specified by an **atomic** construct) may function as an acquire flush or a release flush if a memory ordering clause appears on the construct. On entry to and exit from a **critical** construct there is now an implicit acquire flush and release flush, respectively.

The first example illustrates how the release and acquire flushes implied by a **critical** region guarantee a value written by the first thread is visible to a read of the value on the second thread. Thread 0 writes to x and then executes a **critical** region in which it writes to y; the write to x happens before the execution of the **critical** region, consistent with the program order of the thread. Meanwhile, thread 1 executes a **critical** region in a loop until it reads a non-zero value from y in the **critical** region, after which it prints the value of x; again, the execution of the **critical** regions happen before the read from x based on the program order of the thread. The **critical** regions executed by the two threads execute in a serial manner, with a pair-wise synchronization from the exit of one **critical** region to the entry to the next **critical** region. These pair-wise synchronizations result from the implicit release flushes that occur on exit from **critical** regions and the implicit acquire flushes that occur on entry to **critical** regions; hence, the execution of each **critical** region in the sequence happens before the execution of the next **critical** region. A "happens before" order is therefore established between the assignment to x by thread 0 and the read from x by thread 1, and so thread 1 must see that x equals 10.

C / C++

Example acquire_release.1.c (**omp_5.0**)

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

int main()
{
   int x = 0, y = 0;
   #pragma omp parallel num_threads(2)
   {
      int thrd = omp_get_thread_num();
      if (thrd == 0) {
         x = 10;
         #pragma omp critical
         { y = 1; }
      } else {
         int tmp = 0;
         while (tmp == 0) {
           #pragma omp critical
           { tmp = y; }
         }
         printf("x = %d\n", x); // always "x = 10"
      }
   }
   return 0;
}
```

C / C++

Fortran

Example acquire_release.1.f90 (`omp_5.0`)

```
S-1   program rel_acq_ex1
S-2      use omp_lib
S-3      integer :: x, y, thrd, tmp
S-4      x = 0
S-5      y = 0
S-6      !$omp parallel num_threads(2) private(thrd, tmp)
S-7         thrd = omp_get_thread_num()
S-8         if (thrd == 0) then
S-9            x = 10
S-10           !$omp critical
S-11           y = 1
S-12           !$omp end critical
S-13        else
S-14           tmp = 0
S-15           do while (tmp == 0)
S-16              !$omp critical
S-17              tmp = y
S-18              !$omp end critical
S-19           end do
S-20           print *, "x = ", x  !! always "x = 10"
S-21        end if
S-22     !$omp end parallel
S-23  end program
```

Fortran

In the second example, the **`critical`** constructs are exchanged with **`atomic`** constructs that have *explicit* memory ordering specified. When the *atomic read* operation on thread 1 reads a non-zero value from y, this results in a release/acquire synchronization that in turn implies that the assignment to x on thread 0 happens before the read of x on thread 1. Therefore, thread 1 will print "x = 10".

C / C++

Example acquire_release.2.c (`omp_5.0`)

```
S-1   #include <stdio.h>
S-2   #include <omp.h>
S-3
S-4   int main()
S-5   {
S-6      int x = 0, y = 0;
S-7      #pragma omp parallel num_threads(2)
S-8      {
S-9         int thrd = omp_get_thread_num();
S-10         if (thrd == 0) {
```

```
S-11            x = 10;
S-12            #pragma omp atomic write release // or seq_cst
S-13            y = 1;
S-14         } else {
S-15            int tmp = 0;
S-16            while (tmp == 0) {
S-17              #pragma omp atomic read acquire // or seq_cst
S-18              tmp = y;
S-19            }
S-20            printf("x = %d\n", x); // always "x = 10"
S-21         }
S-22      }
S-23      return 0;
S-24   }
```

C / C++

Fortran

1 *Example acquire_release.2.f90* (`omp_5.0`)

```
 S-1   program rel_acq_ex2
 S-2      use omp_lib
 S-3      integer :: x, y, thrd, tmp
 S-4      x = 0
 S-5      y = 0
 S-6      !$omp parallel num_threads(2) private(thrd, tmp)
 S-7         thrd = omp_get_thread_num()
 S-8         if (thrd == 0) then
 S-9            x = 10
S-10            !$omp atomic write release ! or seq_cst
S-11             y = 1
S-12            !$omp end atomic
S-13         else
S-14            tmp = 0
S-15            do while (tmp == 0)
S-16               !$omp atomic read acquire ! or seq_cst
S-17               tmp = y
S-18               !$omp end atomic
S-19            end do
S-20            print *, "x = ", x  !! always "x = 10"
S-21         end if
S-22      !$omp end parallel
S-23   end program
```

Fortran

In the third example, **atomic** constructs that specify relaxed atomic operations are used with explicit **flush** directives to enforce memory ordering between the two threads. The explicit **flush** directive on thread 0 must specify a release flush and the explicit **flush** directive on thread 1 must specify an acquire flush to establish a release/acquire synchronization between the two threads. The **flush** and **atomic** constructs encountered by thread 0 can be replaced by the **atomic** construct used in Example 2 for thread 0, and similarly the **flush** and **atomic** constructs encountered by thread 1 can be replaced by the **atomic** construct used in Example 2 for thread 1.

C / C++

Example acquire_release.3.c (**omp_5.0**)

```
S-1   #include <stdio.h>
S-2   #include <omp.h>
S-3
S-4   int main()
S-5   {
S-6      int x = 0, y = 0;
S-7      #pragma omp parallel num_threads(2)
S-8      {
S-9         int thrd = omp_get_thread_num();
S-10        if (thrd == 0) {
S-11           x = 10;
S-12           #pragma omp flush // or with acq_rel or release clause
S-13           #pragma omp atomic write // or with relaxed clause
S-14           y = 1;
S-15        } else {
S-16           int tmp = 0;
S-17           while (tmp == 0) {
S-18             #pragma omp atomic read // or with relaxed clause
S-19             tmp = y;
S-20           }
S-21           #pragma omp flush // or with acq_rel or acquire clause
S-22           printf("x = %d\n", x);  // always "x = 10"
S-23        }
S-24     }
S-25     return 0;
S-26  }
```

C / C++

Fortran

Example acquire_release.3.f90 (`omp_5.0`)

```
S-1   program rel_acq_ex3
S-2      use omp_lib
S-3      integer :: x, y, thrd, tmp
S-4      x = 0
S-5      y = 0
S-6      !$omp parallel num_threads(2) private(thrd, tmp)
S-7         thrd = omp_get_thread_num()
S-8         if (thrd == 0) then
S-9            x = 10
S-10           !$omp flush ! or with acq_rel or release clause
S-11           !$omp atomic write
S-12           y = 1
S-13           !$omp end atomic
S-14        else
S-15           tmp = 0
S-16           do while (tmp == 0)
S-17              !$omp atomic read
S-18              tmp = y
S-19              !$omp end atomic
S-20           end do
S-21           !$omp flush ! or with acq_rel or acquire clause
S-22           print *, "x = ", x  !! always "x = 10"
S-23        end if
S-24     !$omp end parallel
S-25  end program
```

Fortran

Example 4 will fail to order the write to x on thread 0 before the read from x on thread 1. Importantly, the implicit release flush on exit from the **`critical`** region will not synchronize with the acquire flush that occurs on the *atomic read* operation performed by thread 1. This is because implicit release flushes that occur on a given construct may only synchronize with implicit acquire flushes on a compatible construct (and vice-versa) that internally makes use of the same synchronization variable. For a **`critical`** construct, this might correspond to a *lock* object that is used by a given implementation (for the synchronization semantics of other constructs due to implicit release and acquire flushes, refer to the *Implicit Flushes* section of the OpenMP Specifications document). Either an explicit **`flush`** directive that provides a release flush (i.e., a flush without a list that does not have the **`acquire`** clause) must be specified between the **`critical`** construct and the *atomic write*, or an atomic operation that modifies y and provides release semantics must be specified.

C / C++

Example acquire_release_broke.4.c (`omp_5.0`)

```
S-1   #include <stdio.h>
S-2   #include <omp.h>
S-3
S-4   int main()
S-5   {
S-6
S-7   // !!! THIS CODE WILL FAIL TO PRODUCE CONSISTENT RESULTS !!!!!!!
S-8   // !!! DO NOT PROGRAM SYNCHRONIZATION THIS WAY !!!!!!!
S-9
S-10     int x = 0, y;
S-11     #pragma omp parallel num_threads(2)
S-12     {
S-13        int thrd = omp_get_thread_num();
S-14        if (thrd == 0) {
S-15           #pragma omp critical
S-16           { x = 10; }
S-17           // an explicit flush directive that provides
S-18           // release semantics is needed here
S-19           // to complete the synchronization.
S-20           #pragma omp atomic write
S-21           y = 1;
S-22        } else {
S-23           int tmp = 0;
S-24           while (tmp == 0) {
S-25              #pragma omp atomic read acquire // or seq_cst
S-26              tmp = y;
S-27           }
S-28           #pragma omp critical
S-29           { printf("x = %d\n", x); }  // !! NOT ALWAYS 10
S-30        }
S-31     }
S-32     return 0;
S-33  }
```

C / C++

Fortran

Example acquire_release_broke.4.f90 (`omp_5.0`)

```
S-1   program rel_acq_ex4
S-2      use omp_lib
S-3      integer :: x, y, thrd
S-4      integer :: tmp
S-5      x = 0
S-6
S-7   !! !!! THIS CODE WILL FAIL TO PRODUCE CONSISTENT RESULTS !!!!!!!
S-8   !! !!! DO NOT PROGRAM SYNCHRONIZATION THIS WAY !!!!!!!
S-9
S-10     !$omp parallel num_threads(2) private(thrd) private(tmp)
S-11        thrd = omp_get_thread_num()
S-12        if (thrd == 0) then
S-13           !$omp critical
S-14           x = 10
S-15           !$omp end critical
S-16           ! an explicit flush directive that provides
S-17           ! release semantics is needed here to
S-18           ! complete the synchronization.
S-19           !$omp atomic write
S-20           y = 1
S-21           !$omp end atomic
S-22        else
S-23           tmp = 0
S-24           do while(tmp == 0)
S-25             !$omp atomic read acquire ! or seq_cst
S-26             tmp = x
S-27             !$omp end atomic
S-28           end do
S-29           !$omp critical
S-30           print *, "x = ", x  !! !! NOT ALWAYS 10
S-31           !$omp end critical
S-32        end if
S-33     !$omp end parallel
S-34  end program
```

Fortran

9.9 `ordered` Clause and `ordered` Construct

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

C / C++

Example ordered.1.c (**pre_omp_3.0**)

```
#include <stdio.h>

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

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

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

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

C / C++

Fortran

Example ordered.1.f (**pre_omp_3.0**)

```
      SUBROUTINE WORK(K)
        INTEGER k

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

      END SUBROUTINE WORK

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

```
S-13
S-14    !$OMP PARALLEL DO ORDERED SCHEDULE(DYNAMIC)
S-15            DO I=LB,UB,STRIDE
S-16              CALL WORK(I)
S-17            END DO
S-18    !$OMP END PARALLEL DO
S-19
S-20          END SUBROUTINE SUB
S-21
S-22          PROGRAM ORDERED_EXAMPLE
S-23            CALL SUB(1,100,5)
S-24          END PROGRAM ORDERED_EXAMPLE
```

Fortran

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

C / C++

4 *Example ordered.2.c* (**pre_omp_3.0**)

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

C / C++

Fortran

Example ordered.2.f (**pre_omp_3.0**)

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

Fortran

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 (**pre_omp_3.0**)

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

```
   }
}
```

C / C++

Fortran

Example ordered.3.f (**pre_omp_3.0**)

```
      SUBROUTINE ORDERED_GOOD(N)
      INTEGER N

!$OMP   DO ORDERED
        DO I = 1,N
          IF (I <= 10) THEN
!$OMP       ORDERED
              CALL WORK(I)
!$OMP       END ORDERED
          ENDIF

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

Fortran

9.10 `depobj` Construct

The stand-alone **depobj** construct provides a mechanism to create a *depend object* that expresses a dependence to be used subsequently in the **depend** clause of another construct. Dependence information is created from a dependence type and a storage location that is specified in the **depend** clause of an **depobj** construct, and it is stored in the depend object. The depend object is represented by a variable of type **omp_depend_t** in C/C++ and by a scalar variable of integer kind **omp_depend_kind** in Fortran.

In the example below the stand-alone **depobj** construct uses the **depend**, **update** and **destroy** clauses to *initialize*, *update* and *uninitialize* a depend object (`obj`).

The first **depobj** construct initializes the `obj` depend object with an **inout** dependence type and with a storage location defined by variable `a`. This dependence is passed into the `driver` routine via the `obj` depend object.

In the first *driver* routine call, *Task 1* uses the dependence of the object (**inout**), while *Task 2* uses an **in** dependence specified directly in a **depend** clause. For these task dependences *Task 1* must execute and complete before *Task 2* begins.

Before the second call to *driver*, *obj* is updated using the **depobj** construct to represent an **in** dependence. Hence, in the second call to *driver*, *Task 1* will have an **in** dependence; and *Task 1* and *Task 2* can execute simultaneously. Note: in an **update** clause, only the dependence type can be (is) updated.

The third **depobj** construct uses the **destroy** clause. It frees resources as it puts the depend object in an uninitialized state – effectively destroying the depend object. After an object has been uninitialized it can be initialized again with a new dependence type and a new variable.

C / C++

Example depobj.1.c (**omp_5.2**)

```
S-1   #include <stdio.h>
S-2   #include <omp.h>
S-3
S-4   #define N 100
S-5   #define TRUE  1
S-6   #define FALSE 0
S-7
S-8   void driver(int update, float a[], float b[], int n, omp_depend_t *obj);
S-9
S-10  void update_copy(int update, float a[], float b[], int n);
S-11  void checkpoint(float a[],int n);
S-12  void init(float a[], int n);
S-13
S-14
S-15  int main(){
S-16
S-17     float a[N],b[N];
S-18     omp_depend_t obj;
S-19
S-20     init(a, N);
S-21
S-22     #pragma omp depobj(obj) depend(inout: a)
S-23
S-24     driver(TRUE,  a,b,N, &obj);  // updating a occurs
S-25
S-26     #pragma omp depobj(obj) update(in)
S-27
S-28     driver(FALSE, a,b,N, &obj);  // no updating of a
S-29
S-30     #pragma omp depobj(obj) destroy(obj)  // obj is set to uninitialized
S-31                                           // state, resources are freed
```

```
S-32        return 0;
S-33
S-34     }
S-35
S-36     void driver(int update, float a[], float b[], int n, omp_depend_t *obj)
S-37     {
S-38        #pragma omp parallel num_threads(2)
S-39        #pragma omp single
S-40        {
S-41
S-42           #pragma omp task depend(depobj: *obj) // Task 1, uses depend object
S-43              update_copy(update, a,b,n); // may update a, always copy a to b
S-44
S-45         #pragma omp task depend(in: a[:n])     // Task 2, only read a
S-46              checkpoint(a,n);
S-47        }
S-48     }
S-49
S-50     void update_copy(int update, float a[], float b[], int n)
S-51     {
S-52        if(update) for(int i=0;i<n;i++) a[i]+=1.0f;
S-53
S-54        for(int i=0;i<n;i++) b[i]=a[i];
S-55     }
S-56
S-57     void checkpoint(float a[], int n)
S-58     {
S-59        for(int i=0;i<n;i++) printf(" %f ",a[i]);
S-60        printf("\n");
S-61     }
S-62
S-63     void init(float a[], int n)
S-64     {
S-65        for(int i=0;i<n;i++) a[i]=i;
S-66     }
```

C / C++

Fortran

1 *Example depobj.1.f90* (`omp_5.2`)

```
S-1      program main
S-2          use omp_lib
S-3          implicit none
S-4
S-5          integer,parameter        :: N=100
S-6          real                     :: a(N),b(N)
S-7          integer(omp_depend_kind) :: obj
```

```
S-8
S-9         call init(a, N)
S-10
S-11        !$omp depobj(obj) depend(inout: a)
S-12
S-13        call driver(.true.,  a,b,N, obj)  !! updating occurs
S-14
S-15        !$omp depobj(obj) update(in)
S-16
S-17        call driver(.false., a,b,N, obj)  !! no updating
S-18
S-19        !$omp depobj(obj) destroy(obj)    !! obj is set to uninitialized
S-20                                          !! state, resources are freed
S-21
S-22    end program
S-23
S-24    subroutine driver(update, a, b, n, obj)
S-25       use omp_lib
S-26       implicit none
S-27       logical :: update
S-28       real    :: a(n), b(n)
S-29       integer :: n
S-30       integer(omp_depend_kind) :: obj
S-31
S-32       !$omp parallel num_threads(2)
S-33
S-34         !$omp single
S-35
S-36           !$omp task depend(depobj: obj)       !! Task 1, uses depend object
S-37             call update_copy(update, a,b,n)
S-38                  !! update a or not, always copy a to b
S-39           !$omp end task
S-40
S-41           !$omp task depend(in: a)             !! Task 2, only read a
S-42             call checkpoint(a,n)
S-43           !$omp end task
S-44
S-45         !$omp end single
S-46
S-47       !$omp end parallel
S-48
S-49    end subroutine
S-50
S-51    subroutine update_copy(update, a, b, n)
S-52       implicit none
S-53       logical :: update
S-54       real    :: a(n), b(n)
```

```
S-55        integer :: n
S-56
S-57        if (update) a = a + 1.0
S-58
S-59        b = a
S-60
S-61     end subroutine
S-62
S-63     subroutine checkpoint( a, n)
S-64        implicit none
S-65        integer :: n
S-66        real    :: a(n)
S-67        integer :: i
S-68
S-69        write(*,'( *(f5.0) )') (a(i), i=1,n)
S-70     end subroutine
S-71
S-72     subroutine init(a,n)
S-73        implicit none
S-74        integer :: n
S-75        real    :: a(n)
S-76        integer :: i
S-77
S-78        a=[ (i, i=1,n) ]
S-79     end subroutine
```

Fortran

9.11 Doacross Loop Nest

An **ordered** clause can be used on a worksharing-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 **doacross** clause on an **ordered** construct within an *ordered* loop describes the dependences of the *doacross* loops.

In the code below, the **doacross(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 **doacross(source:omp_cur_iteration)** or **doacross(source:)** clause indicates the completion of computation from the current iteration (*i*) to satisfy the cross-iteration dependence that arises from the iteration. The **omp_cur_iteration** keyword is optional for the **source** dependence type. For this example the same sequential ordering could have been achieved with an **ordered** clause without a parameter on the worksharing-loop directive, and a single **ordered** directive without the **doacross** clause specified for the statement executing the *bar* function.

C / C++

1 *Example doacross.1.c* (omp_5.2)

```
S-1   float foo(int i);
S-2   float bar(float a, float b);
S-3   float baz(float b);
S-4
S-5   void work( int N, float *A, float *B, float *C )
S-6   {
S-7     int i;
S-8
S-9     #pragma omp for ordered(1)
S-10    for (i=1; i<N; i++)
S-11    {
S-12      A[i] = foo(i);
S-13
S-14    #pragma omp ordered doacross(sink: i-1)
S-15      B[i] = bar(A[i], B[i-1]);
S-16    #pragma omp ordered doacross(source: omp_cur_iteration)
S-17
S-18      C[i] = baz(B[i]);
S-19    }
S-20  }
```

C / C++

Fortran

2 *Example doacross.1.f90* (omp_5.2)

```
S-1   subroutine work( N, A, B, C )
S-2     integer :: N, i
S-3     real, dimension(N) :: A, B, C
S-4     real, external :: foo, bar, baz
S-5
S-6     !$omp do ordered(1)
S-7     do i=2, N
S-8       A(i) = foo(i)
S-9
S-10    !$omp ordered doacross(sink: i-1)
S-11      B(i) = bar(A(i), B(i-1))
S-12    !$omp ordered doacross(source: omp_cur_iteration)
S-13
S-14      C(i) = baz(B(i))
S-15    end do
S-16  end subroutine
```

Fortran

The following code is similar to the previous example but with the *doacross loop nest* extended to two nested loops, `i` and `j`, as specified by the **`ordered(2)`** clause on the worksharing-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 **`doacross(sink:`** `i-1, j`**`)`** and **`doacross(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 **`doacross(sink:`** `j-1, i`**`)`** and **`doacross(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`).

C / C++

Example doacross.2.c (**`omp_5.2`**)

```
S-1   float foo(int i, int j);
S-2   float bar(float a, float b, float c);
S-3   float baz(float b);
S-4
S-5   void work( int N, int M, float **A, float **B, float **C )
S-6   {
S-7     int i, j;
S-8
S-9     #pragma omp for ordered(2)
S-10    for (i=1; i<N; i++)
S-11    {
S-12      for (j=1; j<M; j++)
S-13      {
S-14        A[i][j] = foo(i, j);
S-15
S-16    #pragma omp ordered doacross(sink: i-1,j) doacross(sink: i,j-1)
S-17        B[i][j] = bar(A[i][j], B[i-1][j], B[i][j-1]);
S-18    #pragma omp ordered doacross(source:)
S-19
S-20        C[i][j] = baz(B[i][j]);
S-21      }
S-22    }
S-23  }
```

C / C++

Fortran

Example doacross.2.f90 (**omp_5.2**)

```
S-1   subroutine work( N, M, A, B, C )
S-2     integer :: N, M, i, j
S-3     real, dimension(M,N) :: A, B, C
S-4     real, external :: foo, bar, baz
S-5
S-6     !$omp do ordered(2)
S-7     do j=2, N
S-8       do i=2, M
S-9         A(i,j) = foo(i, j)
S-10
S-11      !$omp ordered doacross(sink: j-1,i) doacross(sink: j,i-1)
S-12        B(i,j) = bar(A(i,j), B(i-1,j), B(i,j-1))
S-13      !$omp ordered doacross(source:)
S-14
S-15        C(i,j) = baz(B(i,j))
S-16      end do
S-17    end do
S-18  end subroutine
```

Fortran

The following example shows an incorrect use of the **ordered** directive with a **doacross** clause. There are two issues with the code. The first issue is a missing **ordered doacross(source:)** directive, which could cause a deadlock. The second issue is the **doacross(sink:** *i+1, j*) and **doacross(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.

C / C++

Example doacross.3.c (**omp_5.2**)

```
S-1   #define N 100
S-2
S-3   void work_wrong(double p[][N][N])
S-4   {
S-5     int i, j, k;
S-6
S-7     #pragma omp parallel for ordered(2) private(i,j,k)
S-8     for (i=1; i<N-1; i++)
S-9     {
S-10      for (j=1; j<N-1; j++)
S-11      {
S-12    #pragma omp ordered doacross(sink: i-1,j) doacross(sink: i+1,j) \
S-13                        doacross(sink: i,j-1) doacross(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;
        }
/* missing #pragma omp ordered doacross(source:) */
      }
    }
}
```

C / C++

Fortran

Example doacross.3.f90 (`omp_5.2`)

```
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 doacross(sink: i-1,j) doacross(sink: i+1,j) &
    !$omp&        doacross(sink: i,j-1) doacross(sink: i,j+1)
      do k=2, N-1
        tmp1 = p(k-1,j,i) + p(k+1,j,i)
        tmp2 = p(k,j-1,i) + p(k,j+1,i)
        tmp3 = p(k,j,i-1) + p(k,j,i+1)
        p(k,j,i) = (tmp1 + tmp2 + tmp3) / 6.0
      end do
! missing !$omp ordered doacross(source:)
    end do
  end do
end subroutine
```

Fortran

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 conforming usage of the **ordered** directive specifying a cross-iteration source that is placed before a corresponding **ordered** directive specifying a cross-iteration sink. There is no requirement that the source specification must follow the sink specification in a given iteration.

C / C++

Example doacross.4.c (omp_5.2)

```
S-1   double foo(int i, int j);
S-2
S-3   void work( int N, int M, double **A, double **B, double **C )
S-4   {
S-5     int i, j;
S-6     double alpha = 1.2;
S-7
S-8     #pragma omp for collapse(2) ordered(2)
S-9     for (i = 1; i < N-1; i++)
S-10    {
S-11      for (j = 1; j < M-1; j++)
S-12      {
S-13        A[i][j] = foo(i, j);
S-14    #pragma omp ordered doacross(source:)
S-15
S-16        B[i][j] = alpha * A[i][j];
S-17
S-18    #pragma omp ordered doacross(sink: i-1,j) doacross(sink: i,j-1)
S-19        C[i][j] = 0.2 * (A[i-1][j] + A[i+1][j] +
S-20                         A[i][j-1] + A[i][j+1] + A[i][j]);
S-21      }
S-22    }
S-23  }
```

C / C++

Fortran

Example doacross.4.f90 (omp_5.2)

```
S-1   subroutine work( N, M, A, B, C )
S-2     integer :: N, M
S-3     real(8), dimension(M, N) :: A, B, C
S-4     real(8), external :: foo
S-5     integer :: i, j
S-6     real(8) :: alpha = 1.2
S-7
S-8     !$omp do collapse(2) ordered(2)
S-9     do j=2, N-1
S-10      do i=2, M-1
S-11        A(i,j) = foo(i, j)
S-12      !$omp ordered doacross(source:)
S-13
S-14        B(i,j) = alpha * A(i,j)
S-15
S-16      !$omp ordered doacross(sink: j,i-1) doacross(sink: j-1,i)
```

```
S-17          C(i,j) = 0.2 * (A(i-1,j) + A(i+1,j) +  &
S-18                    A(i,j-1) + A(i,j+1) + A(i,j))
S-19        end do
S-20      end do
S-21    end subroutine
```

Fortran

9.12 Lock Routines

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

9.12.1 `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++

Example init_lock.1.cpp (**pre_omp_3.0**)

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

C++

Fortran

Example init_lock.1.f (**pre_omp_3.0**)

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

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

      END FUNCTION NEW_LOCKS
```

Fortran

9.12.2 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.

C++

Example init_lock_with_hint.1.cpp (**omp_5.0**)

```
#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],
        static_cast<omp_lock_hint_t>(omp_sync_hint_contended |
                                     omp_sync_hint_speculative));
    }
    return lock;
}
```

C++

Fortran

Example init_lock_with_hint.1.f (**omp_5.0**)

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

        INTEGER I

!$OMP   PARALLEL DO PRIVATE(I)
          DO I=1,1000
            CALL OMP_INIT_LOCK_WITH_HINT(NEW_LOCKS(I),
     &             OMP_SYNC_HINT_CONTENDED + OMP_SYNC_HINT_SPECULATIVE)
          END DO
!$OMP   END PARALLEL DO

      END FUNCTION NEW_LOCKS
```

Fortran

9.12.3 Ownership of Locks

Ownership of locks has changed since OpenMP 2.5. In OpenMP 2.5, locks are owned by threads; so a lock released by the **omp_unset_lock** routine must be owned by the same thread executing the routine. Beginning with OpenMP 3.0, locks are owned by tasks; so a lock released by the **omp_unset_lock** routine in a task must be owned by the same task.

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 (primary thread of **parallel** region and the initial thread are the same). However, it is not conforming beginning with OpenMP 3.0, because the task region that releases the lock *lck* is different from the task region that acquires the lock.

C / C++

Example lock_owner.1.c (omp_5.1)

```
S-1   #include <stdlib.h>
S-2   #include <stdio.h>
S-3   #include <omp.h>
S-4
S-5   int main()
S-6   {
S-7     int x;
S-8     omp_lock_t lck;
S-9
S-10    omp_init_lock (&lck);
S-11    omp_set_lock (&lck);
S-12    x = 0;
S-13
S-14  #pragma omp parallel shared (x)
S-15    {
S-16      #pragma omp masked
S-17        {
S-18          x = x + 1;
S-19          omp_unset_lock (&lck);
S-20        }
S-21
S-22      /* Some more stuff. */
S-23    }
S-24    omp_destroy_lock (&lck);
S-25    return 0;
S-26  }
```

C / C++

Fortran

Example lock_owner.1.f (omp_5.1)

```
S-1           program lock
S-2           use omp_lib
S-3           integer :: x
S-4           integer (kind=omp_lock_kind) :: lck
S-5
S-6           call omp_init_lock (lck)
S-7           call omp_set_lock(lck)
S-8           x = 0
S-9
S-10  !$omp parallel shared (x)
S-11  !$omp masked
S-12          x = x + 1
S-13          call omp_unset_lock(lck)
```

```
!$omp end masked

!       Some more stuff.
!$omp end parallel

        call omp_destroy_lock(lck)

        end
```

Fortran

9.12.4 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** (or **omp_lock_kind** in Fortran), and that there is no need to flush the lock variable (*lck*).

C / C++

Example simple_lock.1.c (**pre_omp_3.0**)

```
#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 */
```

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

C / C++

Fortran

Example simple_lock.1.f (`pre_omp_3.0`)

```
S-1         SUBROUTINE SKIP(ID)
S-2         END SUBROUTINE SKIP
S-3
S-4         SUBROUTINE WORK(ID)
S-5         END SUBROUTINE WORK
S-6
S-7         PROGRAM SIMPLELOCK
S-8
S-9           INCLUDE "omp_lib.h"     ! or USE OMP_LIB
S-10
S-11          INTEGER(OMP_LOCK_KIND) LCK
S-12          INTEGER ID
S-13
S-14          CALL OMP_INIT_LOCK(LCK)
S-15
S-16    !$OMP   PARALLEL SHARED(LCK) PRIVATE(ID)
S-17            ID = OMP_GET_THREAD_NUM()
S-18            CALL OMP_SET_LOCK(LCK)
S-19            PRINT *, 'My thread id is ', ID
S-20            CALL OMP_UNSET_LOCK(LCK)
S-21
S-22            DO WHILE (.NOT. OMP_TEST_LOCK(LCK))
S-23              CALL SKIP(ID)      ! We do not yet have the lock
S-24                                 ! so we must do something else
S-25            END DO
S-26
S-27            CALL WORK(ID)        ! We now have the lock
S-28                                 ! and can do the work
S-29
S-30            CALL OMP_UNSET_LOCK( LCK )
S-31
```

```
!$OMP   END PARALLEL

        CALL OMP_DESTROY_LOCK( LCK )

      END PROGRAM SIMPLELOCK
```

Fortran

9.12.5 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 / C++

Example nestable_lock.1.c (**pre_omp_3.0**)

```
#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);

}
```

```
S-32    void incr_pair(pair *p, int a, int b)
S-33    {
S-34
S-35      omp_set_nest_lock(&p->lck);
S-36      incr_a(p, a);
S-37      incr_b(p, b);
S-38      omp_unset_nest_lock(&p->lck);
S-39
S-40    }
S-41
S-42    void nestlock(pair *p)
S-43    {
S-44
S-45      #pragma omp parallel sections
S-46      {
S-47        #pragma omp section
S-48          incr_pair(p, work1(), work2());
S-49        #pragma omp section
S-50          incr_b(p, work3());
S-51      }
S-52
S-53    }
```

C / C++

Fortran

1 *Example nestable_lock.1.f* (**pre_omp_3.0**)

```
S-1           MODULE DATA
S-2             USE OMP_LIB, ONLY: OMP_NEST_LOCK_KIND
S-3             TYPE LOCKED_PAIR
S-4               INTEGER A
S-5               INTEGER B
S-6               INTEGER (OMP_NEST_LOCK_KIND) LCK
S-7             END TYPE
S-8           END MODULE DATA
S-9
S-10          SUBROUTINE INCR_A(P, A)
S-11            ! called only from INCR_PAIR, no need to lock
S-12            USE DATA
S-13            TYPE(LOCKED_PAIR) :: P
S-14            INTEGER A
S-15            P%A = P%A + A
S-16          END SUBROUTINE INCR_A
S-17
S-18          SUBROUTINE INCR_B(P, B)
S-19            ! called from both INCR_PAIR and elsewhere,
S-20            ! so we need a nestable lock
```

```
S-21            USE OMP_LIB       ! or INCLUDE "omp_lib.h"
S-22            USE DATA
S-23            TYPE(LOCKED_PAIR) :: P
S-24            INTEGER B
S-25            CALL OMP_SET_NEST_LOCK(P%LCK)
S-26            P%B = P%B + B
S-27            CALL OMP_UNSET_NEST_LOCK(P%LCK)
S-28          END SUBROUTINE INCR_B
S-29
S-30          SUBROUTINE INCR_PAIR(P, A, B)
S-31            USE OMP_LIB        ! or INCLUDE "omp_lib.h"
S-32            USE DATA
S-33            TYPE(LOCKED_PAIR) :: P
S-34            INTEGER A
S-35            INTEGER B
S-36
S-37            CALL OMP_SET_NEST_LOCK(P%LCK)
S-38            CALL INCR_A(P, A)
S-39            CALL INCR_B(P, B)
S-40            CALL OMP_UNSET_NEST_LOCK(P%LCK)
S-41          END SUBROUTINE INCR_PAIR
S-42
S-43          SUBROUTINE NESTLOCK(P)
S-44            USE OMP_LIB        ! or INCLUDE "omp_lib.h"
S-45            USE DATA
S-46            TYPE(LOCKED_PAIR) :: P
S-47            INTEGER WORK1, WORK2, WORK3
S-48            EXTERNAL WORK1, WORK2, WORK3
S-49
S-50      !$OMP   PARALLEL SECTIONS
S-51
S-52      !$OMP   SECTION
S-53              CALL INCR_PAIR(P, WORK1(), WORK2())
S-54      !$OMP   SECTION
S-55              CALL INCR_B(P, WORK3())
S-56      !$OMP   END PARALLEL SECTIONS
S-57
S-58          END SUBROUTINE NESTLOCK
```

Fortran

10 Data Environment

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 cannot be listed in a data-sharing 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 (increment by 1 on entry and decrement by 1 on exit) 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.

10.1 `threadprivate` Directive

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

C / C++

Example threadprivate.1.c (**pre_omp_3.0**)

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

C / C++

Fortran

Example threadprivate.1.f (**pre_omp_3.0**)

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

Fortran

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

Example threadprivate.2.c (**pre_omp_3.0**)

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

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

Example threadprivate.3.cpp (**pre_omp_3.0**)

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

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

C / C++

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

Fortran

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

Example threadprivate.2.f (**pre_omp_3.0**)

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

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

Example threadprivate.3.f (**pre_omp_3.0**)

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

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

Example threadprivate.4.f (**pre_omp_3.0**)

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

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

Example threadprivate.5.f (**pre_omp_3.0**)

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

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

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

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

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

8 or

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

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

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

16 *Example threadprivate.6.f* (**pre_omp_3.0**)

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

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

Fortran

C++

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

Example threadprivate.4.cpp (**pre_omp_3.0**)

```
S-1   struct T { T (); T (int); ~T (); int t; };
S-2   int f();
S-3   static T t1;
S-4   #pragma omp threadprivate(t1)
S-5   static T t2( 23 );
S-6   #pragma omp threadprivate(t2)
S-7   static T t3 = f();
S-8   #pragma omp threadprivate(t3)
```

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

Example threadprivate.5.cpp (**pre_omp_3.0**)

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

C++

10.2 `default(none)` Clause

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

C / C++

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

Example default_none.1.c (**`pre_omp_3.0`**)

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

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

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

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

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

C / C++

Fortran

Example default_none.1.f (**pre_omp_3.0**)

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

Fortran

10.3 `private` Clause

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

C / C++

Example private.1.c (**pre_omp_3.0**)

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

C / C++

Fortran

Example private.1.f (**pre_omp_3.0**)

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

Fortran

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

C / C++

Example private.2.c (**pre_omp_3.0**)

```
int a;

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


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

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

C / C++

Fortran

Example private.2.f (**pre_omp_3.0**)

```
      MODULE PRIV_EXAMPLE2
        REAL A

        CONTAINS

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

          SUBROUTINE F(N)
          INTEGER N
          REAL A

            INTEGER I
```

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

      END MODULE PRIV_EXAMPLE2
```

Fortran

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

C / C++

Example private.3.c (**pre_omp_3.0**)

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

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

C / C++

Fortran

Example private.3.f (**pre_omp_3.0**)

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

Fortran

Fortran

10.4 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 the *Loop Construct* section and the *Data-sharing Attribute Rules* section 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 (**pre_omp_3.0**)

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

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

Example fort_loopvar.2.f90 (**pre_omp_3.0**)

```
S-1     SUBROUTINE PLOOP_2(A,B,N,I1,I2)
S-2     REAL A(*), B(*)
S-3     INTEGER I1, I2, N
S-4
S-5     !$OMP PARALLEL SHARED(A,B,I1,I2)
S-6     !$OMP SECTIONS
S-7     !$OMP SECTION
S-8          DO I1 = I1, N
S-9            IF (A(I1).NE.0.0) EXIT
S-10         ENDDO
S-11    !$OMP SECTION
S-12         DO I2 = I2, N
S-13           IF (B(I2).NE.0.0) EXIT
S-14         ENDDO
S-15    !$OMP END SECTIONS
S-16    !$OMP SINGLE
S-17        IF (I1.LE.N) PRINT *, 'ITEMS IN A UP TO ', I1, 'ARE ALL ZERO.'
S-18        IF (I2.LE.N) PRINT *, 'ITEMS IN B UP TO ', I2, 'ARE ALL ZERO.'
S-19    !$OMP END SINGLE
S-20    !$OMP END PARALLEL
S-21
S-22    END SUBROUTINE PLOOP_2
```

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

Fortran

Fortran

10.5 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 (**pre_omp_3.0**)

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

The following example is also conforming:

Example fort_sp_common.2.f (**pre_omp_3.0**)

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

The following example is conforming:

Example fort_sp_common.3.f (**pre_omp_3.0**)

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

```
!$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 (**pre_omp_3.0**)

```
      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 (**pre_omp_3.0**)

```
      SUBROUTINE COMMON_WRONG2()
        COMMON /C/ X,Y
! Incorrect: common block C cannot be declared both
! shared and private
!$OMP   PARALLEL PRIVATE (/C/), SHARED(/C/)
          ! do work here
!$OMP   END PARALLEL

      END SUBROUTINE COMMON_WRONG2
```

Fortran

Fortran

10.6 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.

1 *Example fort_sa_private.1.f* (**pre_omp_3.0**)

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

2 *Example fort_sa_private.2.f* (**pre_omp_3.0**)

```
S-1          PROGRAM PRIV_RESTRICT2
S-2            COMMON /BLOCK2/ X
S-3            X = 1.0
S-4
S-5     !$OMP  PARALLEL PRIVATE (X)
S-6              X = 2.0
S-7              CALL SUB()
S-8     !$OMP  END PARALLEL
S-9
S-10           CONTAINS
S-11
S-12             SUBROUTINE SUB()
S-13             COMMON /BLOCK2/ Y
S-14
S-15             PRINT *,X              ! X is undefined
S-16             PRINT *,Y              ! Y is undefined
S-17             END SUBROUTINE SUB
S-18
S-19           END PROGRAM PRIV_RESTRICT2
```

3 *Example fort_sa_private.3.f* (**pre_omp_3.0**)

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

```
S-7             Y = 10
S-8             PRINT *,X                    ! X is undefined
S-9     !$OMP   END PARALLEL
S-10          END PROGRAM PRIV_RESTRICT3
```

Example fort_sa_private.4.f (`pre_omp_3.0`)

```
S-1           PROGRAM PRIV_RESTRICT4
S-2             INTEGER I, J
S-3             INTEGER A(100), B(100)
S-4             EQUIVALENCE (A(51), B(1))
S-5
S-6     !$OMP PARALLEL DO DEFAULT(PRIVATE) PRIVATE(I,J) LASTPRIVATE(A)
S-7               DO I=1,100
S-8                  DO J=1,100
S-9                    B(J) = J - 1
S-10                 ENDDO
S-11
S-12                 DO J=1,100
S-13                   A(J) = J   ! B becomes undefined at this point
S-14                 ENDDO
S-15
S-16                 DO J=1,50
S-17                   B(J) = B(J) + 1  ! B is undefined
S-18                                    ! A becomes undefined at this point
S-19                 ENDDO
S-20              ENDDO
S-21    !$OMP END PARALLEL DO         ! The LASTPRIVATE write for A has
S-22                                  ! undefined results
S-23
S-24             PRINT *, B    ! B is undefined since the LASTPRIVATE
S-25                           ! write of A was not defined
S-26           END PROGRAM PRIV_RESTRICT4
```

Example fort_sa_private.5.f (`omp_5.1`)

```
S-1           SUBROUTINE SUB1(X)
S-2             DIMENSION X(*)
S-3
S-4             ! This use of X does not conform to the
S-5             ! specification. It would be legal Fortran 90,
S-6             ! but the OpenMP private directive allows the
S-7             ! compiler to break the sequence association that
S-8             ! A had with the rest of the common block.
S-9
S-10            FORALL (I = 1:10) X(I) = I
```

```
S-11          END SUBROUTINE SUB1
S-12
S-13          PROGRAM PRIV_RESTRICT5
S-14            COMMON /BLOCK5/ A
S-15
S-16            DIMENSION A(1),B(10)
S-17            EQUIVALENCE (A,B(1))
S-18
S-19            ! the common block has to be at least 10 words
S-20            A = 0
S-21
S-22    !$OMP   PARALLEL PRIVATE(/BLOCK5/)
S-23
S-24              ! Without the private clause,
S-25              ! we would be passing a member of a sequence
S-26              ! that is at least ten elements long.
S-27              ! With the private clause, A may no longer be
S-28              ! sequence-associated.
S-29
S-30              CALL SUB1(A)
S-31    !$OMP     MASKED
S-32                PRINT *, A
S-33    !$OMP     END MASKED
S-34
S-35    !$OMP   END PARALLEL
S-36          END PROGRAM PRIV_RESTRICT5
```

Fortran

Fortran

10.7 Passing Shared Variable to Procedure in Fortran

Passing a shared variable to a procedure in Fortran may result in the use of temporary storage in place of the actual argument when the corresponding dummy argument does not have the **VALUE** or **CONTIGUOUS** attribute and its data-sharing attribute is implementation-defined as per the rules in Section *Variables Referenced in a Region but not in a Construct* of the OpenMP Specification. These conditions effectively result in references to, and definitions of, the temporary storage during the procedure reference. Furthermore, the value of the shared variable is copied into the intervening temporary storage before the procedure reference when the dummy argument does not have the **INTENT(OUT)** attribute, and is copied out of the temporary storage into the shared variable when the dummy argument does not have the **INTENT(IN)** attribute. Any references to (or definitions of) the shared storage that is associated with the dummy argument by any other task must be synchronized with the procedure reference to avoid possible data races.

The following examples illustrate the implications of passing a shared variable *a* to subroutine *sub1* or *sub2* in a **parallel** region. For *sub1*, an implementation may or may not generate a copy-in/copy-out for the temporary storage associated with variable *b*. If there is a copy-in/copy-out, the code for copy-in/copy-out will result in a race condition, even though there is an **atomic** directive for the update of variable *b(i)* in the subroutine. If the implementation can create a temporary descriptor for *a(::2)* with the correct stride and passed it to subroutine *sub1*, the same memory is accessed inside the subroutine and the result (*sum1*) is then well defined. For *sub2*, there is the **CONTIGUOUS** attribute for variable *b* and the implementation will generate a copy-in/copy-out for the temporary storage. The code will have a race condition and the result (*sum2*) is not well defined.

Example fort_shared_var.1.f90 (**pre_omp_3.0**)

```
S-1   program fort_shared_var
S-2     implicit none
S-3     integer, parameter :: N = 100
S-4     integer a(N)
S-5     integer i
S-6     interface
S-7       subroutine sub1(b)
S-8         integer b(:)
S-9       end subroutine
S-10      subroutine sub2(b)
S-11        integer, contiguous :: b(:)
S-12      end subroutine
S-13    end interface
S-14
S-15    a = [(i, i=1,N)]
S-16    !$omp parallel shared(a) num_threads(2)
S-17      call sub1(a(::2))         ! copy-in/copy-out may or may not occur
S-18    !$omp end parallel
S-19    print *, 'sum1 =', sum(a)   ! sum1 may/may not be well defined
S-20
S-21    a = [(i, i=1,N)]
S-22    !$omp parallel shared(a) num_threads(2)
S-23      call sub2(a(::2))         ! copy-in/copy-out result in a data race
S-24    !$omp end parallel
S-25    print *, 'sum2 =', sum(a)   ! sum2 is not well defined
S-26  end
S-27
S-28  subroutine sub1(b)
S-29    implicit none
S-30    integer b(:)
S-31    integer i
```

```
S-32      do i = 1, size(b)
S-33        !$omp atomic
S-34        b(i) = b(i) + 1
S-35      end do
S-36    end subroutine
S-37
S-38    subroutine sub2(b)
S-39      implicit none
S-40      integer, contiguous :: b(:)
S-41      integer i
S-42      do i = 1, size(b)
S-43        !$omp atomic
S-44        b(i) = b(i) + 1
S-45      end do
S-46    end subroutine
```

Fortran

C / C++

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

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

In this example:

- The type of *A* is array of two arrays of two **`int`**s.
- The type of *B* is adjusted to pointer to array of *n* **`int`**s, 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 **`int`**s.
- The type of *E* is array of *n* arrays of *n* **`int`**s.

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 (**`pre_omp_3.0`**)

```
S-1   #include <assert.h>
S-2
S-3   int A[2][2] = {1, 2, 3, 4};
S-4
S-5   void f(int n, int B[n][n], int C[])
S-6   {
S-7     int D[2][2] = {1, 2, 3, 4};
S-8     int E[n][n];
S-9
S-10    assert(n >= 2);
S-11    E[1][1] = 4;
S-12
S-13    #pragma omp parallel firstprivate(B, C, D, E)
S-14    {
S-15      assert(sizeof(B) == sizeof(int (*)[n]));
S-16      assert(sizeof(C) == sizeof(int*));
S-17      assert(sizeof(D) == 4 * sizeof(int));
S-18      assert(sizeof(E) == n * n * sizeof(int));
S-19
S-20      /* Private B and C have values of original B and C. */
S-21      assert(&B[1][1] == &A[1][1]);
S-22      assert(&C[3] == &A[1][1]);
S-23      assert(D[1][1] == 4);
S-24      assert(E[1][1] == 4);
S-25    }
S-26  }
S-27
S-28  int main() {
S-29    f(2, A, A[0]);
S-30    return 0;
S-31  }
```

C / C++

10.9 `lastprivate` Clause

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

C / C++

Example lastprivate.1.c (**pre_omp_3.0**)

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

C / C++

Fortran

Example lastprivate.1.f (**pre_omp_3.0**)

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

Fortran

The next example illustrates the use of the **conditional** modifier in a **lastprivate** clause to return the last value when it may not come from the last iteration of a loop. That is, users can preserve the serial equivalence semantics of the loop. The conditional lastprivate ensures the final value of the variable after the loop is as if the loop iterations were executed in a sequential order.

C / C++

Example lastprivate.2.c (`omp_5.0`)

```
S-1   #include <math.h>
S-2
S-3   float condlastprivate(float *a, int n)
S-4   {
S-5      float x = 0.0f;
S-6
S-7      #pragma omp parallel for simd lastprivate(conditional: x)
S-8      for (int k = 0; k < n; k++) {
S-9         if (a[k] < 108.5 || a[k] > 208.5) {
S-10           x = sinf(a[k]);
S-11        }
S-12     }
S-13
S-14     return x;
S-15  }
```

C / C++

Fortran

Example lastprivate.2.f90 (`omp_5.0`)

```
S-1   function condlastprivate(a, n) result(x)
S-2      implicit none
S-3      real a(*), x
S-4      integer n, k
S-5
S-6      x = 0.0
S-7
S-8      !$omp parallel do simd lastprivate(conditional: x)
S-9      do k = 1, n
S-10        if (a(k) < 108.5 .or. a(k) > 208.5) then
S-11           x = sin(a(k))
S-12        endif
S-13     end do
S-14
S-15  end function condlastprivate
```

Fortran

10.10 Reduction

This section covers ways to perform reductions in parallel, task, taskloop, and SIMD regions.

10.10.1 `reduction` Clause

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

C / C++

Example reduction.1.c (`omp_3.1`)

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

C / C++

Fortran

Example reduction.1.f90 (`pre_omp_3.0`)

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

Fortran

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

C / C++

Example reduction.2.c (`pre_omp_3.0`)

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

```
      for (i=0; i<n; i++) {
        a_p += x[i];
        b_p ^= y[i];
        if (c_p > y[i]) c_p = y[i];
        d_p = fmaxf(d_p,x[i]);
      }
      #pragma omp critical
      {
        a += a_p;
        b ^= b_p;
        if( c > c_p ) c = c_p;
        d = fmaxf(d,d_p);
      }
    }
}
```

C / C++

Fortran

Example reduction.2.f90 (`pre_omp_3.0`)

```
  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 (**pre_omp_3.0**)

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

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

 END PROGRAM REDUCTION_WRONG

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

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

Example reduction.4.f90 (**pre_omp_3.0**)

```
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 (**pre_omp_3.0**)

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

Fortran

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

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

C / C++

Example reduction.6.c (**omp_5.1**)

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

```
      #pragma omp single
      printf ("Sum is %d\n", a);
    }
    return 0;
}
```

C / C++

Fortran

Example reduction.6.f (**omp_5.1**)

```
      INTEGER A, I

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

!$OMP MASKED
      A = 0
!$OMP END MASKED

      ! To avoid race conditions, add a barrier here.

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

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

!$OMP END PARALLEL

      END
```

Fortran

The 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 has been fixed in the OpenMP 5.0 specification.

C / C++

1 *Example reduction.7.c* (`omp_4.5`)

```
S-1   #include <stdio.h>
S-2
S-3   #define N 100
S-4   void init(int n, float (*b)[N]);
S-5
S-6   int main(){
S-7
S-8     int i,j;
S-9     float a[N], b[N][N];
S-10
S-11    init(N,b);
S-12
S-13    for(i=0; i<N; i++) a[i]=0.0e0;
S-14
S-15    #pragma omp parallel for reduction(+:a[0:N]) private(j)
S-16    for(i=0; i<N; i++){
S-17      for(j=0; j<N; j++){
S-18         a[j] +=  b[i][j];
S-19      }
S-20    }
S-21    printf(" a[0] a[N-1]: %f %f\n", a[0], a[N-1]);
S-22
S-23    return 0;
S-24  }
```

C / C++

Fortran

2 *Example reduction.7.f90* (`pre_omp_3.0`)

```
S-1   program array_red
S-2
S-3     integer,parameter :: n=100
S-4     integer           :: j
S-5     real              :: a(n), b(n,n)
S-6
S-7     call init(n,b)
S-8
S-9     a(:) = 0.0e0
S-10
S-11    !$omp parallel do reduction(+:a)
S-12    do j = 1, n
S-13       a(:) = a(:) + b(:,j)
S-14    end do
S-15
```

```
  print*, " a(1) a(n): ", a(1), a(n)

end program
```

Fortran

10.10.2 Task Reduction

In OpenMP 5.0 the **task_reduction** clause was created for the **taskgroup** construct, to allow reductions among explicit tasks that have an **in_reduction** clause.

In the *task_reduction.1* example below a reduction is performed as the algorithm traverses a linked list. The reduction statement is assigned to be an explicit task using a **task** construct and is specified to be a reduction participant with the **in_reduction** clause. A **taskgroup** construct encloses the tasks participating in the reduction, and specifies, with the **task_reduction** clause, that the taskgroup has tasks participating in a reduction. After the **taskgroup** region the original variable will contain the final value of the reduction.

Note: The *res* variable is private in the *linked_list_sum* routine and is not required to be shared (as in the case of a **parallel** construct reduction).

C / C++

Example task_reduction.1.c (**omp_5.0**)

```
#include<stdlib.h>
#include<stdio.h>
#define N 10

typedef struct node_tag {
    int val;
    struct node_tag *next;
} node_t;

int linked_list_sum(node_t *p)
{
    int res = 0;

    #pragma omp taskgroup task_reduction(+: res)
    {
        node_t* aux = p;
        while(aux != 0)
        {
            #pragma omp task in_reduction(+: res)
            res += aux->val;

            aux = aux->next;
```

```
S-23            }
S-24        }
S-25        return res;
S-26    }
S-27
S-28
S-29    int main() {
S-30        int i;
S-31    //                              Create the root node.
S-32        node_t* root = (node_t*) malloc(sizeof(node_t));
S-33        root->val = 1;
S-34
S-35        node_t* aux = root;
S-36
S-37    //                              Create N-1 more nodes.
S-38        for(i=2;i<=N;++i){
S-39            aux->next = (node_t*) malloc(sizeof(node_t));
S-40            aux = aux->next;
S-41            aux->val = i;
S-42        }
S-43
S-44        aux->next = 0;
S-45
S-46        #pragma omp parallel
S-47        #pragma omp single
S-48        {
S-49            int result = linked_list_sum(root);
S-50            printf( "Calculated: %d  Analytic:%d\n", result, (N*(N+1)/2) );
S-51        }
S-52
S-53        return 0;
S-54    }
```

C / C++

Fortran

1 *Example task_reduction.1.f90* (`omp_5.0`)

```
S-1     module m
S-2         type node_t
S-3             integer :: val
S-4             type(node_t), pointer :: next
S-5         end type
S-6     end module m
S-7
S-8     function linked_list_sum(p) result(res)
S-9         use m
S-10        implicit none
```

```
S-11        type(node_t), pointer :: p
S-12        type(node_t), pointer :: aux
S-13        integer :: res
S-14
S-15        res = 0
S-16
S-17        !$omp taskgroup task_reduction(+: res)
S-18            aux => p
S-19            do while (associated(aux))
S-20                !$omp task in_reduction(+: res)
S-21                    res = res + aux%val
S-22                !$omp end task
S-23                aux => aux%next
S-24            end do
S-25        !$omp end taskgroup
S-26    end function linked_list_sum
S-27
S-28
S-29    program main
S-30        use m
S-31        implicit none
S-32        type(node_t), pointer :: root, aux
S-33        integer :: res, i
S-34        integer, parameter :: N=10
S-35
S-36        interface
S-37            function linked_list_sum(p) result(res)
S-38                use m
S-39                implicit none
S-40                type(node_t), pointer :: p
S-41                integer :: res
S-42            end function
S-43        end interface
S-44    !                       Create the root node.
S-45        allocate(root)
S-46        root%val = 1
S-47        aux => root
S-48
S-49    !                       Create N-1 more nodes.
S-50        do i = 2,N
S-51            allocate(aux%next)
S-52            aux => aux%next
S-53            aux%val = i
S-54        end do
S-55
S-56        aux%next => null()
S-57
```

```
      !$omp parallel
      !$omp single
          res = linked_list_sum(root)
          print *, "Calculated:", res, " Analytic:", (N*(N+1))/2
      !$omp end single
      !$omp end parallel

end program main
```

Fortran

In OpenMP 5.0 the **task** *reduction-modifier* for the **reduction** clause was introduced to provide a means of performing reductions among implicit and explicit tasks.

The **reduction** clause of a **parallel** or worksharing construct may specify the **task** *reduction-modifier* to include explicit task reductions within their region, provided the reduction operators (*reduction-identifiers*) and variables (list items) of the participating tasks match those of the implicit tasks.

There are 2 reduction use cases (identified by USE CASE #) in the *task_reduction.2* example below.

In USE CASE 1 a **task** modifier in the **reduction** clause of the **parallel** construct is used to include the reductions of any participating tasks, those with an **in_reduction** clause and matching *reduction-identifiers* (**+**) and list items (*x*).

Note, a **taskgroup** construct (with a **task_reduction** clause) is not necessary to scope the explicit task reduction (as seen in the example above). Hence, even without the implicit task reduction statement (without the C *x++;* and Fortran *x=x+1* statements), the **task** *reduction-modifier* in a **reduction** clause of the **parallel** construct can be used to avoid having to create a **taskgroup** construct (and its **task_reduction** clause) around the task generating structure.

In USE CASE 2 tasks participating in the reduction are within a worksharing region (a parallel worksharing-loop construct). Here, too, no **taskgroup** is required, and the *reduction-identifier* (**+**) and list item (variable *x*) match as required.

C / C++

Example task_reduction.2.c (**omp_5.0**)

```
#include <stdio.h>
int main(void){
   int N=100, M=10;
   int i, x;

// USE CASE 1  explicit-task reduction + parallel reduction clause
   x=0;
   #pragma omp parallel num_threads(M) reduction(task,+:x)
   {

```

```
      x++;                 // implicit task reduction statement

      #pragma omp single
      for(i=0;i<N;i++)
        #pragma omp task in_reduction(+:x)
        x++;

   }
   printf("x=%d  =M+N\n",x);  // x= 110  =M+N


// USE CASE 2  task reduction +  worksharing reduction clause
   x=0;
   #pragma omp parallel for num_threads(M) reduction(task,+:x)
   for(i=0; i< N; i++){

      x++;

      if( i%2 == 0){
       #pragma omp task in_reduction(+:x)
       x--;
      }
   }
   printf("x=%d  =N-N/2\n",x);  // x= 50  =N-N/2

   return 0;
}
```

C / C++

Fortran

Example task_reduction.2.f90 (`omp_5.0`)

```
program task_modifier

   integer :: N=100, M=10
   integer :: i, x

! USE CASE 1  explicit-task reduction + parallel reduction clause
   x=0
   !$omp parallel num_threads(M) reduction(task,+:x)

     x=x+1                   !! implicit task reduction statement

     !$omp single
       do i = 1,N
         !$omp task in_reduction(+:x)
           x=x+1
```

```
S-16              !$omp end task
S-17            end do
S-18          !$omp end single
S-19
S-20       !$omp end parallel
S-21       write(*,'("x=",I0," =M+N")') x    ! x= 110 =M+N
S-22
S-23
S-24    ! USE CASE 2  task reduction +  worksharing reduction clause
S-25       x=0
S-26       !$omp parallel do num_threads(M) reduction(task,+:x)
S-27         do i = 1,N
S-28
S-29            x=x+1
S-30
S-31            if( mod(i,2) == 0) then
S-32               !$omp task in_reduction(+:x)
S-33                 x=x-1
S-34               !$omp end task
S-35            endif
S-36
S-37         end do
S-38       write(*,'("x=",I0,"  =N-N/2")') x    ! x= 50 =N-N/2
S-39
S-40    end program
```

Fortran

10.10.3 Reduction on Combined Target Constructs

When a **reduction** clause appears on a combined construct that combines a **target** construct with another construct, there is an implicit map of the list items with a **tofrom** map type for the **target** construct. Otherwise, the list items (if they are scalar variables) would be treated as firstprivate by default in the **target** construct, which is unlikely to provide the intended behavior since the result of the reduction that is in the firstprivate variable would be discarded at the end of the **target** region.

In the following example, the use of the **reduction** clause on *sum1* or *sum2* should, by default, result in an implicit **tofrom** map for that variable. So long as neither *sum1* nor *sum2* were already present on the device, the mapping behavior ensures the value for *sum1* computed in the first **target** construct is used in the second **target** construct.

Note: a **declare target** directive is needed for procedures, *f* and *g*, called in **target** region in Fortran codes. This directive is not required in C codes because functions, *f* and *g*, are defined in the same compilation unit of the **target** construct in which these functions are called.

C / C++

Example target_reduction.1.c (`omp_5.0`)

```
#include <stdio.h>
int f(int);
int g(int);
int main()
{
   int sum1=0, sum2=0;
   const int n = 100;

   #pragma omp target teams distribute reduction(+:sum1)
   for (int i = 0; i < n; i++) {
      sum1 += f(i);
   }

   #pragma omp target teams distribute reduction(+:sum2)
   for (int i = 0; i < n; i++) {
      sum2 += g(i) * sum1;
   }

   printf(  "sum1 = %d, sum2 = %d\n", sum1, sum2);
   //OUTPUT: sum1 = 9900, sum2 = 147015000
   return 0;
}

int f(int res){ return res*2; }
int g(int res){ return res*3; }
```

C / C++

Fortran

Example target_reduction.1.f90 (`omp_5.0`)

```
program target_reduction_ex1
   interface
      function f(res)
             integer :: f, res
          end function
      function g(res)
             integer :: g, res
          end function
   end interface
   integer :: sum1, sum2, i
   integer, parameter :: n = 100
   sum1 = 0
   sum2 = 0
   !$omp target teams distribute reduction(+:sum1)
```

```
S-15            do i=1,n
S-16               sum1 = sum1 + f(i)
S-17            end do
S-18        !$omp target teams distribute reduction(+:sum2)
S-19            do i=1,n
S-20               sum2 = sum2 + g(i)*sum1
S-21            end do
S-22        print *, "sum1 = ", sum1, ", sum2 = ", sum2
S-23        !!OUTPUT: sum1 =     10100 , sum2 = 153015000
S-24    end program
S-25
S-26
S-27    integer function f(res)
S-28        integer :: res
S-29        !$omp declare target enter(f)
S-30        f = res*2
S-31    end function
S-32    integer function g(res)
S-33        integer :: res
S-34        !$omp declare target enter(g)
S-35        g = res*3
S-36    end function
```

Fortran

In next example, the variables *sum1* and *sum2* remain on the device for the duration of the **target data** region so that it is their device copies that are updated by the reductions. Note the significance of mapping *sum1* on the second **target** construct; otherwise, it would be treated by default as firstprivate and the result computed for *sum1* in the prior **target** region may not be used. Alternatively, a **target update** construct could be used between the two **target** constructs to update the host version of *sum1* with the value that is in the corresponding device version after the completion of the first construct.

C / C++

Example target_reduction.2.c (**omp_5.0**)

```
S-1     #include <stdio.h>
S-2     int f(int);
S-3     int g(int);
S-4     int main()
S-5     {
S-6        int sum1=0, sum2=0;
S-7        const int n = 100;
S-8
S-9        #pragma omp target data map(sum1,sum2)
S-10       {
S-11          #pragma omp target teams distribute reduction(+:sum1)
S-12          for (int i = 0; i < n; i++) {
```

```
S-13            sum1 += f(i);
S-14         }
S-15
S-16         #pragma omp target teams distribute map(sum1) reduction(+:sum2)
S-17         for (int i = 0; i < n; i++) {
S-18            sum2 += g(i) * sum1;
S-19         }
S-20      }
S-21      printf(  "sum1 = %d, sum2 = %d\n", sum1, sum2);
S-22      //OUTPUT: sum1 = 9900, sum2 = 147015000
S-23      return 0;
S-24   }
S-25
S-26   int f(int res){ return res*2; }
S-27   int g(int res){ return res*3; }
```

C / C++

Fortran

1 *Example target_reduction.2.f90* (`omp_5.0`)

```
 S-1   program target_reduction_ex2
 S-2      interface
 S-3         function f(res)
 S-4                integer :: f, res
 S-5             end function
 S-6         function g(res)
 S-7                integer :: g, res
 S-8             end function
 S-9      end interface
S-10      integer :: sum1, sum2, i
S-11      integer, parameter :: n = 100
S-12      sum1 = 0
S-13      sum2 = 0
S-14      !$omp target data map(sum1, sum2)
S-15          !$omp target teams distribute reduction(+:sum1)
S-16              do i=1,n
S-17                 sum1 = sum1 + f(i)
S-18              end do
S-19          !$omp target teams distribute map(sum1) reduction(+:sum2)
S-20              do i=1,n
S-21                 sum2 = sum2 + g(i)*sum1
S-22              end do
S-23      !$omp end target data
S-24      print *, "sum1 = ", sum1, ", sum2 = ", sum2
S-25      !!OUTPUT: sum1 =     10100 , sum2 = 153015000
S-26   end program
S-27
```

```
integer function f(res)
   integer :: res
   !$omp declare target enter(f)
   f = res*2
end function
integer function g(res)
   integer :: res
   !$omp declare target enter(g)
   g = res*3
end function
```

Fortran

10.10.4 Task Reduction with Target Constructs

The following examples illustrate how task reductions can apply to target tasks that result from a **target** construct with the **in_reduction** clause. Here, the **in_reduction** clause specifies that the target task participates in the task reduction defined in the scope of the enclosing **taskgroup** construct. Partial results from all tasks participating in the task reduction will be combined (in some order) into the original variable listed in the **task_reduction** clause before exiting the **taskgroup** region.

C / C++

Example target_task_reduction.1.c (**omp_5.2**)

```
#include <stdio.h>
void device_compute(int *);
#pragma omp declare target enter(device_compute)
void host_compute(int *);
int main()
{
   int sum = 0;

   #pragma omp parallel masked
   #pragma omp taskgroup task_reduction(+:sum)
   {
      #pragma omp target in_reduction(+:sum) nowait
          device_compute(&sum);

      #pragma omp task in_reduction(+:sum)
          host_compute(&sum);
   }
   printf(  "sum = %d\n", sum);
   //OUTPUT: sum = 2
   return 0;
```

```
S-21  }
S-22
S-23  void device_compute(int *sum){ *sum = 1; }
S-24  void   host_compute(int *sum){ *sum = 1; }
```

C / C++

Fortran

Example target_task_reduction.1.f90 (**omp_5.2**)

```
S-1   program target_task_reduction_ex1
S-2      interface
S-3         subroutine device_compute(res)
S-4         !$omp declare target enter(device_compute)
S-5           integer :: res
S-6         end subroutine device_compute
S-7         subroutine host_compute(res)
S-8           integer :: res
S-9         end subroutine host_compute
S-10     end interface
S-11     integer :: sum
S-12     sum = 0
S-13     !$omp parallel masked
S-14        !$omp taskgroup task_reduction(+:sum)
S-15           !$omp target in_reduction(+:sum) nowait
S-16              call device_compute(sum)
S-17           !$omp end target
S-18           !$omp task in_reduction(+:sum)
S-19              call host_compute(sum)
S-20           !$omp end task
S-21        !$omp end taskgroup
S-22     !$omp end parallel masked
S-23     print *, "sum = ", sum
S-24     !!OUTPUT: sum = 2
S-25  end program
S-26
S-27  subroutine device_compute(sum)
S-28     integer :: sum
S-29     !$omp declare target enter(device_compute)
S-30     sum = 1
S-31  end subroutine
S-32  subroutine host_compute(sum)
S-33     integer :: sum
S-34     sum = 1
S-35  end subroutine
```

Fortran

In the next pair of examples, the task reduction is defined by a **reduction** clause with the **task** modifier, rather than a **task_reduction** clause on a **taskgroup** construct. Again, the partial results from the participating tasks will be combined in some order into the original reduction variable, *sum*.

C / C++

Example target_task_reduction.2a.c (**omp_5.2**)

```
#include <stdio.h>
extern void device_compute(int *);
#pragma omp declare target enter(device_compute)
extern void host_compute(int *);
int main()
{
   int sum = 0;

   #pragma omp parallel sections reduction(task, +:sum)
   {
      #pragma omp section
          {
             #pragma omp target in_reduction(+:sum)
             device_compute(&sum);
          }
      #pragma omp section
          {
             host_compute(&sum);
          }
   }
   printf(  "sum = %d\n", sum);
   //OUTPUT: sum = 2
   return 0;
}

void device_compute(int *sum){ *sum = 1; }
void   host_compute(int *sum){ *sum = 1; }
```

C / C++

Fortran

Example target_task_reduction.2a.f90 (**omp_5.2**)

```
S-1   program target_task_reduction_ex2
S-2      interface
S-3         subroutine device_compute(res)
S-4         !$omp declare target enter(device_compute)
S-5           integer :: res
S-6         end subroutine device_compute
S-7         subroutine host_compute(res)
S-8           integer :: res
S-9         end subroutine host_compute
S-10     end interface
S-11     integer :: sum
S-12     sum = 0
S-13     !$omp parallel sections reduction(task,+:sum)
S-14        !$omp section
S-15           !$omp target in_reduction(+:sum) nowait
S-16             call device_compute(sum)
S-17           !$omp end target
S-18        !$omp section
S-19           call host_compute(sum)
S-20     !$omp end parallel sections
S-21     print *, "sum = ", sum
S-22     !!OUTPUT: sum = 2
S-23  end program
S-24
S-25  subroutine device_compute(sum)
S-26     integer :: sum
S-27     !$omp declare target enter(device_compute)
S-28     sum = 1
S-29  end subroutine
S-30  subroutine host_compute(sum)
S-31     integer :: sum
S-32     sum = 1
S-33  end subroutine
```

Fortran

Next, the **task** modifier is again used to define a task reduction over participating tasks. This time, the participating tasks are a target task resulting from a **target** construct with the **in_reduction** clause, and the implicit task (executing on the primary thread) that calls *host_compute*. As before, the partial results from these participating tasks are combined in some order into the original reduction variable.

C / C++

Example target_task_reduction.2b.c (`omp_5.2`)

```
S-1   #include <stdio.h>
S-2   extern void device_compute(int *);
S-3   #pragma omp declare target enter(device_compute)
S-4   extern void host_compute(int *);
S-5   int main()
S-6   {
S-7      int sum = 0;
S-8
S-9      #pragma omp parallel masked reduction(task, +:sum)
S-10     {
S-11         #pragma omp target in_reduction(+:sum) nowait
S-12         device_compute(&sum);
S-13
S-14         host_compute(&sum);
S-15     }
S-16     printf(  "sum = %d\n", sum);
S-17     //OUTPUT: sum = 2
S-18     return 0;
S-19  }
S-20
S-21  void device_compute(int *sum){ *sum = 1; }
S-22  void   host_compute(int *sum){ *sum = 1; }
```

C / C++

Fortran

Example target_task_reduction.2b.f90 (`omp_5.2`)

```
S-1   program target_task_reduction_ex2b
S-2      interface
S-3         subroutine device_compute(res)
S-4         !$omp declare target enter(device_compute)
S-5           integer :: res
S-6         end subroutine device_compute
S-7         subroutine host_compute(res)
S-8           integer :: res
S-9         end subroutine host_compute
S-10     end interface
S-11     integer :: sum
S-12     sum = 0
S-13     !$omp parallel masked reduction(task,+:sum)
S-14           !$omp target in_reduction(+:sum) nowait
S-15             call device_compute(sum)
S-16           !$omp end target
S-17           call host_compute(sum)
```

```
S-18      !$omp end parallel masked
S-19      print *, "sum = ", sum
S-20      !!OUTPUT: sum = 2
S-21   end program
S-22
S-23
S-24   subroutine device_compute(sum)
S-25      integer :: sum
S-26      !$omp declare target enter(device_compute)
S-27      sum = 1
S-28   end subroutine
S-29   subroutine host_compute(sum)
S-30      integer :: sum
S-31      sum = 1
S-32   end subroutine
```

Fortran

10.10.5 Taskloop Reduction

In the OpenMP 5.0 Specification the **taskloop** construct was extended to include the reductions.

The following two examples show how to implement a reduction over an array using taskloop reduction in two different ways. In the first example we apply the **reduction** clause to the **taskloop** construct. As it was explained above in the task reduction examples, a reduction over tasks is divided in two components: the scope of the reduction, which is defined by a **taskgroup** region, and the tasks that participate in the reduction. In this example, the **reduction** clause defines both semantics. First, it specifies that the implicit **taskgroup** region associated with the **taskloop** construct is the scope of the reduction, and second, it defines all tasks created by the **taskloop** construct as participants of the reduction. About the first property, it is important to note that if we add the **nogroup** clause to the **taskloop** construct the code will be nonconforming, basically because we have a set of tasks that participate in a reduction that has not been defined.

C / C++

Example taskloop_reduction.1.c (**omp_5.0**)

```
S-1    #include <stdio.h>
S-2
S-3    int array_sum(int n, int *v) {
S-4        int i;
S-5        int res = 0;
S-6
S-7        #pragma omp taskloop reduction(+: res)
S-8        for(i = 0; i < n; ++i)
S-9            res += v[i];
```

```
S-10
S-11        return res;
S-12    }
S-13
S-14    int main(int argc, char *argv[]) {
S-15        int n = 10;
S-16        int v[10] = {1,2,3,4,5,6,7,8,9,10};
S-17
S-18        #pragma omp parallel
S-19        #pragma omp single
S-20        {
S-21            int res = array_sum(n, v);
S-22            printf("The result is %d\n", res);
S-23        }
S-24        return 0;
S-25    }
```

C / C++

Fortran

1 *Example taskloop_reduction.1.f90* (`omp_5.0`)

```
S-1     function array_sum(n, v) result(res)
S-2         implicit none
S-3         integer :: n, v(n), res
S-4         integer :: i
S-5
S-6         res = 0
S-7         !$omp taskloop reduction(+: res)
S-8         do i=1, n
S-9             res = res + v(i)
S-10        end do
S-11        !$omp end taskloop
S-12
S-13    end function array_sum
S-14
S-15    program main
S-16        implicit none
S-17        integer :: n, v(10), res
S-18        integer :: i
S-19
S-20        integer, external :: array_sum
S-21
S-22        n = 10
S-23        do i=1, n
S-24            v(i) = i
S-25        end do
S-26
```

```
      !$omp parallel
      !$omp single
      res = array_sum(n, v)
      print *, "The result is", res
      !$omp end single
      !$omp end parallel
end program main
```

Fortran

The second example computes exactly the same value as in the preceding *taskloop_reduction.1* code section, but in a very different way. First, in the `array_sum` function a **`taskgroup`** region is created that defines the scope of a new reduction using the **`task_reduction`** clause. After that, a task and also the tasks generated by a taskloop participate in that reduction by using the **`in_reduction`** clause on the **`task`** and **`taskloop`** constructs, respectively. Note that the **`nogroup`** clause was added to the **`taskloop`** construct. This is allowed because what is expressed with the **`in_reduction`** clause is different from what is expressed with the **`reduction`** clause. In one case the generated tasks are specified to participate in a previously declared reduction (**`in_reduction`** clause) whereas in the other case creation of a new reduction is specified and also all tasks generated by the taskloop will participate on it.

C / C++

Example taskloop_reduction.2.c (**`omp_5.0`**)

```
#include <stdio.h>

int array_sum(int n, int *v) {
    int i;
    int res = 0;

    #pragma omp taskgroup task_reduction(+: res)
    {
        if (n > 0) {
            #pragma omp task in_reduction(+: res)
            res = res + v[0];

            #pragma omp taskloop in_reduction(+: res) nogroup
            for(i = 1; i < n; ++i)
                res += v[i];
        }
    }

    return res;
}

int main() {
    int n = 10;
```

```
S-24        int v[10] = {1,2,3,4,5,6,7,8,9,10};
S-25
S-26        #pragma omp parallel
S-27        #pragma omp single
S-28        {
S-29            int res = array_sum(n, v);
S-30            printf("The result is %d\n", res);
S-31        }
S-32        return 0;
S-33    }
```

C / C++

Fortran

1 *Example taskloop_reduction.2.f90* (`omp_5.0`)

```
S-1     function array_sum(n, v) result(res)
S-2         implicit none
S-3         integer :: n, v(n), res
S-4         integer :: i
S-5
S-6         res = 0
S-7         !$omp taskgroup task_reduction(+: res)
S-8         if (n > 0) then
S-9             !$omp task in_reduction(+: res)
S-10            res = res + v(1)
S-11            !$omp end task
S-12
S-13            !$omp taskloop in_reduction(+: res) nogroup
S-14            do i=2, n
S-15                res = res + v(i)
S-16            end do
S-17            !$omp end taskloop
S-18        endif
S-19        !$omp end taskgroup
S-20
S-21    end function array_sum
S-22
S-23    program main
S-24        implicit none
S-25        integer :: n, v(10), res
S-26        integer :: i
S-27
S-28        integer, external :: array_sum
S-29
S-30        n = 10
S-31        do i=1, n
S-32            v(i) = i
```

```
S-33        end do
S-34
S-35        !$omp parallel
S-36        !$omp single
S-37        res = array_sum(n, v)
S-38        print *, "The result is", res
S-39        !$omp end single
S-40        !$omp end parallel
S-41    end program main
```

Fortran

In the OpenMP 5.0 Specification, **reduction** clauses for the **taskloop simd** construct were also added.

The examples below compare reductions for the **taskloop** and the **taskloop simd** constructs. These examples illustrate the use of **reduction** clauses within "stand-alone" **taskloop** constructs, and the use of **in_reduction** clauses for tasks of taskloops to participate with other reductions within the scope of a parallel region.

taskloop reductions:

In the *taskloop reductions* section of the example below, *taskloop 1* uses the **reduction** clause in a **taskloop** construct for a sum reduction, accumulated in *asum*. The behavior is as though a **taskgroup** construct encloses the taskloop region with a **task_reduction** clause, and each taskloop task has an **in_reduction** clause with the specifications of the **reduction** clause. At the end of the taskloop region *asum* contains the result of the reduction.

The next taskloop, *taskloop 2*, illustrates the use of the **in_reduction** clause to participate in a previously defined reduction scope of a **parallel** construct.

The task reductions of *task 2* and *taskloop 2* are combined across the **taskloop** construct and the single **task** construct, as specified in the **reduction(task,+:** *asum***)** clause of the **parallel** construct. At the end of the parallel region *asum* contains the combined result of all reductions.

taskloop simd reductions:

Reductions for the **taskloop simd** construct are shown in the second half of the code. Since each component construct, **taskloop** and **simd**, can accept a reduction clause, the **taskloop simd** construct is a composite construct, and the specific application of the reduction clause is defined within the **taskloop simd** *Construct* section of the OpenMP 5.0 Specification. The code below illustrates use cases for these reductions.

In the *taskloop simd reduction* section of the example below, *taskloop simd 3* uses the **reduction** clause in a **taskloop simd** construct for a sum reduction within a loop. For this case a **reduction** clause is used, as one would use for a **simd** construct. The SIMD reductions of each task are combined, and the results of these tasks are further combined just as in the **taskloop**

construct with the **reduction** clause for *taskloop 1*. At the end of the taskloop region `asum` contains the combined result of all reductions.

If a **taskloop simd** construct is to participate in a previously defined reduction scope, the reduction participation should be specified with a **in_reduction** clause, as shown in the **parallel** region enclosing *task 4* and *taskloop simd 4* code sections.

Here the **taskloop simd** construct's **in_reduction** clause specifies participation of the construct's tasks as a task reduction within the scope of the parallel region. That is, the results of each task of the **taskloop** construct component contribute to the reduction in a broader level, just as in *parallel reduction a* code section above. Also, each **simd**-component construct occurs as if it has a **reduction** clause, and the SIMD results of each task are combined as though to form a single result for each task (that participates in the **in_reduction** clause). At the end of the parallel region `asum` contains the combined result of all reductions.

C / C++

Example taskloop_simd_reduction.1.c (**omp_5.1**)

```
S-1   #include <stdio.h>
S-2   #define N 100
S-3
S-4   int main(){
S-5      int i, a[N], asum=0;
S-6
S-7      for(i=0;i<N;i++) a[i]=i;
S-8
S-9      // taskloop reductions
S-10
S-11     #pragma omp parallel masked
S-12     #pragma omp taskloop reduction(+:asum) // taskloop 1
S-13        for(i=0;i<N;i++){ asum += a[i]; }
S-14
S-15
S-16     #pragma omp parallel reduction(task, +:asum) // parallel reduction a
S-17     {
S-18        #pragma omp masked
S-19        #pragma omp task            in_reduction(+:asum) // task 2
S-20           for(i=0;i<N;i++){ asum += a[i]; }
S-21
S-22        #pragma omp masked taskloop in_reduction(+:asum) // taskloop 2
S-23           for(i=0;i<N;i++){ asum += a[i]; }
S-24     }
S-25
S-26     // taskloop simd reductions
S-27
S-28     #pragma omp parallel masked
S-29     #pragma omp taskloop simd reduction(+:asum) // taskloop simd 3
```

```
      for(i=0;i<N;i++){ asum += a[i]; }


   #pragma omp parallel reduction(task, +:asum) // parallel reduction b
   {
      #pragma omp masked
      #pragma omp task                 in_reduction(+:asum) // task 4
         for(i=0;i<N;i++){ asum += a[i]; }

      #pragma omp masked taskloop simd in_reduction(+:asum) // taskloop
         for(i=0;i<N;i++){ asum += a[i]; }                  // simd 4

  }

  printf("asum=%d \n",asum); // output: asum=29700
}
```

C / C++

Fortran

Example taskloop_simd_reduction.1.f90 (`omp_5.1`)

```
program main

  use omp_lib
  integer, parameter ::  N=100
  integer            :: i, a(N), asum=0

  a = [( i, i=1,N )]    !! initialize

!! taskloop reductions

  !$omp parallel masked
  !$omp taskloop reduction(+:asum)                  !! taskloop 1
    do i=1,N;  asum = asum + a(i);  enddo
  !$omp end taskloop
  !$omp end parallel masked


  !$omp parallel reduction(task, +:asum)            !! parallel reduction a

     !$omp masked
     !$omp task            in_reduction(+:asum)     !! task 2
       do i=1,N;  asum = asum + a(i);  enddo
     !$omp end task
     !$omp end masked

     !$omp masked taskloop in_reduction(+:asum)     !! taskloop 2
```

```
S-27            do i=1,N;  asum = asum + a(i);  enddo
S-28          !$omp end masked taskloop
S-29
S-30        !$omp end parallel
S-31
S-32      !! taskloop simd reductions
S-33
S-34        !$omp parallel masked
S-35        !$omp taskloop simd reduction(+:asum)              !! taskloop simd 3
S-36          do i=1,N;  asum = asum + a(i);  enddo
S-37        !$omp end taskloop simd
S-38        !$omp end parallel masked
S-39
S-40
S-41        !$omp parallel reduction(task, +:asum)             !! parallel reduction b
S-42
S-43          !$omp masked
S-44          !$omp task                 in_reduction(+:asum) !! task 4
S-45             do i=1,N;  asum = asum + a(i);  enddo
S-46          !$omp end task
S-47          !$omp end masked
S-48
S-49          !$omp masked taskloop simd in_reduction(+:asum) !! taskloop simd 4
S-50             do i=1,N;  asum = asum + a(i);  enddo
S-51          !$omp end masked taskloop simd
S-52
S-53        !$omp end parallel
S-54
S-55        print*,"asum=",asum   !! output: asum=30300
S-56
S-57      end program
```

Fortran

10.10.6 Reduction with the `scope` Construct

The following example illustrates the use of the **`scope`** construct to perform a reduction in a **`parallel`** region. The case is useful for producing a reduction and accessing reduction variables inside a **`parallel`** region without using a worksharing-loop construct.

C++

Example scope_reduction.1.cpp (**omp_5.1**)

```
S-1   #include <stdio.h>
S-2   void do_work(int n, float a[], float &s)
S-3   {
S-4      float loc_s = 0.0f;         // local sum
S-5      static int nthrs;
S-6      #pragma omp for
S-7         for (int i = 0; i < n; i++)
S-8            loc_s += a[i];
S-9      #pragma omp single
S-10     {
S-11        s = 0.0f;                // total sum
S-12        nthrs = 0;
S-13     }
S-14     #pragma omp scope reduction(+:s,nthrs)
S-15     {
S-16        s += loc_s;
S-17        nthrs++;
S-18     }
S-19     #pragma omp masked
S-20        printf("total sum = %f, nthrs = %d\n", s, nthrs);
S-21  }
S-22
S-23  float work(int n, float a[])
S-24  {
S-25     float s;
S-26     #pragma omp parallel
S-27     {
S-28        do_work(n, a, s);
S-29     }
S-30     return s;
S-31  }
```

C++

Fortran

Example scope_reduction.1.f90 (`omp_5.1`)

```
S-1   subroutine do_work(n, a, s)
S-2      implicit none
S-3      integer n, i
S-4      real a(*), s, loc_s
S-5      integer, save :: nthrs
S-6
S-7      loc_s = 0.0                ! local sum
S-8      !$omp do
S-9         do i = 1, n
S-10           loc_s = loc_s + a(i)
S-11        end do
S-12     !$omp single
S-13        s = 0.0                 ! total sum
S-14        nthrs = 0
S-15     !$omp end single
S-16     !$omp scope reduction(+:s,nthrs)
S-17        s = s + loc_s
S-18        nthrs = nthrs + 1
S-19     !$omp end scope
S-20     !$omp masked
S-21        print *, "total sum = ", s, ", nthrs = ", nthrs
S-22     !$omp end masked
S-23  end subroutine
S-24
S-25  function work(n, a) result(s)
S-26     implicit none
S-27     integer n
S-28     real a(*), s
S-29
S-30     !$omp parallel
S-31        call do_work(n, a, s)
S-32     !$omp end parallel
S-33  end function
```

Fortran

10.10.7 Reduction on Private Variables in a `parallel` Region

The following example shows reduction on a private variable (*sum_v*) for an orphaned worksharing loop in routine *do_red*, which is called in a **`parallel`** region. At the end of the loop, private variable of each thread should have the same combined value.

C / C++

1 *Example priv_reduction.1.c* (`omp_6.0`)

```
S-1   #include <stdio.h>
S-2   #include <omp.h>
S-3   #define N 100
S-4
S-5   int do_red(int n, int *v)
S-6   {
S-7     int sum_v = 0;      // sum_v is private
S-8
S-9     #pragma omp for reduction(+: sum_v)
S-10    for (int i = 0; i < n; i++) {
S-11      sum_v += v[i];
S-12    }
S-13    return sum_v;
S-14  }
S-15
S-16  int main(void)
S-17  {
S-18    int v[N];
S-19    for (int i = 0; i < N; i++)
S-20      v[i] = i;
S-21
S-22    #pragma omp parallel
S-23    {
S-24      int s_v = do_red(N, v);
S-25      printf("myid %d: sum of v = %d\n", omp_get_thread_num(), s_v);
S-26    }
S-27    return 0;
S-28  }
```

C / C++

Fortran

2 *Example priv_reduction.1.f90* (`omp_6.0`)

```
S-1   function do_red(n, v) result(sum_v)
S-2     implicit none
S-3     integer :: n, v(*)
S-4     integer :: sum_v      ! sum_v is private
S-5     integer :: i
S-6
S-7     sum_v = 0
S-8     !$omp do reduction(+: sum_v)
S-9     do i = 1, n
S-10      sum_v = sum_v + v(i)
S-11    end do
```

```
end function

program priv_red
  use :: omp_lib, only : omp_get_thread_num
  implicit none
  integer, parameter :: N = 100
  integer :: i, v(N), s_v
  integer, external :: do_red

  do i = 1, N
    v(i) = i - 1
  end do

  !$omp parallel private(s_v)
    s_v = do_red(N, v)
    print 10, omp_get_thread_num(), s_v
 10 format("myid ", i0, ": sum of v = ", i0)
  !$omp end parallel
end program
```

Fortran

The following example is slightly modified from the previous example where the **original(private)** modifier is explicitly specified for variable *sum_v* in the **reduction** clause. This modifier indicates that variable *sum_v* is private for reduction as opposed to shared by default for a variable passed as a procedure argument.

C++

Example priv_reduction.2.cpp (**omp_6.0**)

```
#include <stdio.h>
#include <omp.h>
#define N 100

void do_red(int n, int *v, int &sum_v)
{
  sum_v = 0;      // sum_v is private
  #pragma omp for reduction(original(private),+: sum_v)
  for (int i = 0; i < n; i++) {
    sum_v += v[i];
  }
}

int main(void)
{
  int v[N];
  for (int i = 0; i < N; i++)
    v[i] = i;
```

```
S-19
S-20      #pragma omp parallel
S-21      {
S-22        int s_v;      // s_v is private
S-23        do_red(N, v, s_v);
S-24        printf("myid %d: sum of v = %d\n", omp_get_thread_num(), s_v);
S-25      }
S-26      return 0;
S-27    }
```

C++

Fortran

1 *Example priv_reduction.2.f90* (`omp_6.0`)

```
 S-1    subroutine do_red(n, v, sum_v)
 S-2      implicit none
 S-3      integer :: n, v(*)
 S-4      integer :: sum_v
 S-5      integer :: i
 S-6
 S-7      sum_v = 0          ! sum_v is private
 S-8      !$omp do reduction(original(private),+: sum_v)
 S-9      do i = 1, n
S-10        sum_v = sum_v + v(i)
S-11      end do
S-12    end subroutine
S-13
S-14    program priv_red
S-15      use :: omp_lib, only : omp_get_thread_num
S-16      implicit none
S-17      integer, parameter :: N = 100
S-18      integer :: i, v(N), s_v
S-19
S-20      do i = 1, N
S-21        v(i) = i - 1
S-22      end do
S-23
S-24      !$omp parallel private(s_v)
S-25        call do_red(N, v, s_v)
S-26        print 10, omp_get_thread_num(), s_v
S-27     10 format("myid ", i0, ": sum of v = ", i0)
S-28      !$omp end parallel
S-29    end program
```

Fortran

The following example shows the effect of nested **reduction** constructs. For the **parallel** construct, the reduction is on the shared variable x. For the worksharing loop nested inside the **parallel** region, the reduction is performed on the private copy of x for each thread. With 4 threads assigned for the **parallel** region (enforced by the **strict** modifier in the **num_threads** clause), the code should print 40 at the end.

C / C++

Example priv_reduction.3.c (**omp_6.0**)

```
S-1   #include <stdio.h>
S-2
S-3   int main(void)
S-4   {
S-5     int x;
S-6
S-7     x = 0;
S-8     // parallel reduction on shared x
S-9     #pragma omp parallel reduction(+: x) num_threads(strict: 4)
S-10    {
S-11      #pragma omp for reduction(+: x)   // reduction on private x
S-12        for (int i = 0; i < 10; i++)
S-13          x++;
S-14    }
S-15    printf("x = %d\n", x);    // should print 40, with 4 threads
S-16    return 0;
S-17  }
```

C / C++

Fortran

Example priv_reduction.3.f90 (**omp_6.0**)

```
S-1   program nest_red
S-2     implicit none
S-3     integer :: x
S-4
S-5     x = 0
S-6     ! parallel reduction on shared x
S-7     !$omp parallel reduction(+: x) num_threads(strict: 4)
S-8       !$omp do reduction(+: x)   ! reduction on private x
S-9         do i = 1, 10
S-10          x = x + 1
S-11        end do
S-12      !$omp end do
S-13    !$omp end parallel
```

```
  print *, "x =", x            ! should print 40, with 4 threads
end program
```

Fortran

10.10.8 User-Defined Reduction

The **declare reduction** directive can be used to specify user-defined reductions (UDR) for user data types.

In the following example, **declare reduction** directives are used to define *min* and *max* operations for the *point* data structure for computing the rectangle that encloses a set of 2-D points.

Each **declare reduction** directive defines new reduction identifiers, *min* and *max*, to be used in a **reduction** clause. The next item in the declaration list is the data type (*struct point*) used in the reduction. The **combiner** clause specifies the functions *minproc* and *maxproc* to perform the min and max operations, respectively, on the user data (of type *struct point*). In the function argument list are two special OpenMP variable identifiers, **omp_in** and **omp_out**, that denote the two values to be combined in the "real" function; the **omp_out** identifier indicates which one is to hold the result.

The initializer of the **declare reduction** directive specifies the initial value for the private variable of each implicit task. The **omp_priv** identifier is used to denote the private variable.

C / C++

Example udr.1.c (**omp_6.0**)

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

struct point {
  int x;
  int y;
};

void minproc ( struct point *out, struct point *in )
{
  if ( in->x < out->x ) out->x = in->x;
  if ( in->y < out->y ) out->y = in->y;
}

void maxproc ( struct point *out, struct point *in )
{
  if ( in->x > out->x ) out->x = in->x;
  if ( in->y > out->y ) out->y = in->y;
```

```
}

#pragma omp declare reduction(min : struct point) \
        combiner( minproc(&omp_out, &omp_in) ) \
        initializer( omp_priv = { INT_MAX, INT_MAX } )

#pragma omp declare reduction(max : struct point) \
        combiner( maxproc(&omp_out, &omp_in) ) \
        initializer( omp_priv = { 0, 0 } )

void find_enclosing_rectangle ( int n, struct point points[] )
{
  struct point minp = { INT_MAX, INT_MAX }, maxp = {0,0};
  int i;

#pragma omp parallel for reduction(min:minp) reduction(max:maxp)
  for ( i = 0; i < n; i++ ) {
     minproc(&minp, &points[i]);
     maxproc(&maxp, &points[i]);
  }
  printf("min = (%d, %d)\n", minp.x, minp.y);
  printf("max = (%d, %d)\n", maxp.x, maxp.y);
}
```

C / C++

The following example shows the corresponding code in Fortran. The **declare reduction** directives are specified as part of the declaration in subroutine *find_enclosing_rectangle* and the procedures that perform the min and max operations are specified as subprograms.

Fortran

Example udr.1.f90 (omp_6.0)

```
module data_type

  type :: point
    integer :: x
    integer :: y
  end type

end module data_type

subroutine find_enclosing_rectangle ( n, points )
  use data_type
  implicit none
  integer :: n
  type(point) :: points(*)

```

```
S-16      !$omp declare reduction(min : point) &
S-17      !$omp&  combiner( minproc(omp_out, omp_in) ) &
S-18      !$omp&  initializer( omp_priv = point( HUGE(0), HUGE(0) ) )
S-19
S-20      !$omp declare reduction(max : point) &
S-21      !$omp&  combiner( maxproc(omp_out, omp_in) ) &
S-22      !$omp&  initializer( omp_priv = point( 0, 0 ) )
S-23
S-24      type(point) :: minp = point( HUGE(0), HUGE(0) ), maxp = point( 0, 0 )
S-25      integer :: i
S-26
S-27      !$omp parallel do reduction(min:minp) reduction(max:maxp)
S-28      do i = 1, n
S-29         call minproc(minp, points(i))
S-30         call maxproc(maxp, points(i))
S-31      end do
S-32      print *, "min = (", minp%x, minp%y, ")"
S-33      print *, "max = (", maxp%x, maxp%y, ")"
S-34
S-35     contains
S-36      subroutine minproc ( out, in )
S-37        implicit none
S-38        type(point), intent(inout) :: out
S-39        type(point), intent(in) :: in
S-40
S-41        out%x = min( out%x, in%x )
S-42        out%y = min( out%y, in%y )
S-43      end subroutine minproc
S-44
S-45      subroutine maxproc ( out, in )
S-46        implicit none
S-47        type(point), intent(inout) :: out
S-48        type(point), intent(in) :: in
S-49
S-50        out%x = max( out%x, in%x )
S-51        out%y = max( out%y, in%y )
S-52      end subroutine maxproc
S-53
S-54    end subroutine
```

Fortran

The following example shows the same computation as *udr.1* but it illustrates that you can craft complex expressions in the user-defined reduction declaration. In this case, instead of calling the `minproc` and `maxproc` functions we inline the code in a single expression.

C / C++

Example udr.2.c (**omp_6.0**)

```
S-1   #include <stdio.h>
S-2   #include <limits.h>
S-3
S-4   struct point {
S-5     int x;
S-6     int y;
S-7   };
S-8
S-9   #pragma omp declare reduction(min : struct point)  \
S-10        combiner( omp_out.x = omp_in.x > omp_out.x ? omp_out.x : omp_in.x,  \
S-11                  omp_out.y = omp_in.y > omp_out.y ? omp_out.y : omp_in.y ) \
S-12        initializer( omp_priv = { INT_MAX, INT_MAX } )
S-13
S-14  #pragma omp declare reduction(max : struct point)  \
S-15        combiner( omp_out.x = omp_in.x < omp_out.x ? omp_out.x : omp_in.x,  \
S-16                  omp_out.y = omp_in.y < omp_out.y ? omp_out.y : omp_in.y ) \
S-17        initializer( omp_priv = { 0, 0 } )
S-18
S-19  void find_enclosing_rectangle ( int n, struct point points[] )
S-20  {
S-21    struct point minp = { INT_MAX, INT_MAX }, maxp = {0,0};
S-22    int i;
S-23
S-24  #pragma omp parallel for reduction(min:minp) reduction(max:maxp)
S-25    for ( i = 0; i < n; i++ ) {
S-26      if ( points[i].x < minp.x ) minp.x = points[i].x;
S-27      if ( points[i].y < minp.y ) minp.y = points[i].y;
S-28      if ( points[i].x > maxp.x ) maxp.x = points[i].x;
S-29      if ( points[i].y > maxp.y ) maxp.y = points[i].y;
S-30    }
S-31    printf("min = (%d, %d)\n", minp.x, minp.y);
S-32    printf("max = (%d, %d)\n", maxp.x, maxp.y);
S-33  }
```

C / C++

The corresponding code of the same example in Fortran is very similar except that the assignment expression in the **combiner** clause for the **declare reduction** directive can only be used for a single variable, in this case through a type structure constructor *point(...)*.

Fortran

Example udr.2.f90 (**omp_6.0**)

```
module data_type

  type :: point
    integer :: x
    integer :: y
  end type

end module data_type

subroutine find_enclosing_rectangle ( n, points )
  use data_type
  implicit none
  integer :: n
  type(point) :: points(*)

  !$omp declare reduction( min : point )  &
  !$omp&   combiner( omp_out = point(min( omp_out%x, omp_in%x ), &
  !$omp&                              min( omp_out%y, omp_in%y )) ) &
  !$omp&   initializer( omp_priv = point( HUGE(0), HUGE(0) ) )

  !$omp declare reduction( max : point )  &
  !$omp&   combiner( omp_out = point(max( omp_out%x, omp_in%x ), &
  !$omp&                              max( omp_out%y, omp_in%y )) ) &
  !$omp&   initializer( omp_priv = point( 0, 0 ) )

  type(point) :: minp = point( HUGE(0), HUGE(0) ), maxp = point( 0, 0 )
  integer :: i

  !$omp parallel do reduction(min: minp) reduction(max: maxp)
  do i = 1, n
     minp%x = min(minp%x, points(i)%x)
     minp%y = min(minp%y, points(i)%y)
     maxp%x = max(maxp%x, points(i)%x)
     maxp%y = max(maxp%y, points(i)%y)
  end do
  print *, "min = (", minp%x, minp%y, ")"
  print *, "max = (", maxp%x, maxp%y, ")"

end subroutine
```

Fortran

The following example shows the use of special variables in arguments for combiner (**omp_in** and **omp_out**) and initializer (**omp_priv** and **omp_orig**) routines. This example returns the

maximum value of an array and the corresponding index value. The **declare reduction** directive specifies a user-defined reduction operation *maxloc* for data type *struct mx_s*. The function *mx_combine* is the combiner and the function *mx_init* is the initializer.

C / C++

Example udr.3.c (**omp_6.0**)

```
S-1   #include <stdio.h>
S-2   #define N 100
S-3
S-4   struct mx_s {
S-5      float value;
S-6      int index;
S-7   };
S-8
S-9   /* prototype functions for combiner and initializer in
S-10     the declare reduction */
S-11  void mx_combine(struct mx_s *out, struct mx_s *in);
S-12  void mx_init(struct mx_s *priv, struct mx_s *orig);
S-13
S-14  #pragma omp declare reduction(maxloc: struct mx_s) \
S-15          combiner( mx_combine(&omp_out, &omp_in) )  \
S-16          initializer( mx_init(&omp_priv, &omp_orig) )
S-17
S-18  void mx_combine(struct mx_s *out, struct mx_s *in)
S-19  {
S-20     if ( out->value < in->value ) {
S-21        out->value = in->value;
S-22        out->index = in->index;
S-23     }
S-24  }
S-25
S-26  void mx_init(struct mx_s *priv, struct mx_s *orig)
S-27  {
S-28     priv->value = orig->value;
S-29     priv->index = orig->index;
S-30  }
S-31
S-32  int main(void)
S-33  {
S-34     struct mx_s mx;
S-35     float val[N], d;
S-36     int i, count = N;
S-37
S-38     for (i = 0; i < count; i++) {
S-39        d = (N*0.8f - i);
S-40        val[i] = N * N - d * d;
```

```
S-41        }
S-42
S-43        mx.value = val[0];
S-44        mx.index = 0;
S-45        #pragma omp parallel for reduction(maxloc: mx)
S-46        for (i = 1; i < count; i++) {
S-47           if (mx.value < val[i])
S-48           {
S-49              mx.value = val[i];
S-50              mx.index = i;
S-51           }
S-52        }
S-53
S-54        printf("max value = %g, index = %d\n", mx.value, mx.index);
S-55        /* prints 10000, 80 */
S-56
S-57        return 0;
S-58     }
```

C / C++

Below is the corresponding Fortran version of the above example. The **declare reduction** directive specifies the user-defined operation *maxloc* for user-derived type *mx_s*. The combiner *mx_combine* and the initializer *mx_init* are specified as subprograms.

Fortran

Example udr.3.f90 (**omp_6.0**)

```
S-1     program max_loc
S-2        implicit none
S-3        type :: mx_s
S-4           real value
S-5           integer index
S-6        end type
S-7
S-8        !$omp declare reduction(maxloc: mx_s)  &
S-9        !$omp&         combiner( mx_combine(omp_out, omp_in) )  &
S-10       !$omp&         initializer( mx_init(omp_priv, omp_orig) )
S-11
S-12       integer, parameter :: N = 100
S-13       type(mx_s) :: mx
S-14       real :: val(N), d
S-15       integer :: i, count
S-16
S-17       count = N
S-18       do i = 1, count
S-19          d = N*0.8 - i + 1
S-20          val(i) = N * N - d * d
```

```
S-21        enddo
S-22
S-23        mx%value = val(1)
S-24        mx%index = 1
S-25        !$omp parallel do reduction(maxloc: mx)
S-26        do i = 2, count
S-27           if (mx%value < val(i)) then
S-28              mx%value = val(i)
S-29              mx%index = i
S-30           endif
S-31        enddo
S-32
S-33        print *, 'max value = ', mx%value, ' index = ', mx%index
S-34        ! prints 10000, 81
S-35
S-36      contains
S-37
S-38      subroutine mx_combine(out, in)
S-39        implicit none
S-40        type(mx_s), intent(inout) :: out
S-41        type(mx_s), intent(in) :: in
S-42
S-43        if ( out%value < in%value ) then
S-44           out%value = in%value
S-45           out%index = in%index
S-46        endif
S-47      end subroutine mx_combine
S-48
S-49      subroutine mx_init(priv, orig)
S-50        implicit none
S-51        type(mx_s), intent(out) :: priv
S-52        type(mx_s), intent(in) :: orig
S-53
S-54        priv%value = orig%value
S-55        priv%index = orig%index
S-56      end subroutine mx_init
S-57
S-58    end program
```

Fortran

The following example explains a few details of the user-defined reduction in Fortran through modules. The **declare reduction** directive is declared in a module (*data_red*). The reduction-identifier *.add.* is a user-defined operator that is to allow accessibility in the scope that performs the reduction operation. The user-defined operator *.add.* and the subroutine *dt_init* specified in the **initializer** clause are defined in the same subprogram.

The reduction operation (that is, the **reduction** clause) is in the main program. The reduction identifier *.add.* is accessible by use association. Since *.add.* is a user-defined operator, the explicit interface should also be accessible by use association in the current program unit. Since the **declare reduction** associated to this **reduction** clause has the **initializer** clause, the subroutine specified on the clause must be accessible in the current scoping unit. In this case, the subroutine *dt_init* is accessible by use association.

Fortran

Example udr.4.f90 (**omp_6.0**)

```
S-1   module data_red
S-2   ! Declare data type.
S-3     type dt
S-4       real :: r1
S-5       real :: r2
S-6     end type
S-7
S-8   ! Declare the user-defined operator .add.
S-9     interface operator(.add.)
S-10      module procedure addc
S-11    end interface
S-12
S-13  ! Declare the user-defined reduction operator .add.
S-14  !$omp declare reduction(.add. : dt)  &
S-15  !$omp&  combiner( omp_out=omp_out.add.omp_in ) &
S-16  !$omp&  initializer( dt_init(omp_priv) )
S-17
S-18   contains
S-19  ! Declare the initialization routine.
S-20    subroutine dt_init(u)
S-21      type(dt) :: u
S-22      u%r1 = 0.0
S-23      u%r2 = 0.0
S-24    end subroutine
S-25
S-26  ! Declare the specific procedure for the .add. operator.
S-27    function addc(x1, x2) result(xresult)
S-28      type(dt), intent(in) :: x1, x2
S-29      type(dt) :: xresult
S-30      xresult%r1 = x1%r1 + x2%r2
S-31      xresult%r2 = x1%r2 + x2%r1
S-32    end function
S-33
S-34  end module data_red
S-35
S-36  program main
S-37    use data_red, only : dt, dt_init, operator(.add.)
```

```
S-38
S-39      type(dt) :: xdt1, xdt2
S-40      integer :: i
S-41
S-42      xdt1 = dt(1.0,2.0)
S-43      xdt2 = dt(2.0,3.0)
S-44
S-45    ! The reduction operation
S-46    !$omp parallel do reduction(.add.: xdt1)
S-47      do i = 1, 10
S-48        xdt1 = xdt1 .add. xdt2
S-49      end do
S-50    !$omp end parallel do
S-51
S-52      print *, xdt1
S-53
S-54    end program
```

Fortran

The following example uses user-defined reductions to declare a plus (**+**) reduction for a C++ class. As the **declare reduction** directive is inside the context of the *V* class the expressions in the **declare reduction** directive are resolved in the context of the class. Also, note that the **initializer** clause uses a copy constructor to initialize the private variables of the reduction and it uses as parameter to its original variable by using the special variable **omp_orig**.

C++

Example udr.5.cpp (**omp_6.0**)

```
S-1     class V {
S-2        float *p;
S-3        int n;
S-4
S-5     public:
S-6        V( int _n )      : n(_n)  { p = new float[n]; }
S-7        V( const V& m ) : n(m.n) { p = new float[n]; }
S-8        ~V() { delete[] p; }
S-9
S-10       V& operator+= ( const V& );
S-11
S-12       #pragma omp declare reduction( + : V ) combiner( omp_out += omp_in ) \
S-13               initializer(omp_priv(omp_orig))
S-14    };
```

C++

The following examples shows how user-defined reductions can be defined for some STL containers. The first **declare reduction** defines the plus (**+**) operation for

std::vector<int> by making use of the *std::transform* algorithm. The second and third define the merge (or concatenation) operation for *std::vector<int>* and *std::list<int>*. It shows how the user-defined reduction operation can be applied to specific data types of an STL.

C++

Example udr.6.cpp (**omp_6.0**)

```
S-1   #include <algorithm>
S-2   #include <list>
S-3   #include <vector>
S-4
S-5   #pragma omp declare reduction( + : std::vector<int> )          \
S-6        combiner( std::transform (omp_out.begin(), omp_out.end(), \
S-7                  omp_in.begin(), omp_in.end(),std::plus<int>()) )
S-8
S-9   #pragma omp declare reduction( merge : std::vector<int> )    \
S-10       combiner( omp_out.insert(omp_out.end(), omp_in.begin(), \
S-11                 omp_in.end()) )
S-12
S-13  #pragma omp declare reduction( merge : std::list<int> ) \
S-14       combiner( omp_out.merge(omp_in) )
```

C++

10.11 Induction

This section covers ways to perform inductions in **distribute**, worksharing-loop, **taskloop**, and SIMD regions.

10.11.1 `induction` Clause

The following example demonstrates the basic use of the **induction** clause in Case 1 for variable *xi* in a loop in routine *comp_poly* to evaluate the polynomial of variable *x*. For this case, the induction operation is with the inductor '*****' and induction step *x*. The intermediate value of *xi* is used in producing the reduction sum *result*. The last value of *xi* is well defined after the loop and is printed out together with the final value of *result*. An alternative approach is to use an *inscan* reduction as illustrated in Case 2, but this may not be as optimal as Case 1. An equivalent code without the **induction** clause is given in Case 3 where a non-recursive closed form of the induction operation is used to compute the intermediate value of *xi*. The last value of *xi* is returned with the **lastprivate** clause for this case.

C / C++

Example induction.1.c (`omp_6.0`)

```
S-1   #include <stdio.h>
S-2   #include <math.h>
S-3
S-4   void comp_poly(int N, double x, double c[]) {
S-5      // x:    input: value of x for which to eval the polynomial
S-6      // c[N]: input: the coefficients
S-7      double x0 = 1.0;       // initial value x^0 == 1
S-8      double xi;             // x^i
S-9      double result;         // accumulator for the result
S-10
S-11     // Case 1: induction clause
S-12     xi = x0;
S-13     result = 0.0;
S-14     #pragma omp parallel for reduction(+: result) induction(step(x),*: xi)
S-15     for (int i = 0; i < N; i++) {
S-16        result += c[i] * xi;
S-17        xi *= x;
S-18     }
S-19     printf("C1: result = %f, xn = %f\n", result, xi);
S-20
S-21     // Case 2: inscan reduction
S-22     xi = x0;
S-23     result = 0.0;
S-24     #pragma omp parallel for reduction(+: result) reduction(inscan,*: xi)
S-25     for (int i = 0; i < N; i++) {
S-26        result += c[i] * xi;
S-27        #pragma omp scan exclusive(xi)
S-28        xi *= x;
S-29     }
S-30     printf("C2: result = %f, xn = %f\n", result, xi);
S-31
S-32     // Case 3: closed form
S-33     result = 0.0;
S-34     #pragma omp parallel for reduction(+: result) lastprivate(xi)
S-35     for (int i = 0; i < N; i++) {
S-36        xi = x0 * pow(x, i);     // induction operation in closed form
S-37        result += c[i] * xi;
S-38        xi *= x;
S-39     }
S-40     printf("C3: result = %f, xn = %f\n", result, xi);
S-41  }
```

C / C++

Fortran

Example induction.1.f90 (`omp_6.0`)

```
S-1   subroutine comp_poly(N, x, c)
S-2      implicit none
S-3      ! x:    input: value of x for which to eval the polynomial
S-4      ! c(N): input: the coefficients
S-5      integer :: N
S-6      double precision :: x, c(*)
S-7
S-8      double precision :: x0 = 1.0  ! initial value x^0 == 1
S-9      double precision :: xi        ! x^i
S-10     double precision :: result    ! accumulator for the result
S-11     integer :: i
S-12
S-13     !! Case 1: induction clause
S-14     xi = x0
S-15     result = 0.0
S-16     !$omp parallel do reduction(+: result) induction(step(x),*: xi)
S-17     do i = 1, N
S-18        result = result + c(i) * xi
S-19        xi = xi * x
S-20     end do
S-21     print *, 'C1: result =', result, ', xn =', xi
S-22
S-23     !! Case 2: inscan reduction
S-24     xi = x0
S-25     result = 0.0
S-26     !$omp parallel do reduction(+: result) reduction(inscan,*: xi)
S-27     do i = 1, N
S-28        result = result + c(i) * xi
S-29        !$omp scan exclusive(xi)
S-30        xi = xi * x
S-31     end do
S-32     print *, 'C2: result =', result, ', xn =', xi
S-33
S-34     !! Case 3: closed form
S-35     result = 0.0
S-36     !$omp parallel do reduction(+: result) lastprivate(xi)
S-37     do i = 1, N
S-38        xi = x0 * (x ** (i-1))      ! induction operation in closed form
S-39        result = result + c(i) * xi
S-40        xi = xi * x
S-41     end do
S-42     print *, 'C3: result =', result, ', xn =', xi
S-43  end subroutine
```

Fortran

10.11.2 User-defined Induction

The following is a user-defined induction example that uses the **declare induction** directive and the **induction** clause. The example processes in parallel N points along a line of a given slope starting from a given point, and where adjacent points are separated by a fixed distance. The induction variable *P* represents a point, and the step expression is the distance. The induction identifier *next* is defined in the **declare induction** directive with an appropriate *inductor* via the **inductor** clause and *collector* via the **collector** clause. This identifier together with the **step(***Separation***)** modifier is specified in the **induction** clause for the **parallel for/do** construct in routine *processPointsInLine*.

C++

Example induction.2.cpp (**omp_6.0**)

```
S-1   #include <cmath>
S-2
S-3   class Point {
S-4     float x, y, m;
S-5     char color;
S-6   public:
S-7     Point(float x, float y, float m) : x(x), y(y), m(m) {
S-8       color = (int)(x+y) % 256;
S-9     }
S-10    Point nextPoint(float distance) {
S-11      // return a Point that is 'distance' away along slope m
S-12      //   in the x direction
S-13      float deltaX = distance/(sqrtf(1.0f + m * m));
S-14      float deltaY = m * deltaX;
S-15      Point NewPoint(x+deltaX, y+deltaY, m);
S-16      return NewPoint;
S-17    }
S-18  };
S-19
S-20  #pragma omp declare induction(next : (Point, float))                  \
S-21                      inductor (omp_var = omp_var.nextPoint(omp_step)) \
S-22                      collector(omp_step * omp_idx)
S-23
S-24  extern void process(Point P);
S-25
S-26  void processPointsInLine(Point Start, int NumberOfPoints,
S-27                           float Separation) {
S-28    Point P = Start;
S-29    #pragma omp parallel for induction(step(Separation), next : P)
S-30    for (int i = 0; i < NumberOfPoints; ++i) {
S-31      process(P);
S-32      P = P.nextPoint(Separation);
S-33    }
```

```
S-34  }
S-35
S-36  int main() {
S-37    Point Start(1.0f, -2.0f, 0.5f);
S-38    processPointsInLine(Start, 100, 0.25f);
S-39    return 0;
S-40  }
```

C++

Fortran

1 *Example induction.2.f90* (`omp_6.0`)

```
 S-1  module udi
 S-2    integer, parameter :: I2 = selected_int_kind(3)  ! enough for 256
 S-3    type Point
 S-4      real x, y, m
 S-5      integer(I2) color
 S-6     contains
 S-7      procedure initPoint, nextPoint
 S-8    end type
 S-9
S-10    !$omp declare induction(next : (Point, real))                    &
S-11    !$omp&         inductor (omp_var = omp_var%nextPoint(omp_step)) &
S-12    !$omp&         collector(omp_step * omp_idx)
S-13
S-14   contains
S-15    subroutine initPoint(this, x1, y1, m1)
S-16      implicit none
S-17      class(Point) this
S-18      real x1, y1, m1
S-19      this%x = x1; this%y = y1; this%m = m1
S-20      this%color = mod(int(x1+y1), 256)
S-21    end subroutine
S-22
S-23    function nextPoint(this, distance) result(NewPoint)
S-24    ! return a Point that is 'distance' away along slope m in the x direction
S-25      implicit none
S-26      class(Point) this
S-27      real distance
S-28      type(Point) NewPoint
S-29
S-30      real deltaX, deltaY
S-31      deltaX = distance/(sqrt(1.0 + this%m * this%m))
S-32      deltaY = this%m * deltaX
S-33      call NewPoint%initPoint(this%x+deltaX, this%y+deltaY, this%m)
S-34    end function
```

```
S-35    end module
S-36
S-37    subroutine processPointsInLine(Start, NumberOfPoints, Separation)
S-38      use udi
S-39      implicit none
S-40      type(Point) Start
S-41      integer NumberOfPoints
S-42      real Separation
S-43      type(Point) P
S-44      integer i
S-45
S-46      P = Start
S-47      !$omp parallel do induction(step(Separation), next : P)
S-48      do i = 1, NumberOfPoints
S-49        call process(P)
S-50        P = P%nextPoint(Separation)
S-51      end do
S-52    end subroutine
S-53
S-54    program main
S-55      use udi
S-56      implicit none
S-57      type(Point) Start
S-58
S-59      call Start%initPoint(1.0, -2.0, 0.5)
S-60      call processPointsInLine(Start, 100, 0.25)
S-61    end program
```

Fortran

10.12 `scan` Directive

The following examples illustrate how to parallelize a loop that saves the *prefix sum* of a reduction. This is accomplished by using the **`inscan`** modifier in the **`reduction`** clause for the input variable of the scan, and specifying with a **`scan`** directive whether the storage statement includes or excludes the scan input of the present iteration (`k`).

Basically, the **`inscan`** modifier connects a loop and/or SIMD reduction to the scan operation, and a **`scan`** construct with an **`inclusive`** or **`exclusive`** clause specifies whether the "scan phase" (lexical block before and after the directive, respectively) is to use an *inclusive* or *exclusive* scan value for the list item (`x`).

The first example uses the *inclusive* scan operation on a composite loop-SIMD construct. The **`scan`** directive separates the reduction statement on variable `x` from the use of `x` (saving to array `b`). The order of the statements in this example indicates that value `a[k]` (`a(k)` in Fortran) is included in the computation of the prefix sum `b[k]` (`b(k)` in Fortran) for iteration `k`.

C / C++

Example scan.1.c (omp_5.0)

```
S-1   #include <stdio.h>
S-2   #define N 100
S-3
S-4   int main(void)
S-5   {
S-6      int a[N], b[N];
S-7      int x = 0;
S-8
S-9      // initialization
S-10     for (int k = 0; k < N; k++)
S-11        a[k] = k + 1;
S-12
S-13     // a[k] is included in the computation of producing results in b[k]
S-14     #pragma omp parallel for simd reduction(inscan,+: x)
S-15     for (int k = 0; k < N; k++) {
S-16        x += a[k];
S-17        #pragma omp scan inclusive(x)
S-18        b[k] = x;
S-19     }
S-20
S-21     printf("x = %d, b[0:3] = %d %d %d\n", x, b[0], b[1], b[2]);
S-22     //           5050,        1  3  6
S-23
S-24     return 0;
S-25  }
```

C / C++

Fortran

Example scan.1.f90 (omp_5.0)

```
S-1   program inclusive_scan
S-2      implicit none
S-3      integer, parameter :: n = 100
S-4      integer a(n), b(n)
S-5      integer x, k
S-6
S-7      ! initialization
S-8      x = 0
S-9      do k = 1, n
S-10        a(k) = k
S-11     end do
S-12
S-13     ! a(k) is included in the computation of producing results in b(k)
S-14     !$omp parallel do simd reduction(inscan,+: x)
```

```
S-15        do k = 1, n
S-16           x = x + a(k)
S-17           !$omp scan inclusive(x)
S-18           b(k) = x
S-19        end do
S-20
S-21        print *,'x =', x, ', b(1:3) =', b(1:3)
S-22        !           5050,            1  3  6
S-23
S-24     end program
```

Fortran

The second example uses the *exclusive* scan operation on a composite loop-SIMD construct. The **scan** directive separates the use of x (saving to array b) from the reduction statement on variable x. The order of the statements in this example indicates that value `a[k]` (`a(k)` in Fortran) is excluded from the computation of the prefix sum `b[k]` (`b(k)` in Fortran) for iteration k.

C / C++

Example scan.2.c (**omp_5.0**)

```
S-1      #include <stdio.h>
S-2      #define N 100
S-3
S-4      int main(void)
S-5      {
S-6         int a[N], b[N];
S-7         int x = 0;
S-8
S-9         // initialization
S-10        for (int k = 0; k < N; k++)
S-11           a[k] = k + 1;
S-12
S-13        // a[k] is not included in the computation of producing
S-14        // results in b[k]
S-15        #pragma omp parallel for simd reduction(inscan,+: x)
S-16        for (int k = 0; k < N; k++) {
S-17           b[k] = x;
S-18           #pragma omp scan exclusive(x)
S-19           x += a[k];
S-20        }
S-21
S-22        printf("x = %d, b[0:3] = %d %d %d\n", x, b[0], b[1], b[2]);
S-23        //           5050,        0  1  3
S-24
S-25        return 0;
S-26     }
```

C / C++

Fortran

Example scan.2.f90 (**omp_5.0**)

```
S-1   program exclusive_scan
S-2      implicit none
S-3      integer, parameter :: n = 100
S-4      integer a(n), b(n)
S-5      integer x, k
S-6
S-7      ! initialization
S-8      x = 0
S-9      do k = 1, n
S-10        a(k) = k
S-11     end do
S-12
S-13     ! a(k) is not included in the computation of producing results in b(k)
S-14     !$omp parallel do simd reduction(inscan,+: x)
S-15     do k = 1, n
S-16        b(k) = x
S-17        !$omp scan exclusive(x)
S-18        x = x + a(k)
S-19     end do
S-20
S-21     print *,'x =', x, ', b(1:3) =', b(1:3)
S-22     !           5050,            0  1  3
S-23
S-24  end program
```

Fortran

In OpenMP 6.0, the **scan** directive was extended to support the concept of an *initialization* phase where a private variable can be set for later use in the *input* phase of an *exclusive* scan operation. The following example is a rewrite of the previous exclusive scan example, which uses the **scan init_complete** directive to separate the initialization phase from the other phases of the scan operation. The private variable `tmp` is set in the initialization phase and used later in the input phase to update the prefix sum stored in variable `x`. This case allows the same array `c` to be used for both input and output of the scan results.

C / C++

Example scan.3.c (`omp_6.0`)

```
S-1   #include <stdio.h>
S-2   #define N 100
S-3
S-4   int main(void)
S-5   {
S-6      int c[N], tmp;
S-7      int x = 0;
S-8
S-9      // initialization
S-10     for (int k = 0; k < N; k++)
S-11        c[k] = k + 1;
S-12
S-13     // c[k] is used for both input and output of scan results
S-14     #pragma omp parallel for simd reduction(inscan,+: x) private(tmp)
S-15     for (int k = 0; k < N; k++) {
S-16        // initialization phase
S-17        tmp = c[k];
S-18        #pragma omp scan init_complete
S-19
S-20        // scan (output) phase - cannot use tmp here
S-21        c[k] = x;
S-22
S-23        #pragma omp scan exclusive(x)
S-24
S-25        // input phase - can use tmp here
S-26        x += tmp;
S-27     }
S-28
S-29     printf("x = %d, c[0:3] = %d %d %d\n", x, c[0], c[1], c[2]);
S-30     //           5050,       0  1  3
S-31
S-32     return 0;
S-33  }
```

C / C++

Fortran

Example scan.3.f90 (`omp_6.0`)

```
S-1   program inclusive_scan
S-2      implicit none
S-3      integer, parameter :: n = 100
S-4      integer c(n), tmp
S-5      integer x, k
S-6
S-7      ! initialization
S-8      x = 0
S-9      do k = 1, n
S-10        c(k) = k
S-11     end do
S-12
S-13     ! c(k) is used for both input and output of scan results
S-14     !$omp parallel do simd reduction(inscan,+: x) private(tmp)
S-15     do k = 1, n
S-16        ! initialization phase
S-17        tmp = c(k)
S-18        !$omp scan init_complete
S-19
S-20        ! scan (output) phase - cannot use tmp here
S-21        c(k) = x
S-22
S-23        !$omp scan exclusive(x)
S-24
S-25        ! input phase - can use tmp here
S-26        x = x + tmp
S-27     end do
S-28
S-29     print *,'x =', x, ', c(1:3) =', c(1:3)
S-30     !           5050,             0  1  3
S-31
S-32  end program
```

Fortran

10.13 `copyin` Clause

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

C / C++

1 *Example copyin.1.c* (`pre_omp_3.0`)

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

C / C++

Fortran

2 *Example copyin.1.f* (`pre_omp_3.0`)

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

```
!$OMP   END PARALLEL
      END SUBROUTINE COPYIN_EXAMPLE

      SUBROUTINE BUILD
        USE M
        ALLOCATE(WORK(SIZE))
        WORK = TOL
      END SUBROUTINE BUILD
```

Fortran

10.14 `copyprivate` Clause

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

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

C / C++

Example copyprivate.1.c (**`pre_omp_3.0`**)

```
#include <stdio.h>
float x, y;
#pragma omp threadprivate(x, y)

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

C / C++

Fortran

Example copyprivate.1.f (**pre_omp_3.0**)

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

Fortran

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

C / C++

Example copyprivate.2.c (**omp_5.1**)

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

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

C / C++

Fortran

Example copyprivate.2.f (`omp_5.1`)

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

Fortran

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

C / C++

Example copyprivate.3.c (**pre_omp_3.0**)

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

C / C++

Fortran

Example copyprivate.3.f (**pre_omp_3.0**)

```
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 (**pre_omp_3.0**)

```
      SUBROUTINE S(N)
      INTEGER N

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

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

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

Fortran

C++

10.15 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 (**omp_4.5**)

```
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);
    }
}
```

C++

Fortran

10.16 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 Specification 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 (**omp_4.0**)

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

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

Example associate.2.f (**omp_4.0**)

```
      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 Specification). Inside the **parallel** region, *v* has the value of -1 and *u* has the value of the original *v*.

Example associate.3.f90 (**omp_4.0**)

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

The following example illustrates mapping behavior for a Fortran associate name and its selector for a **target** construct.

For the first 3 **target** constructs the associate name *a_aray* is associated with the selector *aray*, an array. For the **target** construct of code block TARGET 1 just the selector *aray* is used and is implicitly mapped, likewise for the associate name *a_aray* in the TARGET 2 block. However, mapping an associate name and its selector is not valid for the same **target** construct. Hence the TARGET 3 block is non-conforming.

In TARGET 4, the *scalr* selector used in the **target** region has an implicit data-sharing attribute of firstprivate since it is a scalar. Hence, the assigned value is not returned. In TARGET 5, the associate name *a_scalr* is implicitly mapped and the assigned value is returned to the host (default **tofrom** mapping behavior). In TARGET 6, the use of the associate name and its selector in the **target** region is conforming because the scalar firstprivate behavior of the selector and the implicit mapping of the associate name are allowed. At the end of the **target** region only the associate name's value is returned to the host. In TARGET 7, the selector and associate name appear in an explicit mapping for the same **target** construct, hence the code block is non-conforming.

Example associate.4.f90 (**omp_5.1**)

```
S-1   program main
S-2     integer :: scalr, aray(3)
```

```
S-3       scalr = -1 ; aray = -1
S-4
S-5       associate(a_scalr=>scalr, a_aray=>aray)
S-6
S-7      !$omp target            !! TARGET 1
S-8        aray = [1,2,3]
S-9       !$omp end target
S-10      print *, a_aray, aray  !!  1 2 3   1 2 3
S-11
S-12      !$omp target           !! TARGET 2
S-13        a_aray = [4,5,6]
S-14      !$omp end target
S-15      print *, a_aray, aray  !!  4 5 6   4 5 6
S-16
S-17     !!!$omp target          !! TARGET 3
S-18     !!                      !! mapping, in this case implicit,
S-19     !!                      !! of aray AND a_aray NOT ALLOWED
S-20     !!    aray = [4,5,6]
S-21     !!  a_aray = [1,2,3]
S-22     !!!$omp end target
S-23
S-24
S-25      !$omp target              !! TARGET 4
S-26        scalr = 1               !! scalr is firstprivate
S-27      !$omp end target
S-28      print *, a_scalr, scalr   !!  -1  -1
S-29
S-30      !$omp target              !! TARGET 5
S-31        a_scalr = 2             !! a_scalr implicitly mapped
S-32      !$omp end target
S-33      print *, a_scalr, scalr   !!  2   2
S-34
S-35      !$omp target              !! TARGET 6
S-36        scalr = 3               !!          scalr is firstprivate
S-37        print *, a_scalr, scalr !!  2   3
S-38        a_scalr = 4             !!          a_scalr implicitly mapped
S-39        print *, a_scalr, scalr !!  4   3
S-40      !$omp end target
S-41      print *, a_scalr, scalr   !!  4   4
S-42
S-43     !!!$omp target map(a_scalr,scalr)  !! TARGET 7
S-44                                        !! mapping, in this case explicit,
S-45                                        !! of scalr AND a_sclar NOT ALLOWED
S-46     !!    scalr = 5
S-47     !!  a_scalr = 5
S-48     !!!$omp end target
S-49
```

```
S-50      end associate
S-51
S-52    end program
```

Fortran

11 Memory Model

OpenMP provides a shared-memory model that allows all threads on a given device shared access to *memory*. For a given OpenMP region that may be executed by more than one thread or SIMD lane, variables in memory may be *shared* or *private* with respect to those threads or SIMD lanes. A variable's data-sharing attribute indicates whether it is shared (the *shared* attribute) or private (the *private*, *firstprivate*, *lastprivate*, *linear*, and *reduction* attributes) in the data environment of an OpenMP region. While private variables in an OpenMP region are new copies of the original variable (with same name) that may then be concurrently accessed or modified by their respective threads or SIMD lanes, a shared variable in an OpenMP region is the same as the variable of the same name in the enclosing region. Concurrent accesses or modifications to a shared variable may therefore require synchronization to avoid data races.

OpenMP's memory model also includes a *temporary view* of memory that is associated with each thread. Two different threads may see different values for a given variable in their respective temporary views. Threads may employ flush operations for the purposes of making their temporary view of a variable consistent with the value of the variable in memory. The effect of a given flush operation is characterized by its flush properties – some combination of *strong*, *release*, and *acquire* – and, for *strong* flushes, a *flush-set*.

A *strong* flush will force consistency between the temporary view and the memory for all variables in its *flush-set*. Furthermore, all strong flushes in a program that have intersecting flush-sets will execute in some total order, and within a thread strong flushes may not be reordered with respect to other memory operations on variables in its flush-set. *Release* and *acquire* flushes operate in pairs. A release flush may "synchronize" with an acquire flush, and when it does so the local memory operations that precede the release flush will appear to have been completed before the local memory operations on the same variables that follow the acquire flush.

Flush operations arise from explicit **`flush`** directives, implicit **`flush`** directives, and also from the execution of **`atomic`** constructs. 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.

In this chapter, examples illustrate how race conditions may arise for accesses to variables with a *shared* data-sharing attribute when flush operations are not properly employed. A race condition can exist when two or more threads are involved in accessing a variable and at least one of the accesses modifies the variable. In particular, a data race will arise when conflicting accesses do not have a well-defined *completion order*. The existence of data races in OpenMP programs result in undefined behavior, and so they should generally be avoided for programs to be correct. The completion order of accesses to a shared variable is guaranteed in OpenMP through a set of

memory consistency rules that are described in the *OpenMP Memory Consistency* section of the OpenMP Specifications document.

11.1 OpenMP Memory Model

The following examples illustrate two major concerns for concurrent thread execution: ordering of thread execution and memory accesses that may or may not lead to race conditions.

In the following example, at Print 1, the value of `xval` could be either 2 or 5, depending on the timing of the threads. The **`atomic`** directives are necessary for the accesses to `x` by threads 1 and 2 to avoid a data race. If the atomic write completes before the atomic read, thread 1 is guaranteed to see 5 in `xval`. Otherwise, thread 1 is guaranteed to see 2 in `xval`.

The barrier after Print 1 contains implicit flushes on all threads, as well as a thread synchronization, so the programmer is guaranteed that the value 5 will be printed by both Print 2 and Print 3. Since neither Print 2 or Print 3 are modifying `x`, they may concurrently access `x` without requiring **`atomic`** directives to avoid a data race.

C / C++

Example mem_model.1.c (**`omp_3.1`**)

```
S-1   #include <stdio.h>
S-2   #include <omp.h>
S-3
S-4   int main(){
S-5     int x;
S-6
S-7     x = 2;
S-8     #pragma omp parallel num_threads(2) shared(x)
S-9     {
S-10
S-11      if (omp_get_thread_num() == 0) {
S-12         #pragma omp atomic write
S-13         x = 5;
S-14      } else {
S-15        int xval;
S-16        #pragma omp atomic read
S-17        xval = x;
S-18      /* Print 1: xval can be 2 or 5 */
S-19        printf("1: Thread# %d: x = %d\n", omp_get_thread_num(), xval);
S-20      }
S-21
S-22      #pragma omp barrier
S-23
S-24      if (omp_get_thread_num() == 0) {
S-25      /* Print 2 */
```

```
S-26          printf("2: Thread# %d: x = %d\n", omp_get_thread_num(), x);
S-27        } else {
S-28        /* Print 3 */
S-29          printf("3: Thread# %d: x = %d\n", omp_get_thread_num(), x);
S-30        }
S-31      }
S-32      return 0;
S-33    }
```

C / C++

Fortran

1 *Example mem_model.1.f90* (`omp_3.1`)

```
S-1     PROGRAM MEMMODEL
S-2       INCLUDE "omp_lib.h"      ! or USE OMP_LIB
S-3       INTEGER X, XVAL
S-4
S-5       X = 2
S-6     !$OMP PARALLEL NUM_THREADS(2) SHARED(X)
S-7
S-8         IF (OMP_GET_THREAD_NUM() .EQ. 0) THEN
S-9         !$OMP ATOMIC WRITE
S-10           X = 5
S-11        ELSE
S-12        !$OMP ATOMIC READ
S-13          XVAL = X
S-14        ! PRINT 1: XVAL can be 2 or 5
S-15          PRINT *,"1: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", XVAL
S-16        ENDIF
S-17
S-18     !$OMP BARRIER
S-19
S-20        IF (OMP_GET_THREAD_NUM() .EQ. 0) THEN
S-21        ! PRINT 2
S-22          PRINT *,"2: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
S-23        ELSE
S-24        ! PRINT 3
S-25          PRINT *,"3: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
S-26        ENDIF
S-27
S-28    !$OMP END PARALLEL
S-29
S-30    END PROGRAM MEMMODEL
```

Fortran

The following example demonstrates why synchronization is difficult to perform correctly through variables. The write to `flag` on thread 0 and the read from `flag` in the loop on thread 1 must be atomic to avoid a data race. When thread 1 breaks out of the loop, `flag` will have the value of 1. However, `data` will still be undefined at the first print statement. Only after the flush of both `flag` and `data` after the first print statement will `data` have the well-defined value of 42.

C / C++

Example mem_model.2.c (**omp_3.1**)

```
#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 */
          #pragma omp atomic write
          flag = 1;
      }
      else if(omp_get_thread_num()==1)
      {
      /* Loop until we see the update to the flag */
          #pragma omp flush(flag, data)
          int flag_val = 0;
          while (flag_val < 1)
          {
             #pragma omp atomic read
             flag_val = flag;
          }
      /* Value of flag is 1; value of data is undefined */
          printf("flag=%d data=%d\n", flag, data);
          #pragma omp flush(flag, data)
      /* Value of flag is 1; value of data is 42 */
          printf("flag=%d data=%d\n", flag, data);
      }
    }
```

```
S-38        return 0;
S-39    }
```

C / C++

Fortran

Example mem_model.2.f (`omp_3.1`)

```
S-1         PROGRAM EXAMPLE
S-2         INCLUDE "omp_lib.h" ! or USE OMP_LIB
S-3         INTEGER DATA
S-4         INTEGER FLAG, FLAG_VAL
S-5
S-6         FLAG = 0
S-7     !$OMP  PARALLEL NUM_THREADS(2)
S-8           IF(OMP_GET_THREAD_NUM() .EQ. 0) THEN
S-9           ! Write to the data buffer that will be read by thread 1
S-10             DATA = 42
S-11
S-12          ! Flush DATA to thread 1 and strictly order the write to DATA
S-13          ! relative to the write to the FLAG
S-14    !$OMP    FLUSH(FLAG, DATA)
S-15
S-16          ! Set FLAG to release thread 1
S-17    !$OMP    ATOMIC WRITE
S-18             FLAG = 1
S-19
S-20          ELSE IF(OMP_GET_THREAD_NUM() .EQ. 1) THEN
S-21          ! Loop until we see the update to the FLAG
S-22    !$OMP    FLUSH(FLAG, DATA)
S-23             FLAG_VAL = 0
S-24             DO WHILE(FLAG_VAL .LT. 1)
S-25    !$OMP       ATOMIC READ
S-26                FLAG_VAL = FLAG
S-27             ENDDO
S-28
S-29          ! Value of FLAG is 1; value of DATA is undefined
S-30             PRINT *, 'FLAG=', FLAG, ' DATA=', DATA
S-31
S-32    !$OMP    FLUSH(FLAG, DATA)
S-33          ! Value of FLAG is 1; value of DATA is 42
S-34             PRINT *, 'FLAG=', FLAG, ' DATA=', DATA
S-35
S-36          ENDIF
S-37    !$OMP  END PARALLEL
S-38        END
```

Fortran

The next example demonstrates why synchronization is difficult to perform correctly through variables. As in the preceding example, the updates to *flag* and the reading of *flag* in the loops on threads 1 and 2 are performed atomically to avoid data races on *flag*. However, the code still contains data race due to the incorrect use of "flush with a list" after the assignment to *data1* on thread 1. By not including *flag* in the flush-set of that **flush** directive, the assignment can be reordered with respect to the subsequent atomic update to *flag*. Consequentially, *data1* is undefined at the print statement on thread 2.

C / C++

Example mem_model.3.c (**omp_3.1**)

```
S-1   #include <omp.h>
S-2   #include <stdio.h>
S-3
S-4   int data0 = 0, data1 = 0;
S-5
S-6   int main()
S-7   {
S-8      int flag=0;
S-9
S-10     #pragma omp parallel num_threads(3)
S-11     {
S-12        if(omp_get_thread_num()==0)
S-13        {
S-14           data0 = 17;
S-15           #pragma omp flush
S-16           /* Set flag to release thread 1 */
S-17           #pragma omp atomic update
S-18           flag++;
S-19           /* Flush of flag is implied by the atomic directive */
S-20        }
S-21        else if(omp_get_thread_num()==1)
S-22        {
S-23           int flag_val = 0;
S-24           /* Loop until we see that flag reaches 1*/
S-25           while(flag_val < 1)
S-26           {
S-27              #pragma omp atomic read
S-28              flag_val = flag;
S-29           }
S-30           #pragma omp flush
S-31           /* data0 is 17 here */
S-32           printf("Thread 1 awoken (data0 = %d)\n", data0);
S-33           data1 = 42;
S-34           #pragma omp flush(data1)
S-35           /* Set flag to release thread 2 */
S-36           #pragma omp atomic update
```

```
S-37            flag++;
S-38            /* Flush of flag is implied by the atomic directive */
S-39         }
S-40         else if(omp_get_thread_num()==2)
S-41         {
S-42            int flag_val = 0;
S-43            /* Loop until we see that flag reaches 2 */
S-44            while(flag_val < 2)
S-45            {
S-46               #pragma omp atomic read
S-47               flag_val = flag;
S-48            }
S-49            #pragma omp flush(data0,data1)
S-50            /* there is a data race here;
S-51               data0 is 17 and data1 is undefined */
S-52            printf("Thread 2 awoken (data0 = %d, data1 = %d)\n",
S-53                  data0, data1);
S-54         }
S-55      }
S-56      return 0;
S-57   }
```

C / C++

Fortran

1 *Example mem_model.3.f* (`omp_3.1`)

```
S-1          PROGRAM EXAMPLE
S-2          INCLUDE "omp_lib.h" ! or USE OMP_LIB
S-3          INTEGER FLAG, FLAG_VAL
S-4          INTEGER DATA0, DATA1
S-5
S-6          FLAG = 0
S-7    !$OMP  PARALLEL NUM_THREADS(3)
S-8            IF(OMP_GET_THREAD_NUM() .EQ. 0) THEN
S-9                DATA0 = 17
S-10   !$OMP       FLUSH
S-11
S-12           ! Set flag to release thread 1
S-13   !$OMP       ATOMIC UPDATE
S-14               FLAG = FLAG + 1
S-15           ! Flush of FLAG is implied by the atomic directive
S-16
S-17           ELSE IF(OMP_GET_THREAD_NUM() .EQ. 1) THEN
S-18           ! Loop until we see that FLAG reaches 1
S-19               FLAG_VAL = 0
S-20               DO WHILE(FLAG_VAL .LT. 1)
S-21   !$OMP          ATOMIC READ
```

```
S-22                    FLAG_VAL = FLAG
S-23                 ENDDO
S-24    !$OMP        FLUSH
S-25
S-26             ! DATA0 is 17 here
S-27                 PRINT *, 'Thread 1 awoken. DATA0 = ', DATA0
S-28
S-29                 DATA1 = 42
S-30    !$OMP        FLUSH(DATA1)
S-31
S-32             ! Set FLAG to release thread 2
S-33    !$OMP        ATOMIC UPDATE
S-34                 FLAG = FLAG + 1
S-35             ! Flush of FLAG is implied by the atomic directive
S-36
S-37             ELSE IF(OMP_GET_THREAD_NUM() .EQ. 2) THEN
S-38             ! Loop until we see that FLAG reaches 2
S-39                 FLAG_VAL = 0
S-40                 DO WHILE(FLAG_VAL .LT. 2)
S-41    !$OMP           ATOMIC READ
S-42                    FLAG_VAL = FLAG
S-43                 ENDDO
S-44    !$OMP        FLUSH(DATA0, DATA1)
S-45
S-46             ! There is a data race here; data0 is 17 and data1 is undefined
S-47                 PRINT *, 'Thread 2 awoken. DATA0 = ', DATA0,
S-48         &                ' and DATA1 = ', DATA1
S-49
S-50             ENDIF
S-51    !$OMP  END PARALLEL
S-52           END
```

Fortran

The following two examples illustrate the ordering properties of the *flush* operation. The *flush* operations are strong flushes that are applied to the specified flush lists. However, use of a **flush** construct with a list is extremely error prone and users are strongly discouraged from attempting it. In the codes the programmer intends to prevent simultaneous execution of the protected section by the two threads. The atomic directives in the codes ensure that the accesses to shared variables `a` and `b` are atomic write and atomic read operations. Otherwise both examples would contain data races and automatically result in unspecified behavior.

In the following incorrect code example, operations on variables `a` and `b` are not ordered with respect to each other. For instance, nothing prevents the compiler from moving the flush of `b` on thread 0 or the flush of `a` on thread 1 to a position completely after the protected section (assuming that the protected section on thread 0 does not reference `b` and the protected section on thread 1 does not reference `a`). If either re-ordering happens, both threads can simultaneously execute the

protected section. Any shared data accessed in the protected section is not guaranteed to be current or consistent during or after the protected section.

C / C++

Example mem_model.4a.c (`omp_3.1`)

```
#include <omp.h>

void flush_incorrect()
{
  int a, b;
  a = b = 0;
  #pragma omp parallel num_threads(2)
  {
    int myid = omp_get_thread_num();
    int tmp;

    if ( myid == 0 ) {          // thread 0
      #pragma omp atomic write
        b = 1;
      #pragma omp flush(b)      // flushes are not ordered
      #pragma omp flush(a)      // compiler may move them around
      #pragma omp atomic read
        tmp = a;
    }
    else {                      // thread 1
      #pragma omp atomic write
        a = 1;
      #pragma omp flush(a)      // flushes are not ordered
      #pragma omp flush(b)      // compiler may move them around
      #pragma omp atomic read
        tmp = b;
    }
    if ( tmp == 0 ) {           // exclusive access not guaranteed
      /* protected section */
    }
  }
}
```

C / C++

Fortran

Example mem_model.4a.f90 (`omp_3.1`)

```
S-1   subroutine flush_incorrect
S-2     use omp_lib
S-3     implicit none
S-4     integer a, b, tmp
S-5     integer myid
S-6
S-7     a = 0; b = 0
S-8     !$omp parallel private(myid,tmp) num_threads(2)
S-9       myid = omp_get_thread_num()
S-10
S-11      if ( myid == 0 ) then      ! thread 0
S-12        !$omp atomic write
S-13          b = 1
S-14        !$omp flush(b)           ! flushes are not ordered
S-15        !$omp flush(a)           ! compiler may move them around
S-16        !$omp atomic read
S-17          tmp = a
S-18      else                       ! thread 1
S-19        !$omp atomic write
S-20          a = 1
S-21        !$omp flush(a)           ! flushes are not ordered
S-22        !$omp flush(b)           ! compiler may move them around
S-23        !$omp atomic read
S-24          tmp = b
S-25      endif
S-26      if ( tmp == 0 ) then       ! exclusive access not guaranteed
S-27        !! protected section
S-28      endif
S-29    !$omp end parallel
S-30  end subroutine
```

Fortran

The following code example correctly ensures that the protected section is executed by only one thread at a time. Execution of the protected section by neither thread is considered correct in this example. This occurs if both flushes complete prior to either thread executing its **if** statement for the protected section. The compiler is prohibited from moving the flush at all for either thread, ensuring that the respective assignment is complete and the data is flushed before the **if** statement is executed.

C / C++

1 *Example mem_model.4b.c* (`omp_3.1`)

```
S-1   #include <omp.h>
S-2
S-3   void flush_correct()
S-4   {
S-5     int a, b;
S-6     a = b = 0;
S-7     #pragma omp parallel num_threads(2)
S-8     {
S-9       int myid = omp_get_thread_num();
S-10      int tmp;
S-11
S-12      if ( myid == 0 ) {          // thread 0
S-13        #pragma omp atomic write
S-14          b = 1;
S-15        #pragma omp flush(a,b)    // flushes are ordered
S-16        #pragma omp atomic read
S-17          tmp = a;
S-18      }
S-19      else {                      // thread 1
S-20        #pragma omp atomic write
S-21          a = 1;
S-22        #pragma omp flush(a,b)    // flushes are ordered
S-23        #pragma omp atomic read
S-24          tmp = b;
S-25      }
S-26      if ( tmp == 0 ) {           // access by single thread
S-27        /* protected section */
S-28      }
S-29    }
S-30  }
```

C / C++

Fortran

2 *Example mem_model.4b.f90* (`omp_3.1`)

```
S-1   subroutine flush_correct
S-2     use omp_lib
S-3     implicit none
S-4     integer a, b, tmp
S-5     integer myid
S-6
S-7     a = 0; b = 0
S-8     !$omp parallel private(myid,tmp) num_threads(2)
S-9       myid = omp_get_thread_num()
```

```
S-10
S-11        if ( myid == 0 ) then      ! thread 0
S-12          !$omp atomic write
S-13            b = 1
S-14          !$omp flush(a,b)         ! flushes are ordered
S-15          !$omp atomic read
S-16            tmp = a
S-17        else                       ! thread 1
S-18          !$omp atomic write
S-19            a = 1
S-20          !$omp flush(a,b)         ! flushes are ordered
S-21          !$omp atomic read
S-22            tmp = b
S-23        endif
S-24        if ( tmp == 0 ) then       ! access by single thread
S-25          !! protected section
S-26        endif
S-27      !$omp end parallel
S-28    end subroutine
```

Fortran

11.2 Memory Allocators

OpenMP memory allocators can be used to allocate memory with specific allocator traits. In the following example an OpenMP allocator is used to specify an alignment for arrays `x` and `y`. The general approach for attributing traits to variables allocated by OpenMP is to create or specify a pre-defined *memory space*, create an array of *traits*, and then form an *allocator* from the memory space and trait. The allocator is then specified in an OpenMP allocation (using an API **`omp_alloc()`** function for C/C++ code and an **`allocators`** directive for Fortran code in the *allocators.1* example).

In the example below the `xy_memspace` variable is declared and assigned the default memory space (**`omp_default_mem_space`**). Next, an array for *traits* is created. Since only one trait will be used, the array size is `1`. A trait is a structure in C/C++ and a derived type in Fortran, containing 2 components: a key and a corresponding value (key-value pair). The trait key used here is **`omp_atk_alignment`** (an enum for C/C++ and a parameter for Fortran) and the trait value of 64 is specified in the `xy_traits` declaration. These declarations are followed by a call to the **`omp_init_allocator()`** function to combine the memory space (`xy_memspace`) and the traits (`xy_traits`) to form an allocator (`xy_alloc`).

In the C/C++ code the API **`omp_allocate()`** function is used to allocate space, similar to **`malloc`**, except that the allocator is specified as the second argument. In Fortran an **`allocators`** directive is used to specify an allocator for the following Fortran **`allocate`** statement. A variable list in the **`allocate`** clause may be supplied if the allocator is to be applied

to a subset of variables in the Fortran allocate statement. Here, the *xy_alloc* allocator is specified in the modifier of the **allocator** clause, and the set of all variables used in the **allocate** statement is specified in the list.

C / C++

Example allocators.1.c (**omp_5.0**)

```
#include    <omp.h>
#include  <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#define N 1000

int main()
{
   float  *x, *y;
   float s=2.0;

   omp_memspace_handle_t  xy_memspace = omp_default_mem_space;
   omp_alloctrait_t       xy_traits[1]= {omp_atk_alignment, 64};
   omp_allocator_handle_t xy_alloc    =
                           omp_init_allocator(xy_memspace,1,xy_traits);

   x=(float *)omp_alloc(N*sizeof(float), xy_alloc);
   y=(float *)omp_alloc(N*sizeof(float), xy_alloc);

   if( ((intptr_t)(y))%64 != 0 || ((intptr_t)(x))%64 != 0 )
   {
      printf("ERROR: x|y not 64-Byte aligned\n");
      exit(1);
   }

   #pragma omp parallel
   {
      #pragma omp for simd simdlen(16) aligned(x,y:64)
      for(int i=0; i<N; i++){ x[i]=i+1; y[i]=i+1; } // initialize

      #pragma omp for simd simdlen(16) aligned(x,y:64)
      for(int i=0; i<N; i++) y[i] = s*x[i] + y[i];
    }

   printf("y[0],y[N-1]: %5.0f %5.0f\n",y[0],y[N-1]);
   // output y[0],y[N-1]: 3 3000

   omp_free(x, xy_alloc);
   omp_free(y, xy_alloc);
   omp_destroy_allocator(xy_alloc);
```

```
   return 0;
}
```

C / C++

Fortran

Example allocators.1.f90 (omp_5.2)

```
program main
 use omp_lib

 integer, parameter :: N=1000
 real, allocatable  :: x(:),y(:)
 real               :: s = 2.0e0
 integer            :: i

 integer(omp_memspace_handle_kind ) :: xy_memspace = omp_default_mem_space
 type(   omp_alloctrait           ) :: xy_traits(1) = &
                                       [omp_alloctrait(omp_atk_alignment,64)]
 integer(omp_allocator_handle_kind) :: xy_alloc

   xy_alloc   =    omp_init_allocator(   xy_memspace, 1, xy_traits)

   !$omp allocators allocate(allocator(xy_alloc): x, y)
   allocate(x(N),y(N))
                         !! loc is non-standard, but found everywhere
                         !! remove these lines if not available
   if(modulo(loc(x),64) /= 0 .and. modulo(loc(y),64) /=0 ) then
      print*,"ERROR: x|y not 64-byte aligned"; stop
   endif

   !$omp parallel

      !$omp do simd simdlen(16) aligned(x,y: 64) !! 64B aligned
      do i=1,N  !! initialize
        x(i)=i
        y(i)=i
      end do

      !$omp do simd simdlen(16) aligned(x,y: 64) !! 64B aligned
      do i = 1,N
         y(i) = s*x(i) + y(i)
      end do

   !$omp end parallel

   write(*,'("y(1),y(N):",2f6.0)') y(1),y(N) !!output: y... 3. 3000.
```

```
   deallocate(x,y)
   call omp_destroy_allocator(xy_alloc)

end program
```

Fortran

When using the **allocators** construct with optional clauses in Fortran code, users should be aware of the behavior of a reallocation.

In the following example, the *a* variable is allocated with 64-byte alignment through the **align** clause of the **allocators** construct. The alignment of the newly allocated object, *a*, in the (reallocation) assignment *a = b* will not be reallocated with the 64-byte alignment, but with the 32-byte alignment prescribed by the trait of the *my_alloctr* allocator. It is best to avoid this problem by constructing and using an allocator (not the **align** clause) with the required alignment in the **allocators** construct. Note that in the subsequent deallocation of *a* the deallocation must precede the destruction of the allocator used in the allocation of *a*.

Fortran

Example allocators.2.f90 (**omp_5.2**)

```
program main
   use omp_lib
   implicit none

   integer, parameter :: align_32=32
   real, allocatable  :: a(:,:)
   real               :: b(10,10)

   integer(omp_memspace_handle_kind ) :: my_memspace
   type(   omp_alloctrait           ) :: my_traits(1)
   integer(omp_allocator_handle_kind) :: my_alloctr

   my_memspace  =  omp_default_mem_space
   my_traits    = [omp_alloctrait(omp_atk_alignment,align_32)]
!                                      allocator alignment ^^
   my_alloctr   =  omp_init_allocator(my_memspace, 1, my_traits)

   !$omp allocators allocate(allocator(my_alloctr), align(64): a)
   allocate(a(5,5)) ! 64-byte aligned by clause <---------^^

   a = b  ! reallocation occurs with 32-byte alignment
          ! uses just my_alloctr (32-byte align from allocator)

   deallocate(a)  ! Uses my_alloctr in deallocation.
   call omp_destroy_allocator(my_alloctr)
```

```
end program main
```

Fortran

When creating and using an **allocators** construct within a Fortran procedure for allocating storage (and subsequently freeing the allocator storage with an **omp_destroy_allocator** construct), users should be aware of the necessity of using an explicit Fortran deallocation instead of relying on auto-deallocation.

In the following example, a user-defined allocator is used in the allocation of the *c* variable, and then the allocator is destroyed. Auto-deallocation at the end of the *broken_auto_deallocation* procedure will fail without the allocator, hence an explicit deallocation should be used (before the **omp_destroy_allocator** construct). Note that an allocator may be specified directly in the **allocate** clause without using the **allocator** complex modifier, so long as no other modifier is specified in the clause.

Fortran

Example allocators.3.f90 (**omp_5.2**)

```
subroutine broken_auto_deallocation
   use omp_lib
   implicit none
   integer, parameter :: align_32=32
   real, allocatable  :: c(:)

   integer(omp_memspace_handle_kind ) :: my_memspace
   type(   omp_alloctrait           ) :: my_traits(1)
   integer(omp_allocator_handle_kind) :: my_alloctr

   my_memspace  =  omp_default_mem_space
   my_traits    = [omp_alloctrait(omp_atk_alignment,align_32)]
   my_alloctr   =  omp_init_allocator(my_memspace, 1, my_traits)

   !$omp allocators allocate(my_alloctr: c)
   allocate(c(100))

   !...

   call omp_destroy_allocator(my_alloctr)
   ! Auto-deallocation of c fails,
   ! because my_alloctr is no longer available.

end subroutine
```

Fortran

The **allocate** directive is a convenient way to apply an OpenMP allocator to the allocation of declared variables.

This example illustrates the allocation of specific types of storage in a program for use in libraries, privatized variables, and with offloading.

Two groups of variables, {*v1, v2*} and {*v3, v4*}, are used with the **allocate** directive, and the {*v5, v6*} pair is used with the **allocate** clause. Here we explicitly use predefined allocators **omp_high_bw_mem_alloc** and **omp_default_mem_alloc** with the **allocate** directive in CASE 1. Similar effects are achieved for private variables of a task by using the **allocate** clause, as shown in CASE 2.

Note, when the **allocate** directive does not specify an **allocator** clause, an implementation-defined default, stored in the *def-allocator-var* ICV, is used (not illustrated here). Users can set and get the default allocator with the **omp_set_default_allocator** and **omp_get_default_allocator** API routines.

C / C++

Example allocators.4.c (**omp_5.1**)

```
S-1   #include <omp.h>
S-2   #include <stdio.h>
S-3
S-4   void my_init(double *,double *,int, double *,double *,int, \
S-5                double *,double *,int);
S-6   void lib_saxpy(double *,double *,double,int);
S-7   void my_gather(double *,double *,int);
S-8
S-9   #pragma omp begin declare target
S-10  void my_gpu_vxv(double *, double *, int);
S-11  #pragma omp end  declare target
S-12
S-13  #define Nhb 1024*1024      // high bandwith
S-14  #define Nbg 1024*1024*64   // big memory, default
S-15  #define Nll 1024*1024      // low latency memory
S-16
S-17  void test_allocate() {
S-18
S-19    double  v1[Nhb], v2[Nhb];
S-20    double  v3[Nbg], v4[Nbg];
S-21    double  v5[Nll], v6[Nll];
S-22
S-23  /*** CASE 1: USING ALLOCATE DIRECTIVE ***/
S-24    #pragma omp allocate(v1,v2) allocator(omp_high_bw_mem_alloc)
S-25    #pragma omp allocate(v3,v4) allocator(omp_default_mem_alloc)
S-26
S-27    my_init(v1,v2,Nhb, v3,v4,Nbg, v5,v6,Nll);
S-28
```

```
S-29      lib_saxpy(v1,v2,5.0,Nhb);
S-30
S-31      #pragma omp target map(to: v3[0:Nbg], v4[0:Nbg]) map(from:v3[0:Nbg])
S-32      my_gpu_vxv(v3,v4,Nbg);
S-33
S-34    /*** CASE 2: USING ALLOCATE CLAUSE ***/
S-35      #pragma omp task private(v5,v6) \
S-36                       allocate(allocator(omp_low_lat_mem_alloc): v5,v6)
S-37      {
S-38        my_gather(v5,v6,Nll);
S-39      }
S-40
S-41    }
```

C / C++

Fortran

1 *Example allocators.4.f90* (`omp_5.1`)

```
S-1     subroutine test_allocate
S-2        use omp_lib
S-3
S-4        interface
S-5          subroutine my_gpu_vxv(va,vb,n)
S-6          !$omp declare target
S-7          integer :: n
S-8          double precision  :: va(n), vb(n)
S-9          end subroutine
S-10       end interface
S-11
S-12       integer,parameter :: Nhb=1024*1024,    & !! high bandwith
S-13                            Nbg=1024*1024*64,& !! big memory, default
S-14                            Nll=1024*1024      !! low latency memory
S-15
S-16       double precision  ::  v1(Nhb), v2(Nhb)
S-17       double precision  ::  v3(Nbg), v4(Nbg)
S-18       double precision  ::  v5(Nll), v6(Nll)
S-19
S-20     !*** CASE 1: USING ALLOCATE DIRECTIVE ***!
S-21       !$omp allocate(v1,v2) allocator(omp_high_bw_mem_alloc)
S-22       !$omp allocate(v3,v4) allocator(omp_default_mem_alloc)
S-23
S-24       call my_init(v1,v2,Nhb, v3,v4,Nbg, v5,v6,Nll)
S-25
S-26       call lib_saxpy(v1,v2,5.0,Nhb)
S-27
S-28       !$omp target map(to: v3, v4) map(from:v3)
S-29          call my_gpu_vxv(v3,v4,Nbg)
```

```
S-30        !$omp end target
S-31
S-32      !*** CASE 2: USING ALLOCATE CLAUSE ***!
S-33        !$omp task private(v5,v6) &
S-34        !$omp&     allocate(allocator(omp_low_lat_mem_alloc): v5,v6)
S-35           call my_gather(v5,v6,N11)
S-36        !$omp end task
S-37
S-38    end subroutine test_allocate
```

Fortran

The use of allocators in **target** regions is facilitated by the **uses_allocators** clause as shown in the cases below.

In CASE 1, the predefined **omp_cgroup_mem_alloc** allocator is made available on the device in the first **target** construct as specified in the **uses_allocators** clause. The allocator is then used in the **allocate** clause of the **teams** construct to allocate a private array for each team (contention group). The private *xbuf* arrays that are filled by each team are reduced as specified in the **reduction** clause on the **teams** construct.

In CASE 2, user-defined traits are specified in the *cgroup_traits* variable. An allocator is initialized for the **target** region in the **uses_allocators** clause, and the traits specified in *cgroup_traits* are included by the **traits** modifier.

In CASE 3, the *cgroup_alloc* variable is initialized on the host with traits and a memory space. However, these are ignored by the **uses_allocators** clause and a new allocator for the **target** region is initialized with default traits.

C / C++

Example allocators.5.c (**omp_5.2**)

```
S-1     #include <omp.h>
S-2     #include <stdio.h>
S-3
S-4     int calc(int i, int j) { return i * j;}
S-5     #pragma omp declare target(calc)
S-6
S-7     int main()
S-8     {
S-9         #define N 256
S-10        int sum;
S-11        int xbuf[N];
S-12
S-13        omp_allocator_handle_t cgroup_alloc;
S-14        const omp_alloctrait_t cgroup_traits[1]=
S-15                               {{omp_atk_access,omp_atv_cgroup}};
S-16
```

```
S-17        for (int i = 0; i < N; i++) { xbuf[i] = 0; }
S-18
S-19    /*** CASE 1: USING ALLOCATE DIRECTIVE ***/
S-20        // uses predefined allocator omp_cgroup_mem_alloc
S-21        #pragma omp target uses_allocators(omp_cgroup_mem_alloc)
S-22        #pragma omp teams reduction(+:xbuf) thread_limit(N) \
S-23                          allocate(omp_cgroup_mem_alloc:xbuf) num_teams(4)
S-24        {
S-25            #pragma omp parallel for
S-26            for (int i = 0; i < N; i++) {
S-27                xbuf[i] += calc(i,omp_get_team_num());
S-28            }
S-29        }
S-30
S-31        sum = 0;
S-32        #pragma omp parallel for reduction(+:sum)
S-33        for (int i = 0; i < N; i++) {
S-34            sum += xbuf[i];
S-35        }
S-36        if(sum == 3*(N-1)*N) printf("PASSED 1 of 3\n");
S-37
S-38    /*** CASE 2: ***/
S-39
S-40        for (int i = 0; i < N; i++) { xbuf[i] = 0; }
S-41
S-42        cgroup_alloc = omp_null_allocator;
S-43
S-44        // uses custom allocator with specified traits
S-45        #pragma omp target uses_allocators(traits(cgroup_traits): cgroup_alloc)
S-46        #pragma omp teams reduction(+:xbuf) thread_limit(N) \
S-47                          allocate(cgroup_alloc:xbuf) num_teams(4)
S-48        {
S-49            #pragma omp parallel for
S-50            for (int i = 0; i < N; i++) {
S-51                xbuf[i] += calc(i,omp_get_team_num());
S-52            }
S-53        }
S-54
S-55        sum = 0;
S-56        #pragma omp parallel for reduction(+:sum)
S-57        for (int i = 0; i < N; i++) {
S-58            sum += xbuf[i];
S-59        }
S-60        if(sum == 3*(N-1)*N) printf("PASSED 2 of 3\n");
S-61
S-62
S-63    /*** CASE 3: ***/
```

```
S-64
S-65        for (int i = 0; i < N; i++) { xbuf[i] = 0; }
S-66
S-67        cgroup_alloc = omp_init_allocator(
S-68                       omp_default_mem_space, 1, cgroup_traits);
S-69
S-70        // WARNING: uses custom allocator but with DEFAULT traits
S-71        #pragma omp target uses_allocators(cgroup_alloc)
S-72        #pragma omp teams reduction(+:xbuf) thread_limit(N) \
S-73                          allocate(cgroup_alloc:xbuf) num_teams(4)
S-74        {
S-75            #pragma omp parallel for
S-76            for (int i = 0; i < N; i++) {
S-77                xbuf[i] += calc(i,omp_get_team_num());
S-78            }
S-79        }
S-80        omp_destroy_allocator(cgroup_alloc);
S-81
S-82        sum = 0;
S-83        #pragma omp parallel for reduction(+:sum)
S-84        for (int i = 0; i < N; i++) {
S-85            sum += xbuf[i];
S-86        }
S-87        if(sum == 3*(N-1)*N) printf("PASSED 3 of 3\n");
S-88
S-89        return 0;
S-90    }
```

C / C++

Fortran

1 *Example allocators.5.f90* (omp_5.2)

```
S-1     module functions
S-2     contains
S-3        function calc(i,j)  result(ii)
S-4          implicit none
S-5          integer :: i,j,ii
S-6          !$omp declare target(calc)
S-7
S-8          ii = i*j
S-9        end function
S-10    end module
S-11
S-12    program main
S-13
S-14        use omp_lib
S-15        use functions
```

```
S-16        implicit none
S-17        integer, parameter :: N=256
S-18        integer :: sum, i
S-19        integer :: xbuf(N)
S-20
S-21        integer( omp_allocator_handle_kind ) :: cgroup_alloc
S-22        type(omp_alloctrait),parameter       :: cgroup_traits(1)= &
S-23                              [omp_alloctrait(omp_atk_access,omp_atv_cgroup)]
S-24
S-25        do i=1,N; xbuf(i)=0; end do
S-26
S-27     !*** CASE 1: USING ALLOCATE DIRECTIVE ***!
S-28
S-29        !! uses predefined allocator omp_cgroup_mem_alloc
S-30
S-31        !$omp target uses_allocators(omp_cgroup_mem_alloc)
S-32        !$omp teams  reduction(+:xbuf) thread_limit(N) &
S-33        !$omp&       allocate(omp_cgroup_mem_alloc:xbuf) num_teams(4)
S-34
S-35            !$omp parallel do
S-36            do i = 1,N
S-37               xbuf(i) = xbuf(i) + calc(i, omp_get_team_num())
S-38            enddo
S-39
S-40        !$omp end teams
S-41        !$omp end target
S-42
S-43        sum = 0
S-44        !$omp parallel do reduction(+:sum)
S-45        do i = 1,N
S-46            sum = sum + xbuf(i)
S-47        enddo
S-48        if(sum == 3*(N+1)*N) print*, "PASSED 1 of 3"
S-49
S-50     !*** CASE 2: ***!
S-51
S-52        do i=1,N; xbuf(i)=0; end do
S-53
S-54        cgroup_alloc = omp_null_allocator
S-55
S-56        !! uses custom allocator with specified traits
S-57        !$omp  target uses_allocators(traits(cgroup_traits): cgroup_alloc)
S-58        !$omp  teams  reduction(+:xbuf) thread_limit(N) &
S-59        !$omp&        allocate(cgroup_alloc:xbuf) num_teams(4)
S-60
S-61            !$omp parallel do
S-62            do i = 1,N
```

```
S-63                xbuf(i) = xbuf(i) + calc(i,omp_get_team_num())
S-64            enddo
S-65
S-66        !$omp end teams
S-67        !$omp end target
S-68
S-69        sum = 0
S-70        !$omp parallel do reduction(+:sum)
S-71        do i = 1,N
S-72            sum = sum + xbuf(i)
S-73        enddo
S-74        if(sum == 3*(N+1)*N) print*, "PASSED 2 of 3"
S-75
S-76    !*** CASE 3: ***!
S-77
S-78        do i=1,N; xbuf(i)=0; end do
S-79
S-80        cgroup_alloc = omp_init_allocator(omp_default_mem_space, 1, &
S-81                                          cgroup_traits)
S-82
S-83        !! WARNING: uses custom allocator but with DEFAULT traits
S-84        !$omp  target uses_allocators(cgroup_alloc)
S-85        !$omp  teams  reduction(+:xbuf) thread_limit(N) &
S-86        !$omp&        allocate(cgroup_alloc:xbuf) num_teams(4)
S-87
S-88            !$omp parallel do
S-89            do i = 1,N
S-90                xbuf(i) = xbuf(i) + calc(i,omp_get_team_num())
S-91            enddo
S-92
S-93        !$omp end teams
S-94        !$omp end target
S-95
S-96        call omp_destroy_allocator(cgroup_alloc)
S-97
S-98        sum = 0
S-99        !$omp parallel do reduction(+:sum)
S-100       do i = 1,N
S-101           sum = sum + xbuf(i)
S-102       enddo
S-103       if(sum == 3*(N+1)*N) print*, "PASSED 3 of 3"
S-104
S-105   end program main
```

Fortran

The following example shows how to make an allocator available in a **target** region without specifying a **uses_allocators** clause.

In CASE 1, the predefined **omp_cgroup_mem_alloc** allocator is used in the **target** region as in CASE 1 of the previous example, but without specifying a **uses_allocators** clause. This is accomplished by specifying the **requires** directive with a **dynamic_allocators** clause in the same compilation unit, to remove restrictions on allocator usage in **target** regions.

CASE 2 also uses the **dynamic_allocators** clause to remove allocator restrictions in **target** regions. Here, an allocator is initialized by calling the **omp_init_allocator** routine in the **target** region. The allocator is then used for the allocations of array *xbuf* in an **allocate** clause of the **target teams** construct for each team and destroyed after its use. The use of separate **target** regions is needed here since no statement is allowed between a **target** directive and its nested **teams** construct.

C / C++

Example allocators.6.c (**omp_5.2**)

```
S-1   #include <omp.h>
S-2   #include <stdio.h>
S-3
S-4   #pragma omp requires dynamic_allocators
S-5
S-6   int calc(int i, int j) { return i*j;}
S-7   #pragma omp declare target(calc)
S-8
S-9   int main()
S-10  {
S-11      #define N 256
S-12      int sum;
S-13      int xbuf[N];
S-14
S-15      static omp_allocator_handle_t cgroup_alloc;
S-16      #pragma omp declare target(cgroup_alloc)
S-17      const omp_alloctrait_t cgroup_traits[1] =
S-18                              {{omp_atk_access, omp_atv_cgroup}};
S-19
S-20  /*** CASE 1: ***/
S-21
S-22      for (int i = 0; i < N; i++) { xbuf[i] = 0;}
S-23
S-24      // uses predefined allocator, no need to declare it in uses_allocators
S-25      #pragma omp target teams reduction(+:xbuf) thread_limit(N) \
S-26                          allocate(omp_cgroup_mem_alloc:xbuf) num_teams(4)
S-27      {
S-28          #pragma omp parallel for
S-29          for (int i = 0; i < N; i++) {
S-30              xbuf[i] += calc(i,omp_get_team_num());
S-31          }
S-32      }
```

```
S-33
S-34        sum = 0;
S-35        #pragma omp parallel for reduction(+:sum)
S-36        for (int i = 0; i < N; i++) {
S-37            sum += xbuf[i];
S-38        }
S-39        if(sum == 3*(N-1)*N) printf("PASSED 1 of 2\n");
S-40
S-41
S-42    /*** CASE 2: ***/
S-43
S-44        for (int i = 0; i < N; i++) { xbuf[i] = 0; }
S-45
S-46        // initializes the allocator in target region
S-47        #pragma omp target
S-48            cgroup_alloc = omp_init_allocator(
S-49                            omp_default_mem_space, 1, cgroup_traits);
S-50
S-51        // uses the initialized allocator
S-52        #pragma omp target
S-53        #pragma omp teams reduction(+:xbuf) thread_limit(N) \
S-54                          allocate(cgroup_alloc:xbuf) num_teams(4)
S-55        {
S-56            #pragma omp parallel for
S-57            for (int i = 0; i < N; i++) {
S-58                xbuf[i] += calc(i,omp_get_team_num());
S-59            }
S-60        }
S-61
S-62        // destroys the allocator after its use
S-63        #pragma omp target
S-64            omp_destroy_allocator(cgroup_alloc);
S-65
S-66        sum = 0;
S-67        #pragma omp parallel for reduction(+:sum)
S-68        for (int i = 0; i < N; i++) {
S-69            sum += xbuf[i];
S-70        }
S-71        if(sum == 3*(N-1)*N) printf("PASSED 2 of 2\n");
S-72
S-73        return 0;
S-74    }
```

C / C++

Example allocators.6.f90 (**omp_5.2**)

```
S-1   module functions
S-2   contains
S-3      function calc(i,j)  result(ii)
S-4        implicit none
S-5        integer :: i,j,ii
S-6        !$omp declare target(calc)
S-7
S-8        ii = i*j
S-9      end function
S-10  end module
S-11
S-12  program main
S-13
S-14      use omp_lib
S-15      use functions
S-16      implicit none
S-17      integer, parameter :: N=256
S-18      integer :: sum, i
S-19      integer :: xbuf(N)
S-20
S-21      !$omp requires dynamic_allocators
S-22
S-23      integer(omp_allocator_handle_kind),save :: cgroup_alloc
S-24      !$omp declare target(cgroup_alloc)
S-25      type(omp_alloctrait),parameter :: cgroup_traits(1)= &
S-26                           [omp_alloctrait(omp_atk_access,omp_atv_cgroup)]
S-27
S-28  !*** CASE 1: ***!
S-29
S-30      do i=1,N; xbuf(i)=0; end do
S-31
S-32      !! uses predefined allocator, no need to declare it in uses_allocators
S-33      !$omp  target teams reduction(+:xbuf) thread_limit(N) &
S-34      !$omp&               allocate(omp_cgroup_mem_alloc:xbuf) num_teams(4)
S-35
S-36          !$omp parallel do
S-37          do i = 1,N
S-38             xbuf(i) = xbuf(i) + calc(i,omp_get_team_num())
S-39          enddo
S-40
S-41      !$omp end target teams
S-42
S-43      sum = 0
S-44      !$omp parallel do reduction(+:sum)
```

```
S-45        do i = 1,N
S-46            sum = sum + xbuf(i)
S-47        enddo
S-48        if(sum == 3*(N+1)*N) print*, "PASSED 1 of 2"
S-49
S-50    !*** CASE 2: ***!
S-51
S-52        do i=1,N; xbuf(i)=0; end do
S-53
S-54        !! initializes allocator in the target region
S-55        !$omp target
S-56            cgroup_alloc = omp_init_allocator(omp_default_mem_space, 1, &
S-57                                              cgroup_traits)
S-58        !$omp end target
S-59
S-60        !! uses the initialized allocator
S-61        !$omp  target
S-62        !$omp  teams reduction(+:xbuf) thread_limit(N) &
S-63        !$omp&       allocate(cgroup_alloc:xbuf) num_teams(4)
S-64
S-65            !$omp parallel do
S-66            do i = 1,N
S-67               xbuf(i) = xbuf(i) + calc(i,omp_get_team_num())
S-68            enddo
S-69
S-70        !$omp end teams
S-71        !$omp end target
S-72
S-73        !! destroys the allocator after its use
S-74        !$omp target
S-75            call omp_destroy_allocator(cgroup_alloc)
S-76        !$omp end target
S-77
S-78        sum = 0
S-79        !$omp parallel do reduction(+:sum)
S-80        do i = 1,N
S-81            sum = sum + xbuf(i)
S-82        enddo
S-83        if(sum == 3*(N+1)*N) print*, "PASSED 2 of 2"
S-84
S-85    end program main
```

Fortran

Fortran

11.3 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 (**pre_omp_3.0**)

```
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

12 Program Control

Basic concepts and mechanisms for directing and controlling a program compilation and execution are provided in this introduction and illustrated in subsequent examples.

CONDITIONAL COMPILATION and EXECUTION

Conditional compilation can be performed with conventional **`#ifdef`** directives in C, C++, and Fortran, and additionally with OpenMP sentinel (**`!$`**) in Fortran. The **`if`** clause on some directives 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. The **`metadirective`** and **`declare variant`** directives provide conditional selection of directives and routines for compilation (and use), respectively. The **`assume`** and **`requires`** directives provide invariants for optimizing compilation, and essential features for compilation and correct execution, respectively.

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 a **`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 execution 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. Initial ICV values are reported by the runtime if the **OMP_DISPLAY_ENV** environment variable has been set to `TRUE` or `VERBOSE`.

NESTED CONSTRUCTS

Certain combinations of nested constructs are permitted, giving rise to *combined* constructs consisting of two or more directives. 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. For example, 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, unless the nested target is a reverse offload.

The **parallel** construct can be nested, as well as the **task** construct. The parallel execution in the nested **parallel** construct(s) is controlled by the **OMP_MAX_ACTIVE_LEVELS** environment variable, and the **omp_set_max_active_levels** routine. Use the **omp_get_max_active_levels** routine to determine the maximum levels provided by an implementation. As of OpenMP 5.0, use of the **OMP_NESTED** environment variable and the **omp_set_nested** routine has been deprecated.

More details on nesting can be found in the *Nesting of Regions* of the *Directives* chapter in the OpenMP Specifications document.

12.1 Assumption Directives

Assumption directives provide additional information about the expected properties of the program that may be used by an implementation for optimization. Ignoring this information should not alter the behavior of the program.

The C/C++ example shows the use of delimited scope (Case 1) and block-associated (Case 2) assumption directives. A similar effect is shown for Fortran where the **assumes** directive is used in the module (Case 1) and the block-associated directive uses an **end assume** termination (Case 2). The function *fun* is annotated with the **no_parallelism** clause, using the **begin assumes** (C) or **assumes** (Fortran) directive, to indicate that no implicit/explicit tasks are generated and no SIMD constructs are encountered during execution of the function. If the function *fun* contains

task-generating or SIMD constructs then the behavior would be undefined. The block-associated **assume** directive is used to indicate that *N* is a multiple of 8 and will always be equal to or greater than 1. This information, if used for optimization, could eliminate additional checks.

C / C++

Example assumption.1.c (**omp_5.1**)

```
S-1   #include <stdio.h>
S-2   #include <stdlib.h>
S-3
S-4   #pragma omp declare target
S-5   int N;
S-6   #pragma omp end declare target
S-7
S-8   // Case 1: Delimited scope
S-9   #pragma omp begin assumes no_parallelism
S-10  extern void fun(int *A);
S-11  #pragma omp end assumes
S-12
S-13  int main() {
S-14      int *A, *B;
S-15      N = (rand() % 5 + 1) * 16;
S-16      A = (int *) malloc(sizeof(int) * N);
S-17      B = (int *) malloc(sizeof(int) * N);
S-18
S-19      for(int i = 0; i < N; i++){
S-20          A[i] = 0;
S-21          B[i] = i;
S-22      }
S-23
S-24      #pragma omp target teams distribute parallel for map(tofrom: A[0:N])
S-25      for(int i = 0; i < N; i++){
S-26        fun(A);
S-27      }
S-28
S-29  // Case 2: Block associated
S-30      #pragma omp assume holds (N % 8 == 0 && N > 0)
S-31      #pragma omp simd
S-32      for (int i = 0; i < N; ++i){
S-33        A[i] += B[i];
S-34      }
S-35
S-36      return 0;
S-37  }
```

C / C++

Fortran

Example assumption.1.f90 (`omp_5.1`)

```
module m
  !$omp assumes no_parallelism
  interface
    subroutine fun(A, i)
      implicit none
      integer :: A(*),i

    end subroutine
  end interface

end module

program main
   use m
   implicit none
   integer,allocatable :: A(:), B(:)
   integer             :: i, N
   real                :: rand_no

   call random_number(rand_no)   !! create random,
   N = (int(rand_no*5)+1)*16     !! runtime number multiple of 16

   allocate(A(N),B(N))           !! alloc space & initialize
   do i = 1, N
      A(i) = 0; B(i) = i
   end do

!! Case 1: Delimited scope, see module interface
   !$omp target teams distribute parallel do map(tofrom: A)
   do i = 1, N
     call fun(A,i)
   end do

!! Case 2: Block associated
   !$omp assume holds (8*(N/8) == N .and. N>0)  !! N is multiple of 8
   !$omp simd
   do i = 1, N
     A(i) = A(i) + B(i)
   end do
   !$omp end assume

end program
```

Fortran

In the following example the **no_openmp** and **no_parallelism** assumption clauses are used. The **no_openmp** clause is shorthand for the **no_openmp_contructs** and **no_openmp_routines** clauses.

In Case 1 the **assume** directive with the **no_openmp** clause is applied to an external function call *init*. Independent of the compiler's ability to derive necessary information about *init*, the **assume** directive guarantees the absence of OpenMP constructs or OpenMP runtime calls so that the compiler may manage hardware and the runtime in a more optimal manner.

In Case 2, the **assume** directive with **no_parallelism** is nested inside the **target teams loop** directive. By providing the information that no other OpenMP parallelism generating constructs are going to be encountered in the function, the implementation of *element_transform* may have an opportunity to optimize the code in the **loop** construct, which may now be implemented using all additional threads available or via some other concurrency mechanism.

C / C++

Example assumption.2.c (**omp_6.0**)

```
S-1   #include <stdio.h>
S-2   #define N 5
S-3
S-4   void init(int *arr, int len);
S-5   int element_transform(int a);
S-6
S-7   int main() {
S-8     int arr[N], arr_bang[N];
S-9
S-10  //Case 1: Use in sequential code
S-11    #pragma omp assume no_openmp
S-12    {
S-13      init(arr,N);
S-14    }
S-15
S-16  //Case 2: Use inside openmp construct
S-17    #pragma omp target teams loop map(to: arr)  map(from: arr_bang)
S-18    for(int i = 0; i < N; i++) {
S-19      #pragma omp assume no_parallelism
S-20      {
S-21         arr_bang[i] = element_transform(arr[i]);
S-22      }
S-23    }
S-24    printf("%d, %d\n", arr_bang[0], arr_bang[N-1]);
S-25
S-26    return 0;
S-27  }
```

C / C++

Fortran

Example assumption.2.f90 (`omp_6.0`)

```
module mm
  interface
    subroutine init(arr, n)
      integer :: arr(*)
      integer :: n
    end subroutine
    function element_transform(a) result(r)
      !$omp declare target
      integer :: a, r
    end function
  end interface
end module

program main
   use mm
   integer, parameter :: N=5
   integer  :: arr(N), arr_bang(N)

!!Case 1: Use in sequential code
   !$omp assume no_openmp
     call init(arr,N)
   !$omp end assume

!!Case 2: Use inside openmp construct
   !$omp target teams loop map(to: arr) map(from: arr_bang)
   do i=1,N
      !$omp assume no_parallelism
        arr_bang(i) = element_transform(arr(i))
      !$omp end assume
   enddo

   print *, arr_bang(1), arr_bang(N)

end program main
```

Fortran

12.2 Conditional Compilation

C / C++

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

Example cond_comp.1.c (**pre_omp_3.0**)

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

C / C++

Fortran

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

Example cond_comp.1.f (**pre_omp_3.0**)

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

Fortran

12.3 Internal Control Variables (ICVs)

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

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

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

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

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

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

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

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

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

C / C++

Example icv.1.c (`pre_omp_3.0`)

```
S-1   #include <stdio.h>
S-2   #include <omp.h>
S-3
S-4   int main (void)
S-5   {
S-6     omp_set_nested(1);
S-7     omp_set_max_active_levels(8);
S-8     omp_set_dynamic(0);
S-9     omp_set_num_threads(2);
S-10    #pragma omp parallel
S-11      {
S-12        omp_set_num_threads(3);
S-13
S-14        #pragma omp parallel
S-15          {
S-16            omp_set_num_threads(4);
S-17            #pragma omp single
S-18              {
S-19                   // The following should print:
S-20                   // Inner: max_act_lev=8, num_thds=3, max_thds=4
S-21                   // Inner: max_act_lev=8, num_thds=3, max_thds=4
S-22                printf ("Inner: max_act_lev=%d, num_thds=%d, max_thds=%d\n",
S-23                omp_get_max_active_levels(), omp_get_num_threads(),
S-24                omp_get_max_threads());
S-25              }
S-26          }
S-27
S-28        #pragma omp barrier
S-29        #pragma omp single
S-30          {
S-31                   // The following should print:
S-32                   // Outer: max_act_lev=8, num_thds=2, max_thds=3
S-33            printf ("Outer: max_act_lev=%d, num_thds=%d, max_thds=%d\n",
S-34                    omp_get_max_active_levels(), omp_get_num_threads(),
S-35                    omp_get_max_threads());
S-36          }
S-37      }
S-38      return 0;
S-39  }
```

C / C++

Fortran

Example icv.1.f (`pre_omp_3.0`)

```
      program icv
      use omp_lib

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

!$omp parallel
      call omp_set_num_threads(3)

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

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

Fortran

12.3.1 `num_threads` Clause with a List

Prior to OpenMP 6.0, only a single argument can be specified in the **`num_threads`** clause of a **`parallel`** construct. In this case, the clause argument is used as the requested team size for that **`parallel`** region only and does not affect the value of the *nthreads-var* ICV in any generated implicit tasks for nested **`parallel`** regions. That value is instead inherited from the value of the *nthreads-var* ICV in the task that encountered the **`parallel`** construct, stripping away the first integer, if the value of that ICV is a list of multiple integers.

In OpenMP 6.0, the **`num_threads`** clause permits more than one argument. In this case, the first argument is still used as the requested team size for the **`parallel`** region. The difference is the *nthreads-var* ICVs of the generated implicit tasks are set to the list of values given by the remaining clause arguments, rather than inheriting the value of the encountering task's *nthreads-var* ICV. Consequentially, a **`num_threads`** clause with an argument list may be used to control not only the team size for a given **`parallel`** region, but also the requested team size of any nested **`parallel`** regions.

The following example illustrates the effect of the **`num_threads`** clause for nested **`parallel`** regions. The program starts with the environment variable **`OMP_NUM_THREADS`** set to *`"4,5,6"`*, which initializes the *nthreads-var* ICV of the initial task to the list {`4,5,6`}. Case 1 shows how this ICV is used to control the requested team size for a nest of three **`parallel`** regions. As indicated from the comments, with each successive nesting level the *nthreads-var* ICV inherits all but the first integer in the *nthreads-var* ICV of the task that encounters the **`parallel`** construct. This pattern continues until the *nthreads-var* ICV contains only a single integer, at which point that value persists for any further nesting levels. In Case 2, a **`num_threads(`***`8`***`)`** clause appears on the outermost **`parallel`** construct. This only has the effect of altering the requested team size for that **`parallel`** region. Note that the value of the *nthreads-var* ICVs inside the **`parallel`** region are the same as for Case 1. In Case 3, the **`num_threads`** clause is specified with multiple arguments **`(`***`8,2`***`)`**. This sets the *nthreads-var* ICV value in each of the generated implicit tasks to {`2`}, in accordance with the inheritance rules for the *nthreads-var* ICV described above.

C / C++

Example icv.2.c (**`omp_6.0`**)

```
S-1   #include <stdio.h>
S-2   #include <omp.h>
S-3
S-4   void prn_info(int level)
S-5   {
S-6      #pragma omp masked
S-7         printf("LV%d: nthrs_next=%d\n",
S-8                level, omp_get_max_threads());
S-9   }
S-10
S-11  // run with OMP_NUM_THREADS="4,5,6" OMP_MAX_ACTIVE_LEVELS=3
S-12  int main (void)
S-13  {
```

```
S-14        // nthreads-var: 4,5,6
S-15        // max-active-levels-var: 3
S-16
S-17        // Case 1
S-18        #pragma omp parallel // request 4 threads
S-19        {
S-20           prn_info(1);      // LV1: nthrs_next=5
S-21
S-22           // nthreads-var: 5,6
S-23           #pragma omp parallel // request 5 threads
S-24           {
S-25              prn_info(2);      // LV2: nthrs_next=6
S-26
S-27              // nthreads-var: 6
S-28              #pragma omp parallel // request 6 threads
S-29              {
S-30                 prn_info(3);      // LV3: nthrs_next=6
S-31
S-32                 // nthreads-var: 6
S-33              }
S-34           }
S-35        }
S-36
S-37        // Case 2
S-38        #pragma omp parallel num_threads(8)
S-39        {
S-40           prn_info(1);      // LV1: nthrs_next=5
S-41
S-42           // nthreads-var: 5,6
S-43           #pragma omp parallel // request 5 threads
S-44           {
S-45              prn_info(2);      // LV2: nthrs_next=6
S-46
S-47              // nthreads-var: 6
S-48              #pragma omp parallel // request 6 threads
S-49              {
S-50                 prn_info(3);      // LV3: nthrs_next=6
S-51
S-52                 // nthreads-var: 6
S-53              }
S-54           }
S-55        }
S-56
S-57        // Case 3
S-58        #pragma omp parallel num_threads(8,2)
S-59        {
S-60           prn_info(1);      // LV1: nthrs_next=2
```

```
S-61
S-62          // nthreads-var: 2
S-63          #pragma omp parallel // request 2 threads
S-64          {
S-65             prn_info(2);      // LV2: nthrs_next=2
S-66
S-67             // nthreads-var: 2
S-68             #pragma omp parallel // request 2 threads
S-69             {
S-70                prn_info(3);      // LV3: nthrs_next=2
S-71
S-72                // nthreads-var: 2
S-73             }
S-74          }
S-75       }
S-76
S-77       return 0;
S-78    }
```

C / C++

Fortran

1 *Example icv.2.f90* (`omp_6.0`)

```
S-1     subroutine prn_info(level)
S-2        use omp_lib, only : omp_get_max_threads
S-3        implicit none
S-4        integer level
S-5
S-6        !$omp masked
S-7           print 10, level, omp_get_max_threads()
S-8        !$omp end masked
S-9        10 format("LV",i0,": nthrs_next=",i0)
S-10    end subroutine
S-11
S-12    program main
S-13       implicit none
S-14
S-15       !! run with OMP_NUM_THREADS="4,5,6" OMP_MAX_ACTIVE_LEVELS=3
S-16       !! nthreads-var: 4,5,6
S-17       !! max-active-levels-var: 3
S-18
S-19       !! Case 1
S-20       !$omp parallel       ! request 4 threads
S-21          call prn_info(1)  ! LV1: nthrs_next=5
S-22
S-23          !! nthreads-var: 5,6
S-24          !$omp parallel       ! request 5 threads
```

```
S-25            call prn_info(2)  ! LV2: nthrs_next=6
S-26
S-27            !! nthreads-var: 6
S-28            !$omp parallel       ! request 6 threads
S-29               call prn_info(3)  ! LV3: nthrs_next=6
S-30
S-31               !! nthreads-var: 6
S-32            !$omp end parallel
S-33         !$omp end parallel
S-34      !$omp end parallel
S-35
S-36      !! Case 2
S-37      !$omp parallel num_threads(8)
S-38         call prn_info(1)  ! LV1: nthrs_next=5
S-39
S-40         !! nthreads-var: 5,6
S-41         !$omp parallel       ! request 5 threads
S-42            call prn_info(2)  ! LV2: nthrs_next=6
S-43
S-44            !! nthreads-var: 6
S-45            !$omp parallel       ! request 6 threads
S-46               call prn_info(3)  ! LV3: nthrs_next=6
S-47
S-48               !! nthreads-var: 6
S-49            !$omp end parallel
S-50         !$omp end parallel
S-51      !$omp end parallel
S-52
S-53      !! Case 3
S-54      !$omp parallel num_threads(8,2)
S-55         call prn_info(1)  ! LV1: nthrs_next=2
S-56
S-57         !! nthreads-var: 2
S-58         !$omp parallel       ! request 2 threads
S-59            call prn_info(2)  ! LV2: nthrs_next=2
S-60
S-61            !! nthreads-var: 2
S-62            !$omp parallel       ! request 2 threads
S-63               call prn_info(3)  ! LV3: nthrs_next=2
S-64
S-65               !! nthreads-var: 2
S-66            !$omp end parallel
S-67         !$omp end parallel
S-68      !$omp end parallel
S-69
S-70   end program
```

Fortran

12.4 Placement of `flush`, `barrier`, `taskwait` and `taskyield` Directives

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

C / C++

Example standalone.1.c (`omp_3.1`)

```
S-1   void standalone_wrong()
S-2   {
S-3     int a = 1;
S-4
S-5           if (a != 0)
S-6     #pragma omp flush(a)
S-7   /* incorrect as flush cannot be immediate substatement
S-8      of if statement */
S-9
S-10          if (a != 0)
S-11    #pragma omp barrier
S-12  /* incorrect as barrier cannot be immediate substatement
S-13     of if statement */
S-14
S-15          if (a!=0)
S-16    #pragma omp taskyield
S-17  /* incorrect as taskyield cannot be immediate substatement of if statement
S-18  */
S-19
S-20          if (a != 0)
S-21    #pragma omp taskwait
S-22  /* incorrect as taskwait cannot be immediate substatement
S-23     of if statement */
S-24
S-25  }
```

C / C++

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

Fortran

Example standalone.1.f90 (`omp_3.1`)

```
S-1   SUBROUTINE STANDALONE_WRONG()
S-2     INTEGER  A
S-3
S-4     A = 1
S-5
S-6     ! the FLUSH directive must not be the action statement
S-7     ! in an IF statement
S-8     IF (A .NE. 0) !$OMP FLUSH(A)
S-9
S-10    ! the BARRIER directive must not be the action statement
S-11    ! in an IF statement
S-12    IF (A .NE. 0) !$OMP BARRIER
S-13
S-14    ! the TASKWAIT directive must not be the action statement
S-15    ! in an IF statement
S-16    IF (A .NE. 0) !$OMP TASKWAIT
S-17
S-18    ! the TASKYIELD directive must not be the action statement
S-19    ! in an IF statement
S-20    IF (A .NE. 0) !$OMP TASKYIELD
S-21
S-22    GOTO 100
S-23
S-24    ! the FLUSH directive must not be a labeled branch target
S-25    ! statement
S-26    100 !$OMP FLUSH(A)
S-27    GOTO 200
S-28
S-29    ! the BARRIER directive must not be a labeled branch target
S-30    ! statement
S-31    200 !$OMP BARRIER
S-32    GOTO 300
S-33
S-34    ! the TASKWAIT directive must not be a labeled branch target
S-35    ! statement
S-36    300 !$OMP TASKWAIT
S-37    GOTO 400
S-38
S-39    ! the TASKYIELD directive must not be a labeled branch target
S-40    ! statement
S-41    400 !$OMP TASKYIELD
S-42
S-43  END SUBROUTINE
```

Fortran

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

C / C++

Example standalone.2.c (**omp_3.1**)

```
S-1   void standalone_ok()
S-2   {
S-3     int a = 1;
S-4
S-5     #pragma omp parallel
S-6     {
S-7        if (a != 0) {
S-8     #pragma omp flush(a)
S-9        }
S-10       if (a != 0) {
S-11    #pragma omp barrier
S-12       }
S-13       if (a != 0) {
S-14    #pragma omp taskwait
S-15       }
S-16          if (a != 0) {
S-17    #pragma omp taskyield
S-18          }
S-19    }
S-20  }
```

C / C++

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

Fortran

Example standalone.2.f90 (**omp_3.1**)

```
S-1   SUBROUTINE STANDALONE_OK()
S-2     INTEGER  A
S-3     A = 1
S-4     IF (A .NE. 0) THEN
S-5       !$OMP FLUSH(A)
S-6     ENDIF
S-7     IF (A .NE. 0) THEN
S-8       !$OMP BARRIER
S-9     ENDIF
S-10    IF (A .NE. 0) THEN
S-11      !$OMP TASKWAIT
S-12    ENDIF
S-13    IF (A .NE. 0) THEN
```

```
      !$OMP TASKYIELD
    ENDIF
    GOTO 100
    100 CONTINUE
    !$OMP FLUSH(A)
    GOTO 200
    200 CONTINUE
    !$OMP BARRIER
    GOTO 300
    300 CONTINUE
    !$OMP TASKWAIT
    GOTO 400
    400 CONTINUE
    !$OMP TASKYIELD
END SUBROUTINE
```

Fortran

12.5 Cancellation Constructs

The examples in this section show how the **cancel** directive can be used to terminate an OpenMP region. Cancellation of the binding region is activated only if the *cancel-var* ICV is true, in which case the **cancel** construct (except **taskgroup**) causes the encountering **task** to continue execution at the end of the binding. If the *cancel-var* ICV is false, the **cancel** construct is ignored.

In the following example 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 primary 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++

Example cancellation.1.cpp (**omp_4.0**)

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

#define N 10000

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

void example() {
```

```
S-12        std::exception *ex = NULL;
S-13    #pragma omp parallel shared(ex)
S-14        {
S-15    #pragma omp for
S-16            for (int i = 0; i < N; i++) {
S-17                // no 'if' that prevents compiler optimizations
S-18                try {
S-19                    causes_an_exception();
S-20                }
S-21                catch (std::exception *e) {
S-22                    // still must remember exception for later handling
S-23    #pragma omp atomic write
S-24                    ex = e;
S-25                    // cancel worksharing construct
S-26    #pragma omp cancel for
S-27                }
S-28            }
S-29            // if an exception has been raised, cancel parallel region
S-30            if (ex) {
S-31    #pragma omp cancel parallel
S-32            }
S-33            phase_1();
S-34    #pragma omp barrier
S-35            phase_2();
S-36        }
S-37        // continue here if an exception has been thrown in
S-38        // the worksharing loop
S-39        if (ex) {
S-40            // handle exception stored in ex
S-41        }
S-42    }
```

C++

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

Fortran

5 *Example cancellation.1.f90* (**omp_4.0**)

```
S-1     subroutine example(n, dim)
S-2       integer, intent(in) :: n, dim(n)
S-3       integer :: i, s, err
S-4       real, allocatable :: B(:)
S-5       err = 0
```

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

Fortran

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

C / C++

Example cancellation.2.c (**omp_5.1**)

```
#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)
```

```
S-16                {
S-17                    binary_tree_t *found_left;
S-18                    found_left = search_tree(tree->left, value, level + 1);
S-19                    if (found_left) {
S-20    #pragma omp atomic write
S-21                        found = found_left;
S-22    #pragma omp cancel taskgroup
S-23                    }
S-24                }
S-25    #pragma omp task shared(found) if(level < 10)
S-26                {
S-27                    binary_tree_t *found_right;
S-28                    found_right = search_tree(tree->right, value, level + 1);
S-29                    if (found_right) {
S-30    #pragma omp atomic write
S-31                        found = found_right;
S-32    #pragma omp cancel taskgroup
S-33                    }
S-34                }
S-35    #pragma omp taskwait
S-36            }
S-37        }
S-38        return found;
S-39    }
S-40
S-41    binary_tree_t *search_tree_parallel(binary_tree_t *tree, int value) {
S-42        binary_tree_t *found = NULL;
S-43    #pragma omp parallel shared(found, tree, value)
S-44        {
S-45    #pragma omp masked
S-46            {
S-47    #pragma omp taskgroup
S-48                {
S-49                    found = search_tree(tree, value, 0);
S-50                }
S-51            }
S-52        }
S-53        return found;
S-54    }
```

C / C++

The following is the equivalent parallel search example in Fortran. The code uses the **atomic write** directive for atomically updating pointer variables – a feature defined in OpenMP 6.0. For earlier versions of OpenMP, the **critical** directive could be used instead.

Example cancellation.2.f90 (`omp_6.0`)

```
S-1   module parallel_search
S-2     type binary_tree
S-3       integer :: value
S-4       type(binary_tree), pointer :: right
S-5       type(binary_tree), pointer :: left
S-6     end type
S-7
S-8   contains
S-9     recursive function search_tree(tree, value, level) result(found)
S-10      type(binary_tree), intent(in), pointer :: tree
S-11      integer, intent(in) :: value, level
S-12      type(binary_tree), pointer :: found
S-13      type(binary_tree), pointer :: found_left, found_right
S-14
S-15      found => NULL()
S-16      if (associated(tree)) then
S-17        if (tree%value .eq. value) then
S-18          found => tree
S-19        else
S-20  !$omp task shared(found) if(level<10)
S-21          found_left => search_tree(tree%left, value, level+1)
S-22          if (associated(found_left)) then
S-23  !$omp atomic write
S-24            found => found_left
S-25  !$omp end atomic
S-26
S-27  !$omp cancel taskgroup
S-28          endif
S-29  !$omp end task
S-30
S-31  !$omp task shared(found) if(level<10)
S-32          found_right => search_tree(tree%right, value, level+1)
S-33          if (associated(found_right)) then
S-34  !$omp atomic write
S-35            found => found_right
S-36  !$omp end atomic
S-37
S-38  !$omp cancel taskgroup
S-39          endif
S-40  !$omp end task
S-41
S-42  !$omp taskwait
S-43        endif
S-44      endif
```

```
S-45      end function
S-46
S-47      subroutine search_tree_parallel(tree, value, found)
S-48        type(binary_tree), intent(in), pointer :: tree
S-49        integer, intent(in) :: value
S-50        type(binary_tree), pointer :: found
S-51
S-52        found => NULL()
S-53    !$omp parallel shared(found, tree, value)
S-54    !$omp masked
S-55    !$omp taskgroup
S-56        found => search_tree(tree, value, 0)
S-57    !$omp end taskgroup
S-58    !$omp end masked
S-59    !$omp end parallel
S-60      end subroutine
S-61
S-62    end module parallel_search
```

Fortran

12.6 `requires` Directive

The declarative **`requires`** directive can be used to specify features that an implementation must provide to compile and execute correctly.

In the following example the **`unified_shared_memory`** clause of the **`requires`** directive ensures that the host and all devices accessible through OpenMP provide a *unified address* space for memory that is shared by all devices.

The example illustrates the use of the **`requires`** directive specifying *unified shared memory* in file scope, before any device directives or device routines. No **`map`** clause is needed for the `p` structure on the device (and its address `&p`, for the C++ code, is the same address on the host and device). However, scalar variables referenced within the **`target`** construct still have a default data-sharing attribute of **`firstprivate`**. The `q` scalar is incremented on the device, and its change is not updated on the host.

C++

1 *Example requires.1.cpp* (`omp_5.0`)

```
S-1   #include <iostream>
S-2   using namespace std;
S-3
S-4   #pragma omp requires unified_shared_memory
S-5
S-6   typedef struct mypoints
S-7   {
S-8      double res;
S-9      double data[500];
S-10  } mypoints_t;
S-11
S-12  void do_something_with_p(mypoints_t *p, int q);
S-13
S-14  int main()
S-15  {
S-16    mypoints_t p;
S-17    int q=0;
S-18
S-19    #pragma omp target // no map clauses needed
S-20    {                  // q is firstprivate
S-21       q++;
S-22       do_something_with_p(&p,q);
S-23    }
S-24    cout<< p.res << " " << q << endl;  // output 1 0
S-25    return 0;
S-26  }
S-27  void do_something_with_p(mypoints_t *p, int q)
S-28  {
S-29    p->res = q;
S-30    for(int i=0;i<sizeof(p->data)/sizeof(double);i++)
S-31        p->data[i]=q*i;
S-32  }
```

C++

Fortran

2 *Example requires.1.f90* (`omp_5.0`)

```
S-1   module data
S-2   !$omp requires unified_shared_memory
S-3     type,public :: mypoints
S-4        double precision :: res
S-5        double precision :: data(500)
S-6     end type
```

```
end module

program main
  use data
  type(mypoints) :: p
  integer        :: q=0

  !$omp target    !! no map clauses needed
     q = q + 1    !! q is firstprivate
     call do_something_with_p(p,q)
  !$omp end target

  write(*,'(f5.0,i5)') p%res, q    !! output 1.   0

end program

subroutine do_something_with_p(p,q)
  use data
  !$omp declare target
  type(mypoints) :: p
  integer        :: q

  p%res = q;
  do i=1,size(p%data)
     p%data(i)=q*i
  enddo

end subroutine
```

Fortran

12.7 Context-based Variant Selection

Certain directives, including **declare variant**, **begin declare variant**, and **metadirective** directives, specify function or directive variants for callsite or directive substitution. They use *context selectors* to specify the contexts in which the variant may be selected for substitution. A context selector specifies various *trait selectors*, grouped into *trait selector sets*. A trait selector, for a given trait selector set, identifies a corresponding trait (and, in some cases, its trait properties) that may or may not be active in an *OpenMP context*. A context selector is considered to be *compatible* with a given OpenMP context if all traits and trait properties corresponding to trait selectors are active in that context.

Each context selector is a comma-separated list of trait selector sets and each trait selector set has the form *trait-selector-name* **={** *trait-selector-list* **}**, where *trait-selector-list* is a comma-separated list of trait selectors. Some trait selectors may in turn specify one or more *trait properties*.

Additionally, a trait selector may optionally specify a *trait score* for explicit control over variant selection.

Consider this context selector: **construct={teams,parallel,for}, device={arch(nvptx)}, user={condition(***N>32***)}**.

The context selector specifies three distinct trait selector sets, a **construct** trait selector set, a **device** trait selector set, and a **user** trait selector set. The **construct** trait selector set specifies three trait selectors: **teams**, **parallel**, and **for**. The **device** trait selector set specifies one trait selector: **arch(nvptx)**. And the **user** trait selector set specifies one trait selector: **condition(***N>32***)**.

The **teams**, **parallel**, and **for** trait selectors respectively require that the *teams*, *parallel*, and *for* traits are active in the *construct* trait set of the OpenMP context (i.e., the **teams**, **parallel**, and **for** constructs are enclosing constructs that do not appear outside any enclosing **target** construct at the program point of interest). The **arch** trait selector specifies the **nvptx** trait property, requiring that *nvptx* is one of the supported architectures per the *arch* trait of the *device* trait set of the OpenMP context. Finally, the **condition** trait selector specifies the *N>32* expression as a trait property, requiring that *N>32* evaluates to *true* in the OpenMP context.

The remainder of this section presents examples that make use of context selectors for function and directive variant selection. Sections 12.7.1 and 12.7.2 cover cases where only one context selector is compatible. Section 12.7.3 covers cases where multiple compatible context selectors exist and a scoring algorithm determines which one of the variants is selected.

12.7.1 `declare variant` Directive

A **declare variant** directive specifies an alternate function, *function variant*, to be used in place of the *base function* when the trait within the **match** clause matches the OpenMP context at a given callsite. The base function follows the directive in the C and C++ languages. In Fortran, either a subroutine or function may be used as the base function, and the **declare variant** directive must be in the specification part of a subroutine or function (unless a *base-proc-name* modifier is used, as in the case of a procedure declaration statement). See the OpenMP 5.0 Specification for details on the modifier.

When multiple **declare variant** directives are used a function variant becomes a candidate for replacing the base function if the context at the base function call matches the traits of all selectors in the **match** clause. If there are multiple candidates, a score is assigned with rules for each of the selector traits. See Section 12.7.3 for details.

In the first example the *vxv()* function is called within a **parallel** region, a **target** region, and in a sequential part of the program. Two function variants, *p_vxv()* and *t_vxv()*, are defined for the first two regions by using **parallel** and **target** selectors (within the *construct* trait set) in a **match** clause. The *p_vxv()* function variant includes a **for** construct (**do** construct for Fortran) for the **parallel** region, while *t_vxv()* includes a **distribute simd**

construct for the **target** region. The *t_vxv()* function is explicitly compiled for the device using a declare target directive.

Since the two **declare variant** directives have no selectors that match traits for the context of the base function call in the sequential part of the program, the base *vxv()* function is used there, as expected. (The vectors in the *p_vxv* and *t_vxv* functions have been multiplied by 3 and 2, respectively, for checking the validity of the replacement. Normally the purpose of a function variant is to produce the same results by a different method.)

C / C++

Example declare_variant.1.c (**omp_5.1**)

```
S-1   #define N 100
S-2   #include <stdio.h>
S-3   #include <omp.h>
S-4
S-5   void p_vxv(int *v1,int *v2,int *v3,int n);
S-6   void t_vxv(int *v1,int *v2,int *v3,int n);
S-7
S-8   #pragma omp declare variant( p_vxv ) match( construct={parallel} )
S-9   #pragma omp declare variant( t_vxv ) match( construct={target}   )
S-10  void vxv(int *v1,int *v2,int *v3,int n)     // base function
S-11  {
S-12     for (int i= 0; i< n; i++)  v3[i] = v1[i] * v2[i];
S-13  }
S-14
S-15  void p_vxv(int *v1,int *v2,int *v3,int n)   // function variant
S-16  {
S-17     #pragma omp for
S-18     for (int i= 0; i< n; i++)  v3[i] = v1[i] * v2[i]*3;
S-19  }
S-20
S-21  #pragma omp begin declare target
S-22  void t_vxv(int *v1,int *v2,int *v3,int n)   // function variant
S-23  {
S-24     #pragma omp distribute simd
S-25     for (int i= 0; i< n; i++)  v3[i] = v1[i] * v2[i]*2;
S-26  }
S-27  #pragma omp end declare target
S-28
S-29  int main()
S-30  {
S-31     int v1[N], v2[N], v3[N];
S-32     for(int i=0; i<N; i++){ v1[i]=(i+1); v2[i]=-(i+1); v3[i]=0; }   //init
S-33
S-34     #pragma omp parallel
S-35     {
```

```
S-36          vxv(v1,v2,v3,N);
S-37       }
S-38       printf(" %d  %d\n",v3[0],v3[N-1]); //from p_vxv --  output: -3  -30000
S-39
S-40       #pragma omp target teams map(to: v1[:N],v2[:N]) map(from: v3[:N])
S-41       {
S-42          vxv(v1,v2,v3,N);
S-43       }
S-44       printf(" %d  %d\n",v3[0],v3[N-1]); //from t_vxv --  output: -2  -20000
S-45
S-46       vxv(v1,v2,v3,N);
S-47       printf(" %d  %d\n",v3[0],v3[N-1]); //from   vxv --  output: -1  -10000
S-48
S-49       return 0;
S-50    }
```

C / C++

Fortran

Example declare_variant.1.f90 (`omp_5.0`)

```
S-1     module subs
S-2       use omp_lib
S-3     contains
S-4        subroutine vxv(v1, v2, v3)              !! base function
S-5           integer,intent(in)  :: v1(:),v2(:)
S-6           integer,intent(out) :: v3(:)
S-7           integer             :: i,n
S-8           !$omp  declare variant( p_vxv ) match( construct={parallel} )
S-9           !$omp  declare variant( t_vxv ) match( construct={target}   )
S-10
S-11          n=size(v1)
S-12          do i = 1,n; v3(i) = v1(i) * v2(i); enddo
S-13
S-14       end subroutine
S-15
S-16       subroutine p_vxv(v1, v2, v3)             !! function variant
S-17          integer,intent(in)  :: v1(:),v2(:)
S-18          integer,intent(out) :: v3(:)
S-19          integer             :: i,n
S-20          n=size(v1)
S-21
S-22          !$omp do
S-23          do i = 1,n; v3(i) = v1(i) * v2(i) * 3; enddo
S-24
S-25       end subroutine
S-26
S-27       subroutine t_vxv(v1, v2, v3)             !! function variant
```

```
S-28          integer,intent(in)  :: v1(:),v2(:)
S-29          integer,intent(out) :: v3(:)
S-30          integer             :: i,n
S-31          !$omp declare target
S-32          n=size(v1)
S-33
S-34          !$omp distribute simd
S-35          do i = 1,n; v3(i) = v1(i) * v2(i) * 2; enddo
S-36
S-37       end subroutine
S-38
S-39    end module subs
S-40
S-41
S-42    program main
S-43       use omp_lib
S-44       use subs
S-45       integer,parameter :: N = 100
S-46       integer           :: v1(N), v2(N), v3(N)
S-47
S-48       do i= 1,N; v1(i)= i; v2(i)= -i; v3(i)= 0;  enddo  !! init
S-49
S-50       !$omp parallel
S-51          call vxv(v1,v2,v3)
S-52       !$omp end parallel
S-53       print *, v3(1),v3(N)    !! from p_vxv -- output: -3  -30000
S-54
S-55       !$omp target teams map(to: v1,v2) map(from: v3)
S-56          call vxv(v1,v2,v3)
S-57       !$omp end target teams
S-58       print *, v3(1),v3(N)    !! from t_vxv -- output: -2  -20000
S-59
S-60       call vxv(v1,v2,v3)
S-61       print *, v3(1),v3(N)    !! from   vxv -- output: -1  -10000
S-62
S-63    end program
```

Fortran

In this example, traits from the *device* set are used to select a function variant. In the **declare variant** directive, an **isa** trait selector specifies that if the implementation of the "core-avx512" instruction set is detected at compile time the *avx512_saxpy()* variant function is used for the call to *base_saxpy()*.

A compilation of *avx512_saxpy()* is aware of the AVX-512 instruction set that supports 512-bit vector extensions. Within *avx512_saxpy()*, the **parallel for simd** construct performs parallel execution, and takes advantage of 64-byte data alignment. When the

avx512_saxpy() function variant is not selected, the base *base_saxpy()* function variant containing only a basic **parallel for** construct is used for the call to *base_saxpy()*.

C / C++

Example declare_variant.2.c (**omp_5.0**)

```
#include <omp.h>

void   base_saxpy(int, float, float *, float *);
void avx512_saxpy(int, float, float *, float *);

#pragma omp declare variant( avx512_saxpy ) \
                       match( device={isa("core-avx512")} )
void base_saxpy(int n, float s, float *x, float *y)   // base function
{
   #pragma omp parallel for
   for(int i=0; i<n; i++) y[i] = s*x[i] + y[i];
}

void avx512_saxpy(int n, float s, float *x, float *y) //function variant
{
   //assume 64-byte alignment for AVX-512
   #pragma omp parallel for simd simdlen(16) aligned(x,y:64)
   for(int i=0; i<n; i++) y[i] = s*x[i] + y[i];
}

// Above may be in another file scope.

#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#define N 1000

int main()
{
   static float x[N],y[N] __attribute__ ((aligned(64)));
   float s=2.0;
                          // Check for 64-byte aligned
   if( ((intptr_t)y)%64 != 0 || ((intptr_t)x)%64 != 0 )
   { printf("ERROR: x|y not 64-Byte aligned\n"); exit(1); }

   for(int i=0; i<N; i++){ x[i]=i+1; y[i]=i+1; } // initialize

   base_saxpy(N,s,x,y);

   printf("y[0],y[N-1]: %5.0f %5.0f\n",y[0],y[N-1]);
   //output: y[0],y[N-1]: 3  3000
```

```
   return 0;
}
```

C / C++

Fortran

Example declare_variant.2.f90 (`omp_5.0`)

```
module subs
  use omp_lib
contains

   subroutine base_saxpy(s,x,y)                !! base function
      real,intent(inout) :: s,x(:),y(:)
     !$omp  declare variant( avx512_saxpy ) &
     !$omp&           match( device={isa("core-avx512")} )

      y = s*x + y

   end subroutine

   subroutine avx512_saxpy(s,x,y)                !! function variant
      real,intent(inout) :: s,x(:),y(:)
      integer            :: i,n
      n=size(x)
                              !!assume 64-byte alignment for AVX-512
      !$omp parallel do simd simdlen(16) aligned(x,y: 64)
      do i = 1,n
         y(i) = s*x(i) + y(i)
      end do

   end subroutine

end module subs


program main
   use omp_lib
   use subs

   integer, parameter :: N=1000, align=64
   real, allocatable  :: x(:),y(:)
   real               :: s = 2.0e0
   integer            :: i

   allocate(x(N),y(N))   !! Assumes allocation is 64-byte aligned
                         !! (using compiler options, or another
```

```
                              !! allocation method).

                              !! loc is non-standard, but found everywhere
                              !! remove these lines if not available
    if(modulo(loc(x),align) /= 0 .and. modulo(loc(y),align) /=0 ) then
       print*,"ERROR: x|y not 64-byte aligned"; stop
    endif

    do i=1,N  !! initialize
      x(i)=i
      y(i)=i
    end do

    call base_saxpy(s,x,y)

    write(*,'("y(1),y(N):",2f6.0)') y(1),y(N) !!output: y... 3. 3000.

    deallocate(x,y)

end program
```

Fortran

The **begin declare variant** with a paired **end declare variant** directive was introduced for C/C++ in the OpenMP 5.1 to allow nesting of declare variant directives. This example shows a practical situation where nested declare variant directives can be used to include the correct specialized user function based on the underlying vendor **isa** trait. The function name *my_fun()* is identical in all the header files and the version called will differ based on the calling context. The example assumes that either NVIDIA or AMD target devices are used.

C / C++

Example declare_variant.3.c (**omp_5.1**)

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

#ifdef _OPENMP
#pragma omp begin declare variant match(device={kind(nohost)})

  #pragma omp begin declare variant match(implementation={vendor(nvidia)})

    #pragma omp begin declare variant match(device={isa(sm_70)})
      #include "sm70/my_cuda_fun.h"   /* only included if isa is sm70 */
    #pragma omp end declare variant

```

```
S-16        #pragma omp begin declare variant match(device={isa(sm_80)})
S-17          #include "sm80/my_cuda_fun.h"   /* only included if isa is sm80 */
S-18        #pragma omp end declare variant
S-19
S-20      #pragma omp end declare variant
S-21
S-22      #pragma omp begin declare variant match(implementation={vendor(amd)})
S-23        #include "amdgpu/my_hip_fun.h" /* only included for AMD */
S-24      #pragma omp end declare variant
S-25
S-26    #pragma omp end declare variant
S-27
S-28    #pragma omp begin declare variant match(device={kind(host)})
S-29      #include "openmp_host/my_fun.h"
S-30    #pragma omp end declare variant
S-31    #else
S-32        #include "generic/my_fun.h"
S-33    #endif
S-34
S-35    #define N 64
S-36    double array[N];
S-37    #pragma omp target
S-38
S-39    int main() {
S-40       // Array initialization
S-41      for (int i = 0; i < N; ++i) {
S-42        array[i] = 0.0;
S-43      }
S-44
S-45    #pragma omp target map(tofrom: array[0:N])
S-46      for (int i = 0; i < N; ++i) {
S-47        array[i] = my_fun(i);
S-48      }
S-49      return 0;
S-50
S-51    }
```

C / C++

12.7.2 Metadirectives

A **metadirective** directive provides a mechanism to select a directive in a **when** clause to be used, depending upon one or more contexts: implementation, available devices and the present enclosing construct. The directive in an **otherwise** clause is used when a directive of the **when** clause is not selected.

In the **when** clause the *context selector* (or just *selector*) defines traits that are evaluated for selection of the directive that follows the selector. This "selectables" directive is called a *directive variant*.

In the first example the *arch* trait of the **device** selector set specifies that if an *nvptx* architecture is active in the OpenMP context, then the **teams loop** directive variant is selected as the directive; otherwise, the **parallel loop** directive variant of the **otherwise** clause is selected as the directive. That is, if a device of *nvptx* architecture is supported by the implementation within the enclosing **target** construct, its directive variant is selected. The architecture names, such as *nvptx*, are implementation defined. Also, note that the **device** clause specified in a **target** construct specifies a device number, while **device**, as used in the **metadirective** directive as selector set, has traits of *kind*, *isa* and *arch*.

C / C++

Example metadirective.1.c (**omp_5.2**)

```
#define N 100
#include <stdio.h>

int main()
{
   int v1[N], v2[N], v3[N];
   for(int i=0; i<N; i++){ v1[i]=(i+1); v2[i]=-(i+1); }

   #pragma omp target map(to:v1,v2) map(from:v3) device(0)
   #pragma omp metadirective \
                  when(     device={arch("nvptx")}: teams loop) \
                  otherwise(                       parallel loop)
     for (int i= 0; i< N; i++)  v3[i] = v1[i] * v2[i];

   printf(" %d  %d\n",v3[0],v3[N-1]); //output: -1  -10000

   return 0;
}
```

C / C++

Fortran

Example metadirective.1.f90 (**omp_5.2**)

```
S-1   program main
S-2      integer, parameter :: N= 100
S-3      integer ::  v1(N), v2(N), v3(N);
S-4
S-5      do i=1,N;  v1(i)=i; v2(i)=-i;  enddo   ! initialize
S-6
S-7      !$omp  target map(to:v1,v2) map(from:v3) device(0)
S-8      !$omp  metadirective &
S-9      !$omp&     when(     device={arch("nvptx")}: teams loop) &
S-10     !$omp&     otherwise(                     parallel loop)
S-11       do i= 1,N; v3(i) = v1(i) * v2(i); enddo
S-12     !$omp  end target
S-13
S-14     print *, v3(1),v3(N) !!output: -1  -10000
S-15  end program
```

Fortran

In the second example, the **implementation** selector set is specified in the **when** clause to distinguish between platforms. Additionally, specific architectures are specified with the **device** selector set.

In the code, different **teams** constructs are employed as determined by the **metadirective** directive. The number of teams is restricted by a **num_teams** clause and a thread limit is also set by a **thread_limit** clause for vendor platforms and specific architecture traits. Otherwise, just the **teams** construct is used without any clauses, as prescribed by the **otherwise** clause.

C / C++

Example metadirective.2.c (**omp_5.2**)

```
#define N 100
#include <stdio.h>
#include <omp.h>

void work_on_chunk(int idev, int i);

int main()                      //Driver
{
   int i,idev;

   for (idev=0; idev<omp_get_num_devices(); idev++)
   {
      #pragma omp target device(idev)
      #pragma omp metadirective \
               when( implementation={vendor(nvidia)},            \
                                    device={arch("kepler")}):  \
                     teams num_teams(512) thread_limit(32) )     \
               when( implementation={vendor(amd)},               \
                                    device={arch("fiji"  )}):  \
                     teams num_teams(512) thread_limit(64) )     \
               otherwise(                                        \
                     teams)
      #pragma omp distribute parallel for
      for (i=0; i<N; i++) work_on_chunk(idev,i);
   }
   return 0;
}
```

C / C++

Fortran

Example metadirective.2.f90 (**omp_5.2**)

```
S-1   program main                      !!Driver
S-2     use omp_lib
S-3     implicit none
S-4     integer, parameter :: N=1000
S-5     external           :: work_on_chunk
S-6     integer            :: i,idev
S-7
S-8     do idev=0,omp_get_num_devices()-1
S-9
S-10      !$omp target device(idev)
S-11      !$omp begin metadirective &
S-12      !$omp&  when( implementation={vendor(nvidia)},        &
S-13      !$omp&           device={arch("kepler")}:             &
S-14      !$omp&        teams num_teams(512) thread_limit(32) ) &
S-15      !$omp&  when( implementation={vendor(amd)},           &
S-16      !$omp&           device={arch("fiji"  )}:             &
S-17      !$omp&        teams num_teams(512) thread_limit(64) ) &
S-18      !$omp&  otherwise( teams )
S-19      !$omp distribute parallel do
S-20      do i=1,N
S-21         call work_on_chunk(idev,i)
S-22      end do
S-23      !$omp end metadirective
S-24      !$omp end target
S-25
S-26    end do
S-27
S-28  end program
```

Fortran

In the third example, a **construct** selector set is specified in the **when** clause. Here, a **metadirective** directive is used within a function that is also compiled as a function for a target device as directed by a declare target directive. The **target** directive name of the **construct** selector ensures that the **distribute parallel for/do** construct is employed for the target compilation. Otherwise, for the host-compiled version the **parallel for/do simd** construct is used.

In the first call to the *exp_pi_diff()* routine the context is a **target teams** construct and the **distribute parallel for/do** construct version of the function is invoked, while in the second call the **parallel for/do simd** construct version is used.

This case illustrates an important point for users that may want to hoist the **target** directive out of a function that contains the usual **target teams distribute parallel for/do** construct

(for providing alternate constructs through the **metadirective** directive as here). While this combined construct can be decomposed into a **target** and **teams distribute parallel for/do** constructs, the OpenMP 5.0 specification has the restriction: "If a **teams** construct is nested within a **target** construct, that **target** construct must contain no statements, declarations or directives outside of the **teams** construct". So, the **teams** construct must immediately follow the **target** construct without any intervening code statements (which includes function calls). Since the **target** construct alone cannot be hoisted out of a function, the **target teams** construct has been hoisted out of the function, and the **distribute parallel for/do** construct is used as the variant directive of the **metadirective** directive within the function.

C / C++

Example metadirective.3.c (**omp_5.2**)

```
S-1   #include <stdio.h>
S-2   #include  <math.h>
S-3   #define      N 1000
S-4
S-5   #pragma omp begin declare target
S-6   void exp_pi_diff(double *d, double my_pi){
S-7      #pragma omp metadirective \
S-8                  when(   construct={target}: distribute parallel for ) \
S-9                  otherwise(                  parallel for simd )
S-10     for(int i = 0; i<N; i++) d[i] = exp( (M_PI-my_pi)*i );
S-11  }
S-12  #pragma omp end declare target
S-13
S-14  int main()
S-15  {
S-16    //Calculates sequence of exponentials: (M_PI-my_pi) * index
S-17    //M_PI is from math.h, and my_pi is user provided.
S-18
S-19    double d[N];
S-20    double my_pi=3.14159265358979e0;
S-21
S-22        #pragma omp target teams map(tofrom: d[0:N])
S-23        exp_pi_diff(d,my_pi);
S-24                                              // value should be near 1
S-25        printf("d[N-1] = %20.14f\n",d[N-1]); // ...= 1.00000000000311
S-26
S-27        exp_pi_diff(d,my_pi);                 // value should be near 1
S-28        printf("d[N-1] = %20.14f\n",d[N-1]); // ...= 1.00000000000311
S-29  }
```

C / C++

Fortran

Example metadirective.3.f90 (**omp_5.2**)

```
S-1   module params
S-2      integer, parameter :: N=1000
S-3      DOUBLE PRECISION, PARAMETER::M_PI=4.0d0*DATAN(1.0d0)
S-4                                          ! 3.1415926535897932_8
S-5   end module
S-6
S-7   subroutine exp_pi_diff(d,    my_pi)
S-8     use params
S-9     implicit none
S-10    integer          ::  i
S-11    double precision ::  d(N), my_pi
S-12    !$omp declare target
S-13
S-14    !$omp   metadirective &
S-15    !$omp&      when( construct={target}: distribute parallel do )  &
S-16    !$omp&      otherwise(                parallel do simd )
S-17    do i = 1,size(d)
S-18       d(i) = exp( (M_PI-my_pi)*i )
S-19    end do
S-20
S-21  end subroutine
S-22
S-23  program main
S-24    ! Calculates sequence of exponentials: (M_PI-my_pi) * index
S-25    ! M_PI is from usual way, and my_pi is user provided.
S-26    ! Fortran Standard does not provide PI
S-27
S-28    use params
S-29    implicit none
S-30    double precision   :: d(N)
S-31    double precision   :: my_pi=3.14159265358979d0
S-32
S-33        !$omp target teams map(from: d)
S-34        call exp_pi_diff(d,my_pi)
S-35        !$omp end target teams
S-36                                      ! value should be near 1
S-37        print*, "d(N) = ",d(N)        ! 1.00000000000311
S-38
S-39        call exp_pi_diff(d,my_pi) ! value should be near 1
S-40        print*, "d(N) = ",d(N)        ! 1.00000000000311
S-41
S-42  end program
```

Fortran

The **user** selector set can be used in a **metadirective** to select directives at execution time when the **condition(** *boolean-expr* **)** selector expression is not a constant expression. In this case it is a *dynamic* trait set, and the selection is made at run time, rather than at compile time.

In the following example the `foo` function employs the **condition** selector to choose a device for execution at run time. In the `bar` routine metadirectives are nested. At the outer level a selection between serial and parallel execution in performed at run time, followed by another run time selection on the schedule kind in the inner level when the active *construct* trait is **parallel**.

(Note, the variable `b` in two of the "selected" constructs is declared private for the sole purpose of detecting and reporting that the construct is used. Since the variable is private, its value is unchanged outside of the construct region, whereas it is changed if the "unselected" construct is used.)

C / C++

Example metadirective.4.c (**omp_5.2**)

```
#define N 100
#include <stdbool.h>
#include  <stdlib.h>
#include   <stdio.h>
#include     <omp.h>

void foo(int *a, int n, bool use_gpu)
{
   int b=0;   //  use b to detect if run on gpu

   #pragma omp metadirective \
               when( user={condition(use_gpu)}:           \
                     target teams distribute parallel for \
                     private(b) map(from:a[0:n]) )        \
               otherwise(                                 \
                     parallel for )
   for (int i=0; i<n; i++) {a[i]=i; if(i==n-1) b=1;}

   if(b==0) printf("PASSED 1 of 3\n");
}

void bar (int *a, int n, bool run_parallel, bool unbalanced)
{
   int b=0;
   #pragma omp metadirective \
               when(user={condition(run_parallel)}: parallel)
   {
      if(omp_in_parallel() && omp_get_thread_num() == 0)
         printf("PASSED 2 of 3\n");

```

```
S-31          #pragma omp metadirective \
S-32              when( construct={parallel}, \
S-33                    user={condition(unbalanced)}: for schedule(guided) \
S-34                                                  private(b)) \
S-35              when( construct={parallel}        : for schedule(static))
S-36          for (int i=0; i<n; i++) {a[i]=i; if(i==n-1) b=1;}
S-37       }
S-38       // if guided b=0, because b is private
S-39       if(b==0) printf("PASSED 3 of 3\n");
S-40    }
S-41
S-42    void foo(int *a, int n, bool use_gpu);
S-43    void bar(int *a, int n, bool run_parallel, bool unbalanced);
S-44
S-45    int main(){
S-46
S-47       int p[N];
S-48       // App normally sets these, dependent on input parameters
S-49       bool use_gpu=true, run_parallel=true, unbalanced=true;
S-50
S-51       // Testing: set Env Var MK_FAIL to anything to fail tests
S-52       if(getenv("MK_FAIL")!=NULL) {
S-53          use_gpu=false; run_parallel=false; unbalanced=false;
S-54       }
S-55
S-56       foo(p, N, use_gpu);
S-57       bar(p, N, run_parallel,unbalanced);
S-58
S-59       return 0;
S-60    }
```

C / C++

Fortran

1 *Example metadirective.4.f90* (`omp_5.2`)

```
S-1     subroutine foo(a, n, use_gpu)
S-2        integer :: n, a(n)
S-3        logical :: use_gpu
S-4
S-5        integer :: b=0   !! use b to detect if run on gpu
S-6
S-7        !$omp metadirective &
S-8        !$omp&            when(user={condition(use_gpu)}:           &
S-9        !$omp&                 target teams distribute parallel do  &
S-10       !$omp&                 private(b) map(from:a(1:n)) )        &
S-11       !$omp&            otherwise(                                &
S-12       !$omp&                 parallel do)
```

```
S-13      do i = 1,n; a(i)=i; if(i==n) b=1; end do
S-14
S-15      if(b==0) print *, "PASSED 1 of 3"  ! bc b is firstprivate for gpu run
S-16   end subroutine
S-17
S-18   subroutine bar (a, n, run_parallel, unbalanced)
S-19      use omp_lib, only : omp_get_thread_num, omp_in_parallel
S-20      integer :: n, a(n)
S-21      logical :: run_parallel, unbalanced
S-22
S-23      integer :: b=0
S-24      !$omp begin metadirective when(user={condition(run_parallel)}: parallel)
S-25
S-26       if(omp_in_parallel() .and. omp_get_thread_num() == 0) &
S-27          print *,"PASSED 2 of 3"
S-28
S-29       !$omp metadirective &
S-30       !$omp&  when(construct={parallel}, user={condition(unbalanced)}: &
S-31       !$omp&          do schedule(guided) private(b)) &
S-32       !$omp&  when(construct={parallel}: do schedule(static))
S-33       do i = 1,n; a(i)=i; if(i==n) b=1; end do
S-34
S-35      !$omp end metadirective
S-36
S-37      if(b==0) print *, "PASSED 3 of 3"   !!if guided, b=0 since b is private
S-38   end subroutine
S-39
S-40   program meta
S-41      use omp_lib
S-42      integer, parameter :: N=100
S-43      integer :: p(N)
S-44      integer :: env_stat
S-45                  !! App normally sets these, dependent on input parameters
S-46      logical ::  use_gpu=.true., run_parallel=.true., unbalanced=.true.
S-47
S-48                  !! Testing: set Env Var MK_FAIL to anything to fail tests
S-49      call get_environment_variable('MK_FAIL',status=env_stat)
S-50      if(env_stat /= 1) then                ! status =1 when not set!
S-51         use_gpu=.false.; run_parallel=.false.; unbalanced=.false.
S-52      endif
S-53
S-54
S-55      call foo(p, N, use_gpu)
S-56      call bar(p, N, run_parallel,unbalanced)
S-57
S-58   end program
```

Fortran

Metadirectives can be used in conjunction with templates as shown in the C++ code below. Here the template definition generates two versions of the Fibonacci function. The *tasking* boolean is used in the **condition** selector to enable tasking. The true form implements a parallel version with **task** and **taskwait** constructs as in the *tasking.4.c* code in Section 5.1. The false form implements a serial version without any tasking constructs. Note that the serial version is used in the parallel function for optimally processing numbers less than 8.

C++

Example metadirective.5.cpp (**omp_5.0**)

```
S-1   #include <stdio.h>
S-2
S-3   // revised Fibonacci from tasking.4.c example
S-4
S-5   template <bool tasking>
S-6   int fib(int n) {
S-7     int i, j;
S-8     if (n<2) {
S-9       return n;
S-10    } else if ( tasking && n<8 ) { // serial/taskless cutoff for n<8
S-11      return fib<false>(n);
S-12    } else {
S-13      #pragma omp metadirective \
S-14                  when(user={condition(tasking)}: task shared(i))
S-15      {
S-16        i=fib<tasking>(n-1);
S-17      }
S-18      #pragma omp metadirective \
S-19                  when(user={condition(tasking)}: task shared(j))
S-20      {
S-21        j=fib<tasking>(n-2);
S-22      }
S-23      #pragma omp metadirective \
S-24                  when(user={condition(tasking)}: taskwait)
S-25      return i+j;
S-26    }
S-27  }
S-28
S-29  int main(int argc, char** argv) {
S-30    int n = 15;
S-31    #pragma omp parallel
S-32    #pragma omp single
S-33    {
S-34      printf("fib(%i) = %i\n", n, fib<true>(n));
S-35    }
S-36    return 0;
S-37  }
```

```
S-38    // OUTPUT:
S-39    // fib(15) = 610
```

C++

12.7.3 Context Selector Scoring

Each context selector for which all specified traits are active in the current *OpenMP context* is a *compatible context selector*, and the associated function variant or directive variant for such a context selector is a *replacement candidate*. The final *score* of each of the compatible context selectors determine which of the replacement candidates is selected for substitution.

For a given compatible context selector, the score is calculated according to the specified trait selectors and their corresponding traits. If the trait selectors are a strict subset of the trait selectors specified by another compatible context selector then the score of the context selector is zero. Otherwise, the final score is one plus the sum of the score values of each specified trait selector.

A replacement candidate is selected if no other candidate has a higher scoring context selector. If multiple replacement candidates have a context selector with the same highest score, the one specified first on the metadirective is selected. If multiple function variants are replacement candidates that have context selectors with the same highest score, the one that is selected is implementation defined.

If a **construct** selector set is specified in the context selector, each active construct trait that is named in that selector set contributes a score of 2^{p-1}, where p is the position of that trait in the current *construct* trait set (the set of traits in the OpenMP context). If a **device** or **target_device** selector set is specified in the selector, then an active *kind*, *arch*, or *isa* trait that is named in the selector set contributes a score of 2^{l}, 2^{l+1}, and 2^{l+2}, respectively, where l is the number of traits in the *construct* trait set. For any other active traits that are named in the context selector that are not implementation-defined extensions, the contributed score, by default, is zero.

The default score for any active traits other than *construct* traits and the *kind*, *arch*, or *isa* traits may be overridden with an explicit score expression. Specifying an explicit score is only recommended for prioritizing replacement candidates for which a selection is not dependent on construct traits. That is, none of the compatible context selectors specify a **construct** trait selector or a **kind**, **arch**, or **isa** trait selector.

In the following example, four function variants are declared for the procedure `f`: `fx1`, `fx2`, `fx3`, and `fx4`. Suppose that the target device for the **target** region has the *gpu* device kind, has the *nvptx* architecture, and supports the *sm_70* instruction set architecture. Hence, the context selectors for all function variants are compatible with the context at the callsite for `f` inside the **target** region. The *construct* trait set at the callsite, consisting of all enclosing constructs and having a count of *l=6*, is: {*target*, *teams*, *distribute*, *parallel*, *for/do*, *task*}. Note that only *context-matching* constructs, which does not include **distribute** or **task**, may be named by a **construct** trait selector as of OpenMP 5.2. The score for `fx1` is $1 + 2^0 = 2$, for `fx2` is $1 + 2^1 + 2^3 + 2^4 = 27$, for

fx3 is $1 + 2^6 + 2^8 = 321$, and for *fx4* is $1 + 2^7 + 2^8 = 385$. Since *fx4* is the function variant that has the highest scoring selector, it is selected by the implementation at the callsite.

C / C++

Example selector_scoring.1.c (**omp_5.0**)

```
S-1
S-2     #include <stdlib.h>
S-3     #include <stdio.h>
S-4
S-5     void fx1(int *a, int i)
S-6     {
S-7       *a = i;
S-8     }
S-9
S-10    void fx2(int *a, int i)
S-11    {
S-12       *a = 2*i;
S-13    }
S-14
S-15    void fx3(int *a, int i)
S-16    {
S-17       *a = 3*i;
S-18    }
S-19
S-20    void fx4(int *a, int i)
S-21    {
S-22       *a = 4*i;
S-23    }
S-24
S-25    #pragma omp declare variant(fx1) match(construct={target})
S-26    #pragma omp declare variant(fx2) match(construct={teams,parallel,for})
S-27    #pragma omp declare variant(fx3) match(device={kind(gpu),isa(sm_70)})
S-28    #pragma omp declare variant(fx4) match(device={arch(nvptx),isa(sm_70)})
S-29    void f(int *a, int i)
S-30    {
S-31      *a = i;
S-32    }
S-33
S-34    int main()
S-35    {
S-36       #define N 4
S-37       int a[N];
S-38       #pragma omp target teams distribute parallel for map(a[:N])
S-39       for (int i = 0; i < N; i++) {
S-40          #pragma omp task
S-41          {
```

```
S-42            f(&a[i], i);
S-43         }
S-44      }
S-45
S-46      for (int i = 0; i < N; i++) {
S-47         if (a[i] != 4*i) {
S-48            printf("Failed\n");
S-49            return 1;
S-50         }
S-51      }
S-52
S-53      printf("Passed\n");
S-54      return 0;
S-55   }
```

C / C++

Fortran

1 *Example selector_scoring.1.f90* (`omp_5.0`)

```
S-1    module subs
S-2    contains
S-3
S-4       subroutine f(a,i)
S-5          !$omp declare variant(fx1) match(construct={target})
S-6          !$omp declare variant(fx2) match(construct={teams,parallel,do})
S-7          !$omp declare variant(fx3) match(device={kind(gpu),isa(sm_70)})
S-8          !$omp declare variant(fx4) match(device={arch(nvptx),isa(sm_70)})
S-9          integer, intent(out) :: a
S-10         integer, value :: i
S-11         a = i
S-12      end subroutine
S-13
S-14      subroutine fx1(a,i)
S-15         integer, intent(out) :: a
S-16         integer, value :: i
S-17         a = i
S-18      end subroutine
S-19
S-20      subroutine fx2(a,i)
S-21         integer, intent(out) :: a
S-22         integer, value :: i
S-23         a = 2*i
S-24      end subroutine
S-25
S-26      subroutine fx3(a,i)
S-27         integer, intent(out) :: a
S-28         integer, value :: i
```

```
S-29          a = 3*i
S-30       end subroutine
S-31
S-32       subroutine fx4(a,i)
S-33          integer, intent(out) :: a
S-34          integer, value :: i
S-35          a = 4*i
S-36       end subroutine
S-37
S-38    end module subs
S-39
S-40
S-41    program main
S-42       use subs
S-43       integer, parameter :: N = 4
S-44       integer :: a(N)
S-45       integer :: i
S-46
S-47       !$omp target teams distribute parallel do map(a)
S-48       do i = 1, N
S-49          !$omp task
S-50             call f(a(i),i)
S-51          !$omp end task
S-52       end do
S-53
S-54       do i = 1, N
S-55          if (a(i) /= 4*i ) then
S-56             print *, "Failed"
S-57             error stop
S-58          end if
S-59       end do
S-60
S-61       print *, "Passed"
S-62
S-63    end program
```

Fortran

In the next example, three function variants are declared for the procedure `kernel`: `kernel_target_ua`, `kernel_target_usm`, and `kernel_target_usm_v2`. Suppose that the implementation supports the *unified_address* and *unified_shared_memory* requirements, so that the context selectors for all function variants are compatible. The score for `kernel_target_ua` is 1, which is one plus the zero score associated with the active *unified_address* requirement. The score for `kernel_target_usm` is 0, as the selector is a strict subset of the selector for `kernel_target_usm_v2`. The score for `kernel_target_usm_v2` is 2, which is one plus the explicit score of 1 for the *condition* trait and the zero score associated with the acive *unified_shared_memory* requirement . Since

kernel_target_usm_v2 is the function variant that has the highest scoring selector, it is selected by the implementation at the callsite.

C / C++

Example selector_scoring.2.c (**omp_5.0**)

```
S-1
S-2   #include <stdio.h>
S-3
S-4   // The unified_address and/or unified_shared_memory requirements may be
S-5   // implicitly enforced by the implementation via compiler flags.
S-6
S-7   const int version = 2;
S-8
S-9   void kernel_target_ua(int *a, int n)
S-10  {
S-11     #pragma omp target data map(a[:n]) use_device_ptr(a)
S-12     #pragma omp target parallel for
S-13     for (int i = 0; i < n; i++) {
S-14        a[i] = 2*i*i;
S-15     }
S-16  }
S-17
S-18  void kernel_target_usm(int *a, int n)
S-19  {
S-20     #pragma omp target parallel for
S-21     for (int i = 0; i < n; i++) {
S-22        a[i] = 3*i*i;
S-23     }
S-24  }
S-25
S-26  void kernel_target_usm_v2(int *a, int n)
S-27  {
S-28     #pragma omp target teams loop
S-29     for (int i = 0; i < n; i++) {
S-30        a[i] = 4*i*i;
S-31     }
S-32  }
S-33
S-34  #pragma omp declare variant(kernel_target_ua) \
S-35     match(implementation={requires(unified_address)})
S-36  #pragma omp declare variant(kernel_target_usm) \
S-37     match(implementation={requires(unified_shared_memory)})
S-38  #pragma omp declare variant(kernel_target_usm_v2) \
S-39     match(implementation={requires(unified_shared_memory)}, \
S-40           user={condition(score(1): version==2)})
S-41  void kernel(int *a, int n)
```

```
{
   #pragma omp parallel for
   for (int i = 0; i < n; i++) {
      a[i] = i*i;
   }
}

int main()
{
   int a[1000];

   kernel(a, 1000);

   for (int i = 0; i < 1000; i++) {
      if (a[i] != 4*i*i) {
         printf("Failed\n");
         return 1;
      }
   }

   printf("Passed\n");
   return 0;
}
```

C / C++

Fortran

Example selector_scoring.2.f90 (`omp_5.0`)

```
module subs
   ! The unified_address and/or unified_shared_memory requirements may be
   ! implicitly enforced by the implementation via compiler flags.

   integer, parameter :: version = 2
contains

   subroutine kernel(a, n)
      !$omp declare variant(kernel_target_ua) &
      !$omp     match(implementation={requires(unified_address)})

      !$omp declare variant(kernel_target_usm) &
      !$omp     match(implementation={requires(unified_shared_memory)})

      !$omp declare variant(kernel_target_usm_v2) &
      !$omp     match(implementation={requires(unified_shared_memory)}, &
      !$omp           user={condition(score(1): version==2)})

      integer, target :: a(n)
```

```
S-20          integer, value :: n
S-21          integer :: i
S-22          !$omp parallel do
S-23          do i = 1, n
S-24             a(i) = i*i
S-25          end do
S-26       end subroutine
S-27
S-28       subroutine kernel_target_ua(a, n)
S-29          use iso_c_binding
S-30          integer, target :: a(n)
S-31          integer, value :: n
S-32          type(c_ptr) :: c_ap
S-33          integer, pointer :: ap(:)
S-34          integer :: i
S-35          c_ap = c_loc(a)
S-36          ap => null()
S-37          !$omp target data map(a(:n)) use_device_ptr(c_ap)
S-38          !$omp target
S-39             call c_f_pointer(c_ap, ap, [n])
S-40             !$omp parallel do
S-41             do i = 1, n
S-42                ap(i) = 2*i*i
S-43             end do
S-44             ap => null() ! reset pointer association status
S-45          !$omp end target
S-46          !$omp end target data
S-47       end subroutine
S-48
S-49       subroutine kernel_target_usm(a, n)
S-50          integer, target :: a(n)
S-51          integer, value :: n
S-52          integer :: i
S-53          !$omp target parallel do
S-54          do i = 1, n
S-55             a(i) = 3*i*i
S-56          end do
S-57       end subroutine
S-58
S-59       subroutine kernel_target_usm_v2(a, n)
S-60          integer, target :: a(n)
S-61          integer, value :: n
S-62          integer :: i
S-63          !$omp target teams loop
S-64          do i = 1, n
S-65             a(i) = 4*i*i
S-66          end do
```

```
S-67        end subroutine
S-68
S-69     end module subs
S-70
S-71
S-72     program main
S-73        use subs
S-74        integer, target :: a(1000)
S-75
S-76        call kernel(a, 1000)
S-77
S-78        do i = 1, 1000
S-79           if (a(i) /= 4*i*i ) then
S-80              print *, "Failed"
S-81              error stop
S-82           end if
S-83        end do
S-84
S-85        print *, "Passed"
S-86
S-87     end program
```

Fortran

12.8 `dispatch` Construct

The **dispatch** directive can be applied to a statement that performs a procedure call to control variant substitution for the called procedure.

In the example below, the *foo_variant1()* and *foo_variant2()* procedures are declared as variants for *foo()* using the **declare variant** directive, with matching requirements specified by the **match** clause's context selector. To be selected for substitution, both variants require that the condition *foo_sub* evaluates to *true*.

In Cases 1 and 2, the calls to *foo()* are not controlled by a **dispatch** construct. Hence, there can be no match for the *foo_variant2()* variant. A *foo_variant1()* call is substituted for the call to *foo()* in Case 1, as the matching requirement is satisfied by *foo_sub* being *true*. In Case 2, there is no variant substitution as *foo_sub* is *false*.

Cases 3 through 6 illustrate some uses of the **dispatch** construct, including uses of the **novariants** and **nocontext** clauses on the directive.

In Case 3, variant substitution does not occur as *foo_sub* is *false*.

In Case 4, *foo_sub* is *true* and the **dispatch** construct is part of the OpenMP context; therefore, the matching requirements for both variants to *foo()* are satisfied. As the matching requirements for the *foo_variant1()* variant are a subset of the matching requirements for the *foo_variant2()* variant (per the OpenMP specification, its computed score for matching purposes is smaller), *foo_variant2()* is selected for variant substitution. (Note that prior to OpenMP 6.0, which of the two variants are selected for substitution is implementation defined since the earlier specifications did not require an implementation to treat the **dispatch** construct as part of the OpenMP context at the call site.)

In Case 5, the **novariants** clause disables variant substitution for the call to *foo()*, despite the matching requirements being satisfied for both variants.

In Case 6, the **nocontext** clause directs the implementation to not include the **dispatch** construct in the OpenMP context at the call site for *foo()*. Hence, the *foo_variant2()* variant is not considered and *foo_variant1()* is instead selected for variant substitution.

C / C++

Example dispatch.1.c (**omp_6.0**)

```
S-1   #include <stdio.h>
S-2
S-3   int foo_sub;
S-4
S-5   void foo_variant1()
S-6   { printf("in foo_variant1\n"); }
S-7
S-8   void foo_variant2()
S-9   { printf("in foo_variant2\n"); }
```

```
S-10
S-11    #pragma omp declare variant(foo_variant1) \
S-12                        match(user={condition(foo_sub)})
S-13    #pragma omp declare variant(foo_variant2) \
S-14                        match(construct={dispatch},user={condition(foo_sub)})
S-15    void foo()
S-16    { printf("in foo\n"); }
S-17
S-18    int main()
S-19    {
S-20       // Case 1
S-21       foo_sub = 1;
S-22       foo();         // "in foo_variant1"
S-23
S-24       // Case 2
S-25       foo_sub = 0;
S-26       foo();         // "in foo"
S-27
S-28       // Dispatch Cases
S-29
S-30       // Case 3
S-31       foo_sub = 0;
S-32       #pragma omp dispatch
S-33       foo();         // "in foo"
S-34
S-35       // Case 4
S-36       foo_sub = 1;
S-37       #pragma omp dispatch
S-38       foo();         // "in foo_variant2"
S-39                      // see discussion for OpenMP 5.1/5.2
S-40
S-41       // Case 5
S-42       foo_sub = 1;
S-43       #pragma omp dispatch novariants(1)
S-44       foo();         // "in foo"
S-45
S-46       // Case 6
S-47       foo_sub = 1;
S-48       #pragma omp dispatch nocontext(1)
S-49       foo();         // "in foo_variant1"
S-50
S-51       return 0;
S-52    }
```

C / C++

Example dispatch.1.f90 (`omp_6.0`)

```
S-1   module funcs
S-2     logical :: foo_sub
S-3
S-4   contains
S-5     subroutine foo_variant1()
S-6       print*, "in foo_variant1"
S-7     end subroutine
S-8
S-9     subroutine foo_variant2()
S-10      print*, "in foo_variant2"
S-11    end subroutine
S-12
S-13    subroutine foo()
S-14      !$omp  declare variant(foo_variant1) &
S-15      !$omp&         match(user={condition(foo_sub)})
S-16      !$omp  declare variant(foo_variant2) &
S-17      !$omp          match(construct={dispatch},user={condition(foo_sub)})
S-18      print*, "in foo"
S-19    end subroutine
S-20
S-21  end module funcs
S-22
S-23  program main
S-24    use funcs
S-25
S-26    !! Case 1
S-27    foo_sub = .TRUE.
S-28    call foo()          !! "in foo_variant1"
S-29
S-30    !! Case 2
S-31    foo_sub = .FALSE.
S-32    call foo()          !! "in foo"
S-33
S-34    !! Dispatch Cases
S-35
S-36    !! Case 3
S-37    foo_sub=.FALSE.
S-38    !$omp dispatch
S-39    call foo()          !! "in foo"
S-40
S-41    !! Case 4
S-42    foo_sub = .TRUE.
S-43    !$omp dispatch
S-44    call foo();         !! "in foo_variant2"
```

```
                        !! see discussion for OpenMP 5.1/5.2

  !! Case 5
  foo_sub = .TRUE.
  !$omp dispatch novariants(.true.)
  call foo();         !!  "in foo"

  !! Case 6
  foo_sub = .TRUE.
  !$omp dispatch nocontext(.true.)
  call foo();         !! "in foo_variant1"

end program
```

Fortran

12.9 Nested Loop Constructs

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

C / C++

Example nested_loop.1.c (**pre_omp_3.0**)

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

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

C / C++

Fortran

Example nested_loop.1.f (`pre_omp_3.0`)

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

Fortran

The following variation of the preceding example is also conforming:

C / C++

Example nested_loop.2.c (`pre_omp_3.0`)

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

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

void good_nesting2(int n)
{
  int i;
  #pragma omp parallel default(shared)
```

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

C / C++

Fortran

Example nested_loop.2.f (`pre_omp_3.0`)

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

Fortran

12.10 Restrictions on Nesting of Regions

The examples in this section illustrate the region nesting rules.

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

C / C++

Example nesting_restrict.1.c (`pre_omp_3.0`)

```
S-1   void work(int i, int j) {}
S-2
S-3   void wrong1(int n)
S-4   {
S-5
S-6     #pragma omp parallel default(shared)
S-7     {
S-8       int i, j;
S-9       #pragma omp for
S-10      for (i=0; i<n; i++) {
S-11         /* incorrect nesting of loop regions */
S-12         #pragma omp for
S-13           for (j=0; j<n; j++)
S-14             work(i, j);
S-15      }
S-16    }
S-17
S-18  }
```

C / C++

Fortran

Example nesting_restrict.1.f (`pre_omp_3.0`)

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

```
S-16                 END DO
S-17               END DO
S-18     !$OMP   END PARALLEL
S-19
S-20           END SUBROUTINE WRONG1
```

Fortran

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

C / C++

2 *Example nesting_restrict.2.c* (**pre_omp_3.0**)

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

C / C++

Fortran

Example nesting_restrict.2.f (`pre_omp_3.0`)

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

Fortran

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

C / C++

Example nesting_restrict.3.c (`pre_omp_3.0`)

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

C / C++

Fortran

Example nesting_restrict.3.f (**pre_omp_3.0**)

```
      SUBROUTINE WRONG3(N)
      INTEGER N

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

Fortran

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

C / C++

Example nesting_restrict.4.c (**pre_omp_3.0**)

```
void work(int i, int j) {}
void wrong4(int n)
{

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

C / C++

Fortran

Example nesting_restrict.4.f (`pre_omp_3.0`)

```
      SUBROUTINE WRONG4(N)
      INTEGER N

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

Fortran

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

C / C++

Example nesting_restrict.5.c (`pre_omp_3.0`)

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

C / C++

Fortran

Example nesting_restrict.5.f (**pre_omp_3.0**)

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

Fortran

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

C / C++

Example nesting_restrict.6.c (**pre_omp_3.0**)

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

C / C++

Fortran

Example nesting_restrict.6.f (**pre_omp_3.0**)

```
      SUBROUTINE WRONG6(N)
      INTEGER N

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

Fortran

12.11 Target Offload

In the OpenMP 5.0 implementation the **OMP_TARGET_OFFLOAD** environment variable was defined to change default offload behavior. By default the target code (region) is executed on the host if the target device does not exist or the implementation does not support the target device.

In an OpenMP 5.0 compliant implementation, setting the **OMP_TARGET_OFFLOAD** variable to `MANDATORY` will force the program to terminate execution when a **target** construct is encountered and the target device is not supported or is not available. With a value `DEFAULT` the target region will execute on a device if the device exists and is supported by the implementation, otherwise it will execute on the host. Support for the `DISABLED` value is optional; when it is supported the behavior is as if only the host device exists (other devices are considered non-existent to the runtime), and target regions are executed on the host.

The following example reports execution behavior for different values of the **OMP_TARGET_OFFLOAD** variable. A handy routine for extracting the **OMP_TARGET_OFFLOAD** environment variable value is deployed here, because the OpenMP API does not have a routine for obtaining the value.

Note: The example issues a warning when a pre-5.0 implementation is used, indicating that the **OMP_TARGET_OFFLOAD** is ignored. The value of the **OMP_TARGET_OFFLOAD** variable is reported when the **OMP_DISPLAY_ENV** environment variable is set to `TRUE` or `VERBOSE`.

C / C++

Example target_offload_control.1.c (`omp_5.0`)

```
#include     <omp.h>
#include   <stdio.h>
#include   <ctype.h>
#include  <stdlib.h>
#include  <string.h>
#include <strings.h>

typedef enum offload_policy
{MANDATORY, DISABLED, DEFAULT, UNKNOWN, NOTSET} offload_policy_t;


offload_policy_t get_offload_policy()
{
   char *env, *end;
   size_t n;

   env = getenv("OMP_TARGET_OFFLOAD");
   if(env  == NULL) return NOTSET;

   end = env + strlen(env);                 //Find trimmed beginning/end
   while (      *env && isspace(*(env  )) ) env++;
   while (end != env && isspace(*(end-1)) ) end--;
   n = (int)(end - env);

                     //Find ONLY string -nothing more, case insensitive
   if      (n == 9 && !strncasecmp(env, "MANDATORY",n)) return MANDATORY;
   else if (n == 8 && !strncasecmp(env, "DISABLED" ,n)) return DISABLED ;
   else if (n == 7 && !strncasecmp(env, "DEFAULT"  ,n)) return DEFAULT  ;
   else                                                 return UNKNOWN  ;
}


int main()
{
   int device_num, on_init_dev;

   // get policy from OMP_TARGET_OFFLOAD variable
   offload_policy_t policy = get_offload_policy();

   if(_OPENMP< 201811)
   {
      printf("Warning: OMP_TARGET_OFFLOAD NOT supported, version %d\n",
             _OPENMP );
      printf("         If OMP_TARGET_OFFLOAD is set, "
```

```
S-45                "it will be ignored.\n");
S-46      }
S-47
S-48      // Set target device number to an unavailable
S-49      // device to test offload policy.
S-50      device_num = omp_get_num_devices() + 1;
S-51
S-52      // Policy:
S-53      printf("OMP_TARGET_OFFLOAD Policy:  ");
S-54      if     (policy==MANDATORY)
S-55         printf("MANDATORY-Terminate if dev. not avail\n");
S-56      else if(policy==DISABLED )
S-57         printf("DISABLED -(if supported) Only on Host\n");
S-58      else if(policy==DEFAULT  )
S-59         printf("DEFAULT  -On host if device not avail\n");
S-60      else if(policy==UNKNOWN  )
S-61         printf("OMP_TARGET_OFFLOAD has unknown value\n" );
S-62      else if(policy==NOTSET   )
S-63         printf("OMP_TARGET_OFFLOAD not set\n" );
S-64
S-65
S-66      on_init_dev = 1;
S-67      // device# out of range--not supported
S-68      #pragma omp target device(device_num) map(tofrom: on_init_dev)
S-69        on_init_dev=omp_is_initial_device();
S-70
S-71      if (policy == MANDATORY && _OPENMP >= 201811)
S-72         printf("ERROR: OpenMP implementation ignored MANDATORY policy.\n");
S-73
S-74      printf("Target region executed on init dev %s\n",
S-75             on_init_dev ? "TRUE":"FALSE");
S-76
S-77      return 0;
S-78   }
```

C / C++

Fortran

1 *Example target_offload_control.1.f90* (`omp_5.0`)

```
S-1    module offload_policy
S-2      implicit none
S-3      integer, parameter :: LEN_POLICY=10
S-4    contains
S-5      character(LEN_POLICY) function get_offload_policy()
S-6         character(64) :: env
S-7         integer       :: length, i
S-8         env=repeat(' ',len(env))
```

```
S-9                                     !policy is blank if not found *
S-10        call get_environment_variable("OMP_TARGET_OFFLOAD",env,length)
S-11
S-12        do i = 1,len(env)               !Makes a-z upper case
S-13           if(iachar(env(i:i))>96) env(i:i)=achar(iachar(env(i:i))-32)
S-14        end do
S-15
S-16        get_offload_policy = trim(adjustl(env)) !remove peripheral spaces
S-17
S-18        if(length==0) get_offload_policy="NOTSET"
S-19
S-20        return
S-21
S-22      end function
S-23
S-24    end module
S-25
S-26    program policy_test
S-27
S-28      use omp_lib
S-29      use offload_policy
S-30
S-31      integer                 :: i, device_num
S-32      logical                 :: on_init_dev
S-33      character(LEN_POLICY)   :: policy
S-34
S-35      policy = get_offload_policy() !!Get OMP_TARGET_OFFLOAD value
S-36
S-37      if (OPENMP_VERSION < 201811) then
S-38         print*,"Warning: OMP_TARGET_OFFLOAD NOT supported by VER.", &
S-39                OPENMP_VERSION
S-40         print*,"         If OMP_TARGET_OFFLOAD is set, it will be ignored."
S-41      endif
S-42
S-43         ! Set target device number to an unavailable device
S-44         ! to test offload policy.
S-45      device_num = omp_get_num_devices() + 1
S-46
S-47                           !! Report OMP_TARGET_OFFOAD value
S-48      select CASE (policy)
S-49         case("MANDATORY")
S-50              print*,"Policy:  MANDATORY-Terminate if dev. not avail."
S-51         case("DISABLED")
S-52              print*,"Policy:  DISABLED-(if supported) Only on Host."
S-53         case("DEFAULT")
S-54              print*,"Policy:  DEFAULT On host if device not avail."
S-55         case("NOTSET")
```

```
            print*,"         OMP_TARGET_OFFLOAD is not set."
       case DEFAULT
            print*,"         OMP_TARGET_OFFLOAD has unknown value."
            print*,"         UPPER CASE VALUE=",policy
  end select


  on_init_dev = .FALSE.
                      !! device# out of range--not supported
  !$omp target  device(device_num) map(tofrom: on_init_dev)
     on_init_dev=omp_is_initial_device()
  !$omp end target

  if (policy=="MANDATORY" .and. OPENMP_VERSION>=201811) then
     print*,"OMP ERROR: ", &
            "OpenMP 5.0 implementation ignored MANDATORY policy."
     print*,"           Termination should have occurred", &
            " at target directive."
  endif

  print*, "Target executed on init dev (T|F): ", on_init_dev

end program policy_test
```

Fortran

12.12 `omp_pause_resource` and `omp_pause_resource_all` Routines

Sometimes, it is necessary to relinquish resources created or allocated for the OpenMP runtime environment to avoid interference with subsequent actions as illustrated by the following example. In the beginning either a call to the **`omp_get_max_threads`** routine or the subsequent **`parallel`** construct may trigger resource allocation by the OpenMP runtime, which may cause unexpected side effects for the subsequent *`fork`* call. It is desirable to relinquish OpenMP resources allocated before the fork by using the **`omp_pause_resource`** routine for a given device, in this case the host device. The host device number is returned by the **`omp_get_initial_device`** routine. The **`omp_pause_hard`** value is used here to free as many OpenMP resources as possible. After the fork, the child process will initialize its OpenMP runtime environment when encountering the **`parallel`** construct.

C / C++

Example pause_resource.1.c (`omp_5.0`)

```
S-1   #include <stdio.h>
S-2   #include <stdlib.h>
S-3   #include <unistd.h>
S-4   #include <sys/wait.h>
S-5   #include <omp.h>
S-6
S-7   int main()
S-8   {
S-9      pid_t pid;
S-10     int nt = omp_get_max_threads();
S-11
S-12     #pragma omp parallel
S-13     {
S-14        #pragma omp single
S-15           printf("number of threads = %d (max = %d)\n",
S-16                   omp_get_num_threads(), nt);
S-17     }
S-18
S-19     /* clean up thread environment before fork */
S-20     omp_pause_resource(omp_pause_hard, omp_get_initial_device());
S-21
S-22     pid = fork();
S-23     if (pid < 0) {
S-24        printf("fork failed\n");
S-25        exit(1);
S-26     }
S-27     else if (pid == 0) {
S-28        /* child process */
S-29        #pragma omp parallel num_threads(nt)
S-30        {
S-31           int myid = omp_get_thread_num();
S-32           printf("child: myid %d of %d\n", myid, nt);
S-33        }
S-34     }
S-35     else {
S-36        /* parent process */
S-37        int s;
S-38        printf("parent process - waiting pid %d\n", pid);
S-39        waitpid(pid, &s, 0);
S-40     }
S-41     return 0;
S-42  }
```

C / C++

The following example illustrates a different use case. After executing the first parallel code (parallel region 1), the *relinquish* program switches to executing an external parallel program (called *subprogram*, which is compiled from *pause_resource.2b*). In order to make resources available for the external subprogram, *relinquish* calls **omp_pause_resource_all** to relinquish OpenMP resources used by the current program before calling *execute_command_line* to execute *subprogram*. The **omp_pause_soft** value is used here to allow subsequent OpenMP regions (parallel region 2) to restart more quickly.

Fortran

Example pause_resource.2a.f90 (**omp_5.0**)

```
S-1   program relinquish
S-2      use omp_lib
S-3      implicit none
S-4      integer :: err
S-5
S-6      write (*,*) 'In relinquish'
S-7
S-8      !$omp parallel
S-9      write (*,*) 'In parallel region 1'
S-10     !$omp end parallel
S-11
S-12     err = omp_pause_resource_all(omp_pause_soft)
S-13
S-14     ! execute the external subprogram produced from pause_resource.2b
S-15     call execute_command_line(command='./subprogram', wait=.true., &
S-16                               cmdstat=err)
S-17     if (err /= 0) write (*,*) 'Warning: subprogram failed to execute'
S-18
S-19     !$omp parallel
S-20     write (*,*) 'In parallel region 2'
S-21     !$omp end parallel
S-22
S-23  end program relinquish
```

Fortran

Fortran

Example pause_resource.2b.f90 (**pre_omp_3.0**)

```
S-1   ! This program compiles to an executable "subprogram"
S-2   subroutine compute(i, j, k, r)
S-3      implicit none
S-4      integer :: i, j, k
S-5      real(8) :: r
S-6      r = i + j + k
S-7   end subroutine compute
S-8
S-9   program subprogram
S-10     implicit none
S-11     integer :: i, j, k
S-12     real(8) :: s, r
S-13     integer, parameter :: n = 500
S-14
S-15     write (*,*) 'In subprogram'
S-16     s = 0.d0
S-17  !$omp parallel do private(r) reduction(+:s)
S-18     do i = 1, n
S-19        do j = 1, n
S-20           do k = 1, n
S-21              call compute(i, j, k, r)
S-22              s = s + r
S-23           end do
S-24        end do
S-25     end do
S-26  !$omp end parallel do
S-27
S-28     write (*,*) 'Sum = ',s
S-29  end program subprogram
```

Fortran

12.13 Controlling Concurrency and Reproducibility with the `order` Clause

The **order** clause is used for controlling the parallel execution of loop iterations for one or more loops that are associated with a directive. It is specified with a clause argument and optional modifier. The only supported argument, introduced in OpenMP 5.0, is the keyword **concurrent** which indicates that the loop iterations may execute concurrently, including iterations in the same chunk per the loop schedule. Because of the relaxed execution permitted with an

order(concurrent) clause, codes must not assume that any cross-iteration data dependences would be preserved or that any two iterations may execute on the same thread.

The following example in this section demonstrates the use of the **order(concurrent)** clause, without any modifiers, for controlling the parallel execution of loop iterations. The **order(concurrent)** clause cannot be used for the second and third **parallel for/do** constructs because of either having data dependences or accessing threadprivate variables.

C / C++

Example reproducible.1.c (**omp_5.0**)

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

int main()
{
   const int n = 1000;
   int v[n], u[n];
   static int sum;
   #pragma omp threadprivate(sum)

   // no data dependences, so can execute concurrently
   #pragma omp parallel for order(concurrent)
   for (int i = 0; i < n; i++) {
      u[i] = i;
      v[i] = i;
      v[i] += u[i] * u[i];
   }

   // with data dependences, so cannot execute iterations
   // concurrently with the order(concurrent) clause
   #pragma omp parallel for ordered
   for (int i = 1; i < n; i++) {
      v[i] += u[i] * u[i];
      #pragma omp ordered
          v[i] += v[i-1];
   }

   sum = 0;
   // accessing a threadprivate variable, which would not be
   // permitted if the order(concurrent) clause was present
   #pragma omp parallel for copyin(sum)
   for (int i = 0; i < n; i++) {
      sum += v[i];
   }

   #pragma omp parallel
```

```
S-37        {
S-38           printf("sum = %d on thread %d\n", sum, omp_get_thread_num());
S-39        }
S-40
S-41        return 0;
S-42     }
```

C / C++

Fortran

Example reproducible.1.f90 (`omp_5.0`)

```
 S-1     program main
 S-2        use omp_lib
 S-3        implicit none
 S-4        integer, parameter :: n = 1000
 S-5        integer :: v(n), u(n)
 S-6        integer :: i
 S-7        integer, save :: sum
 S-8        !$omp threadprivate(sum)
 S-9
S-10        !! no data dependences, so can execute concurrently
S-11        !$omp parallel do order(concurrent)
S-12        do i = 1, n
S-13           u(i) = i
S-14           v(i) = i
S-15           v(i) = v(i) + u(i) * u(i)
S-16        end do
S-17
S-18        !! with data dependences, so cannot execute iterations
S-19        !! concurrently with the order(concurrent) clause
S-20        !$omp parallel do ordered
S-21        do i = 2, n
S-22           v(i) = v(i) + u(i) * u(i)
S-23           !$omp ordered
S-24              v(i) = v(i) + v(i-1)
S-25           !$omp end ordered
S-26        end do
S-27
S-28        sum = 0
S-29        !! accessing a threadprivate variable, which would not be
S-30        !! permitted if the order(concurrent) clause was present
S-31        !$omp parallel do copyin(sum)
S-32        do i = 2, n
S-33           sum = sum + v(i)
S-34        end do
S-35
S-36        !$omp parallel
```

```
S-37        print *,"sum = ",sum," on thread ", omp_get_thread_num()
S-38     !$omp end parallel
S-39
S-40  end program
```

Fortran

Modifiers to the **order** clause, introduced in OpenMP 5.1, may be specified to control the reproducibility of the loop schedule for the associated loop(s). A reproducible loop schedule will consistently yield the same mapping of iterations to threads (or SIMD lanes) if the directive name, loop schedule, iteration space, and binding region remain the same. The **reproducible** modifier indicates the loop schedule must be reproducible, while the **unconstrained** modifier indicates that the loop schedule is not reproducible. If a modifier is not specified, then the **order** clause does not affect the reproducibility of the loop schedule.

The next example demonstrates the use of the **order(concurrent)** clause with modifiers for additionally controlling the reproducibility of a loop's schedule. The two worksharing-loop constructs in the first **parallel** construct specify that the loops have reproducible schedules, thus memory effects from iteration *i* from the first loop will be observable to iteration *i* in the second loop. In the second **parallel** construct, the **order** clause does not control reproducibility for the loop schedules. However, since both loops specify the same static schedules, the schedules are reproducible and the data dependences between the loops are preserved by the execution. In the third **parallel** construct, the **order** clause indicates that the loops are not reproducible, overriding the default reproducibility prescribed by the specified static schedule. Consequentially, the **nowait** clause on the first worksharing-loop construct should not be used to ensure that the data dependences are preserved by the execution.

C / C++

Example reproducible.2.c (**omp_5.1**)

```
S-1   #include <stdio.h>
S-2
S-3   int main()
S-4   {
S-5      const int n = 1000;
S-6      int v[n], u[n];
S-7
S-8      #pragma omp parallel
S-9      {
S-10        // reproducible schedules are used for the following two constructs
S-11        #pragma omp for order(reproducible: concurrent) nowait
S-12        for (int i = 0; i < n; i++) {
S-13           u[i] = i;
S-14           v[i] = i;
S-15        }
S-16        #pragma omp for order(reproducible: concurrent)
S-17        for (int i = 0; i < n; i++) {
```

```
S-18            v[i]  += u[i] * u[i];
S-19         }
S-20      }
S-21
S-22      #pragma omp parallel
S-23      {
S-24         // static schedules preserve data dependences between the loops
S-25         #pragma omp for schedule(static) order(concurrent) nowait
S-26         for (int i = 0; i < n; i++) {
S-27            u[i] = i;
S-28            v[i] = i;
S-29         }
S-30         #pragma omp for schedule(static) order(concurrent)
S-31         for (int i = 0; i < n; i++) {
S-32            v[i]  += u[i] * u[i];
S-33         }
S-34      }
S-35
S-36      #pragma omp parallel
S-37      {
S-38         // the default reproducibility by the static schedule is not
S-39         // preserved due to the unconstrained order clause.
S-40         // use of nowait here could result in data race.
S-41         #pragma omp for schedule(static) order(unconstrained: concurrent)
S-42         for (int i = 0; i < n; i++) {
S-43            u[i] = i;
S-44            v[i] = i;
S-45         }
S-46         #pragma omp for schedule(static) order(unconstrained: concurrent)
S-47         for (int i = 0; i < n; i++) {
S-48            v[i]  += u[i] * u[i];
S-49         }
S-50      }
S-51
S-52      return 0;
S-53   }
```

C / C++

Fortran

1 *Example reproducible.2.f90* (`omp_5.1`)

```
S-1    program main
S-2       implicit none
S-3       integer, parameter :: n = 1000
S-4       integer :: v(n), u(n)
S-5       integer :: i
S-6
```

```
S-7       !$omp parallel
S-8          !! reproducible schedules are used the following two constructs
S-9          !$omp do order(reproducible: concurrent)
S-10         do i = 1, n
S-11            u(i) = i
S-12            v(i) = i
S-13         end do
S-14         !$omp end do nowait
S-15         !$omp do order(reproducible: concurrent)
S-16         do i = 1, n
S-17            v(i) = v(i) + u(i) * u(i)
S-18         end do
S-19      !$omp end parallel
S-20
S-21      !$omp parallel
S-22         !! static schedules preserve data dependences between the loops
S-23         !$omp do schedule(static) order(concurrent)
S-24         do i = 1, n
S-25            u(i) = i
S-26            v(i) = i
S-27         end do
S-28         !$omp end do nowait
S-29         !$omp do schedule(static) order(concurrent)
S-30         do i = 1, n
S-31            v(i) = v(i) + u(i) * u(i)
S-32         end do
S-33      !$omp end parallel
S-34
S-35      !$omp parallel
S-36         !! the default reproducibility by the static schedule is not
S-37         !! preserved due to the unconstrained order clause.
S-38         !! use of nowait here could result in data race.
S-39         !$omp do schedule(static) order(unconstrained: concurrent)
S-40         do i = 1, n
S-41            u(i) = i
S-42            v(i) = i
S-43         end do
S-44         !$omp do schedule(static) order(unconstrained: concurrent)
S-45         do i = 1, n
S-46            v(i) = v(i) + u(i) * u(i)
S-47         end do
S-48      !$omp end parallel
S-49
S-50   end program
```

Fortran

12.14 `interop` Construct

The **interop** construct allows OpenMP to interoperate with foreign runtime environments. In the example below, asynchronous cuda memory copies and a *cublasDaxpy* routine are executed in a cuda stream. Also, an asynchronous target task execution (having a **nowait** clause) and two explicit tasks are executed through OpenMP directives. Scheduling dependences (synchronization) are imposed on the foreign stream and the OpenMP tasks through **depend** clauses.

First, an interop object, *obj*, is initialized for synchronization by including the **targetsync** *interop-type* in the interop **init** clause (**init(targetsync:** *obj***)**). The object provides access to the foreign runtime. The **depend** clause provides a dependence behavior for foreign tasks associated with a valid object.

Next, the **omp_get_interop_int** routine is used to extract the foreign runtime id (**omp_ipr_fr_id**), and a test in the next statement ensures that the cuda runtime (**omp_ifr_cuda**) is available.

Within the block for executing the *cublasDaxpy* routine, a stream is acquired with the **omp_get_interop_ptr** routine, which returns a cuda stream (*s*). The stream is included in the cublas handle, and used directly in the asynchronous memory routines. The following **interop** construct, with the **destroy** clause, ensures that the foreign tasks have completed.

C / C++

Example interop.1.c (**omp_5.1**)

```
S-1   #include <omp.h>
S-2   #include <stdio.h>
S-3   #include <stdlib.h>
S-4   #include <cublas_v2.h>
S-5   #include <cuda_runtime_api.h>
S-6
S-7   #define N 16384
S-8
S-9   void myVectorSet(int n, double s, double *x)
S-10  {
S-11     for(int i=0; i<n; ++i) x[i] = s*(i+1);
S-12  }
S-13  void myDaxpy(int n, double s, double *x, double *y)
S-14  {
S-15     for(int i=0; i<n; ++i) y[i] = s*x[i]+y[i];
S-16  }
S-17  void myDscal(int n, double s, double *x)
S-18  {
S-19     for(int i=0; i<n; ++i) x[i] = s*x[i];
S-20  }
S-21
S-22
```

```
S-23    int main(){
S-24      const  double scalar=2.0;
S-25      double *x, *y, *d_x, *d_y;
S-26      int    dev;
S-27
S-28      omp_interop_t obj=omp_interop_none;
S-29      intptr_t  type;
S-30
S-31      // Async Memcpy requires pinned memory
S-32      cudaMallocHost( (void**)&x,   N*sizeof(double) );
S-33      cudaMallocHost( (void**)&y,   N*sizeof(double) );
S-34      cudaMalloc(     (void**)&d_x, N*sizeof(double) );
S-35      cudaMalloc(     (void**)&d_y, N*sizeof(double) );
S-36
S-37      dev = omp_get_default_device();
S-38      omp_target_associate_ptr(&x[0], d_x, sizeof(double)*N, 0, dev);
S-39      omp_target_associate_ptr(&y[0], d_y, sizeof(double)*N, 0, dev);
S-40
S-41      #pragma omp target nowait depend(out: x[0:N]) \
S-42                           map(from: x[0:N]) device(dev)
S-43      myVectorSet(N,  1.0, x);
S-44
S-45      #pragma omp task depend(out: y[0:N])
S-46      myVectorSet(N, -1.0, y);
S-47
S-48      // get obj for syncing
S-49      #pragma omp interop init(targetsync: obj) device(dev) \
S-50                      depend(in: x[0:N]) depend(inout: y[0:N])
S-51
S-52                                                //foreign rt id and string name
S-53      int        id = (int  )omp_get_interop_int(obj, omp_ipr_fr_id,   NULL);
S-54      char* rt_name = (char*)omp_get_interop_str(obj, omp_ipr_fr_name, NULL);
S-55
S-56      if(obj != omp_interop_none && id == omp_ifr_cuda) {
S-57
S-58        printf(" OpenMP working with %s runtime to execute cublas daxpy.\n",
S-59               rt_name);
S-60        cublasHandle_t handle;
S-61        int rc;
S-62        cublasCreate(&handle);
S-63
S-64        cudaStream_t s=
S-65            (cudaStream_t)omp_get_interop_ptr(obj, omp_ipr_targetsync, &rc);
S-66        if(rc != omp_irc_success) {
S-67           fprintf(stderr,"ERROR: Failed to get %s stream, rt error= %d.\n",
S-68                   rt_name, rc);
S-69           if(rc == omp_irc_no_value)
```

```
S-70              fprintf(stderr,
S-71                      "Parameters valid, no meaningful value available.");
S-72           exit(1);
S-73        }
S-74
S-75        cublasSetStream( handle,s );
S-76        cudaMemcpyAsync( d_x, x, N*sizeof(double),
S-77                         cudaMemcpyHostToDevice, s );
S-78        cudaMemcpyAsync( d_y, y, N*sizeof(double),
S-79                         cudaMemcpyHostToDevice, s );
S-80        cublasDaxpy(     handle, N, &scalar, &d_x[0], 1, &d_y[0], 1 ) ;
S-81        cudaMemcpyAsync( y, d_y, N*sizeof(double),
S-82                         cudaMemcpyDeviceToHost, s );
S-83
S-84      } else {     // Execute as OpenMP offload.
S-85
S-86        printf(" Notice: Offloading myDaxpy to perform daxpy calculation.\n");
S-87
S-88        #pragma omp target depend(inout: y[0:N]) depend(in: x[0:N]) nowait \
S-89                           map(to: x[0:N]) map(tofrom: y[0:N]) device(dev)
S-90        myDaxpy(N, scalar, x, y);
S-91
S-92      }
S-93
S-94       // This also ensures foreign tasks complete.
S-95      #pragma omp interop destroy(obj) nowait depend(out: y[0:N])
S-96
S-97      #pragma omp target depend(inout: x[0:N])
S-98      myDscal(N, scalar, x);
S-99
S-100     #pragma omp taskwait
S-101     printf("(-1:-16384) %f:%f\n", y[0], y[N-1]);
S-102     printf("(-2:-32768) %f:%f\n", x[0], x[N-1]);
S-103
S-104   }
```

C / C++

12.15 Utilities

This section contains examples of utility routines and features.

12.15.1 Timing Routines

The **omp_get_wtime** routine can be used to measure the elapsed wall clock time (in seconds) of code execution in a program. The routine is thread safe and can be executed by multiple threads concurrently. The precision of the timer can be obtained by a call to the **omp_get_wtick** routine. The following example shows a use case.

C / C++

Example get_wtime.1.c (**pre_omp_3.0**)

```
S-1   #include <stdio.h>
S-2   #include <unistd.h>
S-3   #include <omp.h>
S-4
S-5   void work_to_be_timed()
S-6   {
S-7     sleep(2);
S-8   }
S-9
S-10  int main()
S-11  {
S-12    double start, end;
S-13
S-14    start = omp_get_wtime();
S-15    work_to_be_timed();     // any parallel or serial codes
S-16    end = omp_get_wtime();
S-17
S-18    printf("Work took %f seconds\n", end - start);
S-19    printf("Precision of the timer is %f (sec)\n", omp_get_wtick());
S-20    return 0;
S-21  }
```

C / C++

Fortran

Example get_wtime.1.f90 (**pre_omp_3.0**)

```
S-1   subroutine work_to_be_timed
S-2     use, intrinsic :: iso_c_binding, only: c_int
S-3     interface
S-4        subroutine fsleep(sec) bind(C, name="sleep")
S-5          import c_int
S-6          integer(c_int), value :: sec
S-7        end subroutine
S-8     end interface
S-9     call fsleep(2)
S-10  end subroutine
S-11
S-12  program do_work
S-13    use omp_lib
S-14    implicit none
S-15    double precision :: start, end
S-16
S-17    start = omp_get_wtime()
S-18    call work_to_be_timed    ! any parallel or serial codes
S-19    end = omp_get_wtime()
S-20
S-21    print *, "Work took", end - start, "seconds"
S-22    print *, "Precision of the timer is", omp_get_wtick(), "(sec)"
S-23  end program
```

Fortran

12.15.2 Environment Display

The OpenMP version number and the values of ICVs associated with the relevant environment variables can be displayed at runtime by setting the **OMP_DISPLAY_ENV** environment variable to either TRUE or VERBOSE. The information is displayed once by the runtime.

A more flexible or controllable approach is to call the **omp_display_env** API routine at any desired point of a code to display the same information. This OpenMP 5.1 API routine takes a single *verbose* argument. A value of 0 or **.false.** (for C/C++ or Fortran) indicates the required OpenMP ICVs associated with environment variables be displayed, and a value of 1 or **.true.** (for C/C++ or Fortran) will include vendor-specific ICVs that can be modified by environment variables.

The following example illustrates the conditional execution of the API **omp_display_env** routine. Typically it would be invoked in various debug modes of an application. An important use case is to have a single MPI process (e.g., rank = 0) of a hybrid (MPI+OpenMP) code execute the

routine, instead of all MPI processes, as would be done by setting the **OMP_DISPLAY_ENV** to TRUE or VERBOSE.

C / C++

Example display_env.1.c (**omp_5.1**)

```
S-1   #include <omp.h>
S-2
S-3   //implementers: customize debug routines for app debugging
S-4   int debug(){ return 1; }
S-5   int debug_omp_verbose(){ return 0; }
S-6
S-7   int main()
S-8   {
S-9      if( debug() ) omp_display_env( debug_omp_verbose() );
S-10     // ...
S-11     return 0;
S-12  }
```

C / C++

Fortran

Example display_env.1.f90 (**omp_5.1**)

```
S-1   !implementers: customize debug routines for app debugging
S-2   function  debug()
S-3     logical :: debug
S-4     debug = .true.
S-5   end function
S-6
S-7   function debug_omp_verbose()
S-8     logical :: debug_omp_verbose
S-9     debug_omp_verbose = .false.
S-10  end function
S-11
S-12  program display_omp_environment
S-13    use omp_lib
S-14    logical :: debug, debug_omp_verbose
S-15
S-16    if( debug() ) call omp_display_env( debug_omp_verbose() )
S-17    !! ...
S-18  end program
```

Fortran

A sample output from the execution of the code might look like:

```
OPENMP DISPLAY ENVIRONMENT BEGIN
   _OPENMP='202011'
  [host] OMP_AFFINITY_FORMAT='(null)'
  [host] OMP_ALLOCATOR='omp_default_mem_alloc'
  [host] OMP_CANCELLATION='FALSE'
  [host] OMP_DEFAULT_DEVICE='0'
  [host] OMP_DISPLAY_AFFINITY='FALSE'
  [host] OMP_DISPLAY_ENV='FALSE'
  [host] OMP_DYNAMIC='FALSE'
  [host] OMP_MAX_ACTIVE_LEVELS='1'
  [host] OMP_MAX_TASK_PRIORITY='0'
  [host] OMP_NESTED: deprecated; max-active-levels-var=1
  [host] OMP_NUM_THREADS: value is not defined
  [host] OMP_PLACES: value is not defined
  [host] OMP_PROC_BIND: value is not defined
  [host] OMP_SCHEDULE='static'
  [host] OMP_STACKSIZE='4M'
  [host] OMP_TARGET_OFFLOAD=DEFAULT
  [host] OMP_THREAD_LIMIT='0'
  [host] OMP_TOOL='enabled'
  [host] OMP_TOOL_LIBRARIES: value is not defined
OPENMP DISPLAY ENVIRONMENT END
```

12.15.3 `error` Directive

The **`error`** directive provides a consistent method for C, C++, and Fortran to emit a **`fatal`** or **`warning`** message at **`compilation`** or **`execution`** time, as determined by a **`severity`** or an **`at`** clause, respectively. When **`severity(fatal)`** is present, the compilation or execution is aborted. Without any clauses the default behavior is as if **`at(compilation)`** and **`severity(fatal)`** were specified.

The C, C++, and Fortran examples below show all the cases for reporting messages.

C / C++

Example error.1.c (**`omp_5.2`**)

```
S-1  #include <stdio.h>
S-2  #include   <omp.h>
S-3
S-4  int main(){
S-5
S-6  #pragma omp metadirective \
S-7              when(implementation={vendor(gnu)}: nothing )    \
S-8              otherwise(error at(compilation) severity(fatal) \
```

```
                        message("GNU compiler required."))

  if( omp_get_num_procs() < 3 ){
    #pragma omp error at(execution) severity(fatal) \
                        message("3 or more procs required.")
  }

  #pragma omp parallel master
  {
    // Give notice about master deprecation at compile time and run time.
    #pragma omp error at(compilation) severity(warning) \
                        message("Notice: master is deprecated.")
    #pragma omp error at(execution) severity(warning) \
                        message("Notice: masked used next release.")

     printf(" Hello from thread number 0.\n");
  }

}
```

C / C++

Fortran

Example error.1.f90 (`omp_5.2`)

```
program main
use omp_lib

!$omp  metadirective  &
!$omp&      when( implementation={vendor(gnu)}: nothing    ) &
!$omp&      otherwise( error at(compilation) severity(fatal) &
!$omp&                 message( "GNU compiler required." ) )


if( omp_get_num_procs() < 3 ) then
   !$omp  error at(execution) severity(fatal) &
   !$omp&       message("3 or more procs required.")
endif

  !$omp parallel master

!! Give notice about master deprecation at compile time and run time.
  !$omp  error at(compilation) severity(warning) &
  !$omp&       message("Notice: master is deprecated.")
  !$omp  error at(execution) severity(warning) &
  !$omp&       message("Notice: masked to be used in next release.")

  print*," Hello from thread number 0."
```

```
S-24
S-25        !$omp end parallel master
S-26
S-27    end program
```

Fortran

This page intentionally left blank

13 OMPT Interface

OMPT defines mechanisms and an API for interfacing with tools in the OpenMP program. The OMPT API provides the following functionality:

- examines the state associated with an OpenMP thread
- interprets the call stack of an OpenMP thread
- receives notification about OpenMP events
- traces activity on OpenMP target devices
- assesses implementation-dependent details
- controls a tool from an OpenMP application

The following sections will illustrate basic mechanisms and operations of the OMPT API.

13.1 OMPT Start

There are three steps an OpenMP implementation takes to activate a tool. This section explains how the tool and an OpenMP implementation interact to accomplish tool activation.

Step 1. *Determine Whether to Initialize*

A tool is activated by the OMPT interface when it returns a non-**NULL** pointer to an **ompt_start_tool_result_t** structure on a call to **ompt_start_tool** by the OpenMP implementation. There are three ways that a tool can provide a definition of **ompt_start_tool** to an OpenMP implementation: (1) Statically linking the tool's definition of **ompt_start_tool** into an OpenMP application. (2) Introducing a dynamically linked library that includes the tool's definition of **ompt_start_tool** into the application's address space. (3) Providing the name of a dynamically linked library appropriate for the architecture and operating system used by the application in the *tool-libraries-var* ICV.

Step 2. *Initializing a First-Party tool*

If a tool-provided implementation of **ompt_start_tool** returns a non-**NULL** pointer to an **ompt_start_tool_result_t** structure, the OpenMP implementation will invoke the tool initializer specified in this structure prior to the occurrence of any OpenMP event.

Step 3. *Monitoring Activity on the Host*

To monitor execution of an OpenMP program on the host device, a tool's initializer must register to receive notification of events that occur as an OpenMP program executes. A tool can register callbacks for OpenMP events using the runtime entry point known as **ompt_set_callback**, which has the following possible return codes: **ompt_set_error**, **ompt_set_never**, **ompt_set_impossible**, **ompt_set_sometimes**, **ompt_set_sometimes_paired**, **ompt_set_always**.

If the **ompt_set_callback** runtime entry point is called outside a tool's initializer, registration of supported callbacks may fail with a return code of **ompt_set_error**. All callbacks registered with **ompt_set_callback** or returned by **ompt_get_callback** use the dummy type signature **ompt_callback_t**. While this is a compromise, it is better than providing unique runtime entry points with precise type signatures to set and get the callback for each unique runtime entry point type signature.

———————-

To use the OMPT interface a tool must provide a globally-visible implementation of the **ompt_start_tool** function. The function returns a pointer to an **ompt_start_tool_result_t** structure that contains callback pointers for tool initialization and finalization as well as a data word, *tool_data*, that is to be passed by reference to these callbacks. A **NULL** return indicates the tool will not use the OMPT interface. The runtime execution of **ompt_start_tool** is triggered by the first OpenMP directive or OpenMP API routine call.

In the example below, the user-provided **ompt_start_tool** function performs a check to make sure the runtime OpenMP version that OMPT supports (provided by the *omp_version* argument) is identical to the OpenMP implementation (compile-time) version. Also, a **NULL** is returned to indicate that the OMPT interface is not used (no callbacks and tool data are specified).

Note: The *omp-tools.h* file is included.

C / C++

Example ompt_start.1.c (**omp_5.0**)

```
S-1   #include <stdio.h>
S-2   #include <omp.h>
S-3   #include <omp-tools.h>
S-4
S-5   ompt_start_tool_result_t *ompt_start_tool(
S-6   unsigned int omp_version,
S-7   const char *runtime_version
S-8   ){
S-9     if(omp_version != _OPENMP)
S-10      printf("Warning: OpenMP runtime version (%i) "
S-11             "does not match the compile time version (%i)"
S-12             " for runtime identifying as %s\n",
S-13             omp_version, _OPENMP, runtime_version);
S-14    // Returning NULL will disable this as an OMPT tool,
S-15    // allowing other tools to be loaded
S-16    return NULL;
S-17  }
S-18
S-19  int main(void){
S-20    printf("Running with %i threads\n", omp_get_max_threads());
S-21    return 0;
S-22  }
```

C / C++

This page intentionally left blank

A Feature Deprecations and Updates in Examples

Deprecation of features began in OpenMP 5.0. Examples that use a deprecated feature have been updated with an equivalent replacement feature.

Table A.1 summarizes deprecated features and their replacements in each version. Affected examples are updated accordingly and listed in Section A.1.

TABLE A.1: Deprecated Features and Their Replacements

Version	Deprecated Feature	Replacement
6.0	**declare reduction**(*reduction-id*: *typename-list*: *combiner*)	**declare reduction**(*reduction-id*: *typename-list*) **combiner**(*combiner-exp*)
5.2	**default** clause on metadirectives	**otherwise** clause
5.2	delimited **declare target** directive for C/C++	**begin declare target** directive
5.2	**to** clause on **declare target** directive	**enter** clause
5.2	non-argument **destroy** clause on **depobj** construct	**destroy**(*argument*)
5.2	**allocate** directive for Fortran **ALLOCATE** statements	**allocators** directive
5.2	**depend** clause on **ordered** construct	**doacross** clause
5.2	**linear**(*modifier(list): linear-step*) clause	**linear**(*list:* **step**(*linear-step*), *modifier*) clause
5.1	**master** construct	**masked** construct
5.1	**master** affinity policy	**primary** affinity policy
5.0	**omp_lock_hint_*** constants	**omp_sync_hint_*** constants

These replacements appear in examples that illustrate, otherwise, earlier features. When using a compiler that is compliant with a version prior to the indicated version, the earlier form of an example for a previous version is listed as a reference.

A.1 Updated Examples for Different Versions

The following tables list the updated examples for different versions as a result of feature deprecation. The *Earlier Version* column of the tables shows the version tag of the earlier version. It also shows the prior name of an example when it has been renamed.

Table A.2 lists the updated examples for features deprecated in OpenMP 6.0 in the Examples Document Version 6.0. The *Earlier Version* column of the table lists the earlier version tags of the examples that can be found in the Examples Document Version 5.2.

TABLE A.2: Updated Examples for Features Deprecated in Version 6.0

Example Name	Earlier Version	Feature Updated
udr.1.c, *f90*	4.0	*combiner* expression in **`declare reduction`** directive changed to use **`combiner`** clause
udr.2.c, *f90*	4.0	
udr.3.c, *f90*	4.0	
udr.4.f90	4.0	
udr.5.cpp	4.0	
udr.6.cpp	4.0	

Table A.3 lists the updated examples for features deprecated in OpenMP 5.2 in the Examples Document Version 5.2. The *Earlier Version* column of the table lists the earlier version tags of the examples that can be found in the Examples Document Version 5.1.

TABLE A.3: Updated Examples for Features Deprecated in Version 5.2

Example Name	Earlier Version	Feature Updated
error.1.c, *f90*	5.1	**`default`** clause on metadirectives replaced with **`otherwise`** clause
metadirective.1.c, *f90*	5.0	
metadirective.2.c, *f90*	5.0	
metadirective.3.c, *f90*	5.0	
metadirective.4.c, *f90*	5.1	
target_ptr_map.4.c	5.1	
target_ptr_map.5.c, *f90*	5.1	
array_shaping.1.f90	5.0	**`to`** clause on **`declare target`** directive replaced with **`enter`** clause
target_reverse_offload.7.c	5.0	
target_task_reduction.1.c, *f90*	5.1	
target_task_reduction.2a.c, *f90*	5.0	
target_task_reduction.2b.c, *f90*	5.1	
array_shaping.1.c	5.0	delimited **`declare target`** directive replaced with
async_target.1.c	4.0	

table continued on next page

table continued from previous page

Example Name	Earlier Version	Feature Updated
async_target.2.c	4.0	**begin declare target** directive for C/C++
declare_target.1.c	4.0	
declare_target.2c.cpp	4.0	
declare_target.3.c	4.0	
declare_target.4.c	4.0	
declare_target.5.c	4.0	
declare_target.6.c	4.0	
declare_variant.1.c	5.0	
device.1.c	4.0	
metadirective.3.c	5.0	
target_ptr_map.2.c	5.0	
target_ptr_map.3a.c	5.0	
target_ptr_map.3b.c	5.0	
target_struct_map.1.c	5.0	
target_struct_map.2.cpp	5.0	
target_struct_map.3.c	5.0	
target_struct_map.4.c	5.0	
doacross.1.c, *f90*	4.5	**depend** clause on **ordered** construct replaced with **doacross** clause
doacross.2.c, *f90*	4.5	
doacross.3.c, *f90*	4.5	
doacross.4.c, *f90*	4.5	
linear_modifier.1.cpp, *f90*	4.5	modifier syntax change for **linear** clause on **declare simd** directive
linear_modifier.2.cpp, *f90*	4.5	
linear_modifier.3.c, *f90*	4.5	
allocators.1.f90	5.0	**allocate** directive replaced with **allocators** directive for Fortran **allocate** statements
depobj.1.c, *f90*	5.0	argument added to **destroy** clause on **depobj** construct

Table A.4 lists the updated examples for features deprecated in OpenMP 5.1 in the Examples Document Version 5.1. The *Earlier Version* column of the table lists the earlier version tags and prior names of the examples that can be found in the Examples Document Version 5.0.1.

TABLE A.4: Updated Examples for Features Deprecated in Version 5.1

Example Name	Earlier Version	Feature Updated
affinity.5.c, *f*	4.0	**master** affinity policy replaced with **primary** policy
async_target.3.c, *f90*	5.0	**master** construct replaced with **masked** construct
cancellation.2.c, *f90*	4.0	
copyprivate.2.c, *f*	3.0	
fort_sa_private.5.f	3.0	
lock_owner.1.c, *f*	3.0	
masked.1.c, *f*	3.0: *master.1.c*, *f*	
parallel_masked_taskloop.1.c, *f90*	5.0: *parallel_master_taskloop.1.c*, *f90*	
reduction.6.c, *f*	3.0	
target_task_reduction.1.c, *f90*	5.0	
target_task_reduction.2b.c, *f90*	5.0	
taskloop_simd_reduction.1.c, *f90*	5.0	
task_detach.1.c, *f90*	5.0	

Table A.5 lists the updated examples for features deprecated in OpenMP 5.0 in the Examples Document Version 5.1. The *Earlier Version* column of the table lists the earlier version tags of the examples that can be found in the Examples Document Version 5.0.1.

TABLE A.5: Updated Examples for Features Deprecated in Version 5.0

Example Name	Earlier Version	Feature Updated
critical.2.c, *f*	4.5	**omp_lock_hint_*** constants replaced with **omp_sync_hint_*** constants
init_lock_with_hint.1.cpp, *f*	4.5	

B Document Revision History

B.1 Changes from 5.2.2 to 6.0

- General changes:
  - Added a set of structured LaTeX environments for specifying language-dependent text. This allows extracting language-specific content of the Examples document. Refer to the content of v6.0/Contributions.md for details.
- Added the following examples for the 6.0 features:
  - **`omp::decl`** attribute for declarative directives in C/C++ (Section 2.2 on page 5)
  - **`transparent`** clause on the **`task`** construct to enable dependences between non-sibling tasks (Section 5.3.10 on page 126)
  - Task dependences for **`taskloop`** construct (Section 5.9 on page 144)
  - **`num_threads`** clause that appears inside **`target`** region (Section 6.16.7 on page 253)
  - **`nowait`** clause with argument on the **`target`** construct to control deferment of target task (Section 6.17.4 on page 263)
  - Traits for specifying devices (Section 6.19 on page 282)
  - **`apply`** clause with modifier argument to support selective loop transformations (Section 8.4 on page 327)
  - Reduction on private variables in a **`parallel`** region (Section 10.10.7 on page 445)
  - **`induction`** clause (Section 10.11.1 on page 460) and user-defined induction (Section 10.11.2 on page 463)
  - **`init_complete`** clause for **`scan`** directive to support initialization phase in scan operation (Section 10.12 on page 465)
  - **`assume`** construct with **`no_openmp`** and **`no_parallelism`** clauses (Section 12.1 on page 510)
  - **`num_threads`** clause with a list (Section 12.3.1 on page 519)
  - **`dispatch`** construct to control variant substitution for a procedure call (Section 12.8 on page 560)
- Other changes:
  - Changed attribute specifier as a directive form from C++ only to C/C++ (Section 2 on page 3)

- Added missing **include <omp.h>** in Example *atomic.4.c* and **use omp_lib** in Example *atomic.4.f90* (Section 9.7 on page 360)
- Fixed the function declaration order for variant functions in Examples *selector_scoring.[12].c* and Fortran pointer initialization in Example *selector_scoring.2.f90* (Section 12.7.3 on page 552)
- Replaced the deprecated use of *combiner-exp* in **declare reduction** directive with **combiner** clause (Section 10.10.8 on page 450 and Section A.1 on page 600)
- Fixed the initialization of Fortran pointers in Example *cancellation.2.f90* and changed to use **atomic write** for performing atomic writes (Section 12.5 on page 526)
- Added missing **declare target** directive for external procedure called inside **target** region in Example *requires.1.f90* (Section 12.6 on page 531)

B.2 Changes from 5.2.1 to 5.2.2

- To improve the style of the document, a set of macros was introduced and consistently used for language keywords, names, concepts, and user codes in the text description of the document. Refer to the content of v5.2.2/Contributions.md for details.
- Added the following examples:
  - Orphaned and nested **loop** constructs (Section 3.15 on page 49)
  - **all** variable category for the **defaultmap** clause (Section 6.2 on page 160)
  - **target update** construct using a custom mapper (Section 6.13.3 on page 221)
  - **indirect** clause for indirect procedure calls in a **target** region (Section 6.14.2 on page 225)
  - **omp_target_memcpy_async** routine with depend object (Section 6.18.5 on page 272)
  - Synchronization hint for atomic operation (Section 9.7 on page 360)
  - Implication of passing shared variable to a procedure in Fortran (Section 10.7 on page 409)
  - Assumption directives for providing additional information about program properties (Section 12.1 on page 510)
  - Mapping behavior of scalars, pointers, references (C++) and associate names (Fortran) when unified shared memory is required (Section 12.6 on page 531)
  - **begin declare variant** paired with **end declare variant** example to show use of nested declare variant directives (Section 12.7.1 on page 534)
  - Explicit scoring in context selectors (Section 12.7.3 on page 552)

- Miscellaneous changes:
  - Included a general statement in Introduction about the number of threads used throughout the examples document (Section 1.1 on page 2)
  - Clarified the mapping of virtual functions in **`target`** regions (Section 6.8 on page 191)
  - Added missing **`declare target`** directive for procedures called inside **`target`** region in *Examples declare_mapper.1.f90* (Section 6.10 on page 197), *target_reduction.*.f90* (Section 10.10.3 on page 427), and *target_task_reduction.*.f90* (Section 10.10.4 on page 431)
  - Added missing **`end target`** directive in *Example declare_mapper.3.f90* (Section 6.10 on page 197)
  - Removed example for **`flush`** without a list from Synchronization since the example is confusing and the use of **`flush`** is already covered in other examples (Section 9 on page 341)
  - *declare variant Directive* and *Metadirective* sections were moved to subsections in the new *Context-based Variant Selection* section, with a section introduction on context selectors. (Section 12.7 on page 533)
  - Fixed a typo ('**`for`**' → '**`do`**') in *Example metadirective.4.f90* (Section 12.7.2 on page 541)

B.3 Changes from 5.2 to 5.2.1

- General changes:
  - Updated source metadata tags for all examples to use an improved form (see v5.2.1/Contributions.md)
  - Explicitly included the version tag (**`pre_omp_3.0`**) in those examples that did not contain a version tag previously
- Added the following examples for the 5.2 features:
  - **`uses_allocators`** clause for the use of allocators in **`target`** regions (Section 11.2 on page 492)
- Added the following examples for the 5.1 features:
  - The **`inoutset`** dependence type (Section 5.3.4 on page 107)
  - Atomic compare and capture (Section 9.5 on page 354)
- Added the following examples for the 5.0 features:
  - **`declare target`** directive with **`device_type(nohost)`** clause (Section 6.14.7 on page 239)
  - **`omp_pause_resource`** and **`omp_pause_resource_all`** routines (Section 12.12 on page 576)

- Miscellaneous fixes:
  - Cast to implementation-defined enum type **omp_event_handle_t** now uses **uintptr_t** (not **void ***) in *Example task_detach.2.c* (Section 5.4 on page 130)
  - Moved Fortran **requires** directive into program main (`rev_off`), the program unit, in *Example target_reverse_offload.7.f90* (Section 6.1.6 on page 158)
  - Fixed an inconsistent use of mapper in *Example target_mapper.3.f90* (Section 6.10 on page 197)
  - Added a missing semicolon at end of `XOR1` class definition in *Example declare_target.2a.cpp* (Section 6.14.3 on page 227)
  - Fixed the placement of **declare simd** directive in *Examples linear_modifier.*.f90* (Section 7.4 on page 296) and added a general statement about where a Fortran declarative directive can appear (Section 2 on page 3)
  - Fixed mismatched argument list in *Example fort_sa_private.5.f* (Section 10.6 on page 406)
  - Moved the placement of **declare target enter** directive after function declaration (Section 10.10.4 on page 431)
  - Fixed an incorrect use of **omp_in_parallel** routine in *Example metadirective.4* (Section 12.7.2 on page 541)
  - Fixed an incorrect value for **at** clause (Section 12.15.3 on page 591)

B.4 Changes from 5.1 to 5.2

- General changes:
  - Included a description of the semantics for OpenMP directive syntax (see Section 2 on page 3)
  - Reorganized the Introduction Chapter and moved the Feature Deprecation Chapter to Appendix A
  - Included a list of examples that were updated for feature deprecation and replacement in each version (see Appendix A.1)
  - Added Index entries
- Updated the examples for feature deprecation and replacement in OpenMP 5.2. See Table A.1 and Table A.3 for details.
- Added the following examples for the 5.2 features:
  - Mapping class objects with virtual functions (Section 6.8 on page 191)
  - **allocators** construct for Fortran **allocate** statement (Section 11.2 on page 492)

  - Behavior of reallocation of variables through OpenMP allocator in Fortran (Section 11.2 on page 492)

- Added the following examples for the 5.1 features:
  - Clarification of optional **end** directive for strictly structured block in Fortran (Section 2.4 on page 10)
  - **filter** clause on **masked** construct (Section 3.14 on page 47)
  - **omp_all_memory** reserved locator for specifying task dependences (Section 5.3.9 on page 123)
  - Behavior of Fortran allocatable variables in **target** regions (Section 6.5 on page 181)
  - Device memory routines in Fortran (Section 6.18.5 on page 272)
  - Partial tiles from **tile** construct (Section 8.2 on page 309)
  - Fortran associate names and selectors in **target** region (Section 10.16 on page 478)
  - **allocate** directive for variable declarations and **allocate** clause on **task** constructs (Section 11.2 on page 492)
  - Controlling concurrency and reproducibility with **order** clause (Section 12.13 on page 579)
- Added other examples:
  - Using lambda expressions with **target** constructs (Section 6.15 on page 241)
  - Target memory and device pointer routines (Section 6.18.5 on page 272)
  - Examples to illustrate the ordering properties of the *flush* operation (Section 11.1 on page 482)
  - User selector in the **metadirective** directive (Section 12.7.2 on page 541)

B.5 Changes from 5.0.1 to 5.1

- General changes:
  - Replaced **master** construct example with equivalent **masked** construct example (Section 3.14 on page 47)
  - Primary thread is now used to describe thread number 0 in the current team
  - **primary** thread affinity policy is now used to specify that every thread in the team is assigned to the same place as the primary thread (Section 4.1.3 on page 65)
  - The **omp_lock_hint_*** constants have been renamed **omp_sync_hint_*** (Section 9.1 on page 342, Section 9.12 on page 382)
- Added the following new chapters:

- Deprecated Features (on page 599)
- Directive Syntax (Section 2 on page 3)
- Loop Transformations (Section 8 on page 305)
- OMPT Interface (Section 13 on page 595)

- Added the following examples for the 5.1 features:
  - OpenMP directives in C++ *attribute* specifiers (Section 2.2 on page 5)
  - Directive syntax adjustment to allow Fortran **BLOCK** ... **END BLOCK** as a structured block (Section 2.4 on page 10)
  - **omp_target_is_accessible** API routine (Section 6.3 on page 166)
  - Fortran allocatable array mapping in **target** regions (Section 6.5 on page 181)
  - **begin declare target** (with **end declare target**) directive (Section 6.14.3 on page 227)
  - **tile** construct (Section 8.1 on page 305)
  - **unroll** construct (Section 8.3 on page 317)
  - Reduction with the **scope** construct (Section 10.10.6 on page 443)
  - **metadirective** directive with dynamic **condition** selector (Section 12.7.2 on page 541)
  - **interop** construct (Section 12.14 on page 585)
  - Environment display with the **omp_display_env** routine (Section 12.15.2 on page 589)
  - **error** directive (Section 12.15.3 on page 591)
- Included additional examples for the 5.0 features:
  - **collapse** clause for non-rectangular loop nest (Section 3.8 on page 31)
  - **detach** clause for tasks (Section 5.4 on page 130)
  - Pointer attachment for a structure member (Section 6.4 on page 175)
  - Host and device pointer association with the **omp_target_associate_ptr** routine (Section 6.18.4 on page 269)
  - Sample code on activating the tool interface (Section 13.1 on page 596)
- Added other examples:
  - The **omp_get_wtime** routine (Section 12.15.1 on page 588)

B.6 Changes from 5.0.0 to 5.0.1

- Added version tags (**omp**_*x.y*) in example labels and the corresponding source codes for all examples that feature OpenMP 3.0 and later.
- Included additional examples for the 5.0 features:
  - Extension to the **defaultmap** clause (Section 6.2 on page 160)
  - Transferring noncontiguous data with the **target update** directive in Fortran (Section 6.9 on page 194)
  - **conditional** modifier for the **lastprivate** clause (Section 10.9 on page 412)
  - **task** modifier for the **reduction** clause (Section 10.10.2 on page 422)
  - Reduction on combined target constructs (Section 10.10.3 on page 427)
  - Task reduction with **target** constructs (Section 10.10.4 on page 431)
  - **scan** directive for returning the *prefix sum* of a reduction (Section 10.12 on page 465)
- Included additional examples for the 4.x features:
  - Dependence for undeferred tasks (Section 5.3.9 on page 123)
  - **ref**, **val**, **uval** modifiers for **linear** clause (Section 7.4 on page 296)
- Clarified the description of pointer mapping and pointer attachment in Section 6.3 on page 166.
- Clarified the description of memory model examples in Section 11.1 on page 482.

B.7 Changes from 4.5.0 to 5.0.0

- Added the following examples for the 5.0 features:
  - Extended **teams** construct for host execution (Section 3.3 on page 20)
  - **loop** and **teams loop** constructs specify loop iterations that can execute concurrently (Section 3.15 on page 49)
  - Task data affinity is indicated by **affinity** clause of **task** construct (Section 4.2 on page 66)
  - Display thread affinity with **OMP_DISPLAY_AFFINITY** environment variable or **omp_display_affinity()** API routine (Section 4.3 on page 67)
  - **taskwait** with dependences (Section 5.3.6 on page 112)
  - **mutexinoutset** task dependences (Section 5.3.7 on page 118)
  - Multidependence Iterators (in **depend** clauses) (Section 5.3.8 on page 121)

- Combined constructs: **parallel master taskloop** and **parallel master taskloop simd** (Section 5.8 on page 141)
- Reverse Offload through **ancestor** modifier of **device** clause. (Section 6.1.6 on page 158)
- Pointer Mapping - behavior of mapped pointers (Section 6.3 on page 166)
- Structure Mapping - behavior of mapped structures (Section 6.4 on page 175)
- Array Shaping with the *shape-operator* (Section 6.9 on page 194)
- The **declare mapper** directive (Section 6.10 on page 197)
- Acquire and Release Semantics Synchronization: Memory ordering clauses **acquire**, **release**, and **acq_rel** were added to flush and atomic constructs (Section 9.8 on page 361)
- **depobj** construct provides dependence objects for subsequent use in **depend** clauses (Section 9.10 on page 372)
- **reduction** clause for **task** construct (Section 10.10.2 on page 422)
- **reduction** clause for **taskloop** construct (Section 10.10.5 on page 436)
- **reduction** clause for **taskloop simd** construct (Section 10.10.5 on page 436)
- Memory Allocators for making OpenMP memory requests with traits (Section 11.2 on page 492)
- **requires** directive specifies required features of implementation (Section 12.6 on page 531)
- **declare variant** directive - for function variants (Section 12.7.1 on page 534)
- **metadirective** directive - for directive variants (Section 12.7.2 on page 541)
- **OMP_TARGET_OFFLOAD** Environment Variable - controls offload behavior (Section 12.11 on page 572)

- Included the following additional examples for the 4.x features:
  - more taskloop examples (Section 5.7 on page 138)
  - user-defined reduction (UDR) (Section 10.10.8 on page 450)

B.8 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 3.9 on page 37)
  - **priority** clause for **task** construct (Section 5.2 on page 102)
  - **taskloop** construct (Section 5.7 on page 138)
  - *directive-name* modifier in multiple **if** clauses on a combined construct (Section 6.1.5 on page 155)
  - unstructured data mapping (Section 6.12 on page 215)
  - **link** clause for **declare target** directive (Section 6.14.6 on page 237)
  - asynchronous target execution with **nowait** clause (Section 6.17 on page 255)
  - device memory routines and device pointers (Section 6.18.5 on page 272)
  - doacross loop nest (Section 9.11 on page 376)
  - locks with hints (Section 9.12 on page 382)
  - C/C++ array reduction (Section 10.10.1 on page 415)
  - C++ reference types in data sharing clauses (Section 10.15 on page 476)

B.9 Changes from 4.0.1 to 4.0.2

- Names of examples were changed from numbers to mnemonics
- Added SIMD examples (Section 7.1 on page 283)
- Applied miscellaneous fixes in several source codes
- Added the revision history

B.10 Changes from 4.0 to 4.0.1

Added the following new examples:

- the **proc_bind** clause (Section 4.1 on page 60)
- the **taskgroup** construct (Section 5.5 on page 134)

B.11 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 5.3 on page 103)
  - **`target`** construct (Section 6.1 on page 150)
  - array sections in device constructs (Section 6.6 on page 185)
  - **`target data`** construct (Section 6.11 on page 203)
  - **`target update`** construct (Section 6.13 on page 217)
  - **`declare target`** directive (Section 6.14 on page 223)
  - **`teams`** constructs (Section 6.16 on page 243)
  - asynchronous execution of a **`target`** region using tasks (Section 6.17.1 on page 255)
  - device runtime routines (Section 6.18 on page 265)
  - Fortran ASSOCIATE construct (Section 10.16 on page 477)
  - cancellation constructs (Section 12.5 on page 526)

Index

A

acq_rel clause, 361
acquire clause, 361
affinity
  affinity clause, 66
  close policy, 63
  master policy, 602
  primary policy, 65, 602
  proc_bind clause, 60
  spread policy, 60, 79
  task affinity, 66
affinity clause, 66
affinity display
  OMP_AFFINITY_FORMAT, 67
  omp_capture_affinity routine, 75
  OMP_DISPLAY_AFFINITY, 67
  omp_display_affinity routine, 67
  omp_get_affinity_format routine, 74
  omp_set_affinity_format routine, 74
affinity query
  omp_get_num_places routine, 79
  omp_get_place_num routine, 79
  omp_get_place_num_procs routine, 79
alloc map-type, 215
allocate directive, 497, 600
  allocator clause, 497
allocator clause, 492
allocators directive, 492, 600
  allocator clause, 492
always modifier, 269
ancestor modifier, 158
apply clause, 327
array sections
  in **map** clause, 185
array shaping
  in *motion-clause*, 194
ASSOCIATE construct, Fortran, 477
assume directive, 510
assumes directive, 510
at clause, 591
atomic construct, 349, 354, 357, 360, 361, 363, 482
  capture clause, 352
  hint clause, 360
  memory ordering clauses, 363
  read clause, 351, 363
  relaxed atomic operations, 365
  update clause, 349
  write clause, 351, 363
attribute syntax, C/C++, 5

B

barrier construct, 523
begin assumes directive, 510
begin declare target directive, 168, 169, 175, 194, 223, 227, 231, 234, 237, 239, 255, 265, 545, 600
begin declare variant directive, 540
binding
  barrier regions, 346

C

cancel construct, 526
cancellation
  cancel construct, 526
  cancellation point construct, 527
  for **parallel** region, 526
  for **taskgroup** region, 528
  for worksharing region, 526
cancellation point construct, 527
capture clause, 352, 354
clauses
  acq_rel, 361

acquire, 361
affinity, 66
allocator, 492
apply, 327
at, 591
capture, 352, 354
collapse, 31, 33, 289
combiner, 600
compare, 354
copyin, 470
copyprivate, 472
data-sharing, C++ reference in, 476
default, 600
default(none), 398
defaultmap, 160
depend, 103, 112, 144, 258, 261, 372, 585, 600
destroy, 372, 585, 600
detach, 130
device, 158
device_type, 158, 239
dist_schedule, 249
doacross, 376, 600
dynamic_allocators, 503
enter, 158, 431, 600
exclusive, 465
filter, 47
final, 98
firstprivate, 40, 411
from, 217
full, 317
grainsize, 138
hint, 342, 360
holds, 510
if, 100, 155, 211, 219
in_reduction, 422
inbranch, 290
inclusive, 465
indirect, 225
induction, 460
init, 585
initializer, 450
is_device_ptr, 273
lastprivate, 412
linear, 37, 284, 296, 600
link, 237
map, 151, 185
match, 534
memory ordering clauses, 361
mergeable, 97
motion-clause, 217
no_parallelism, 510
nocontext, 560
nogroup, 138
notinbranch, 290
novariants, 560
nowait, 28, 259, 261, 263
num_teams, 243
num_threads, 25, 519
order(concurrent), 579
ordered, 33, 369
otherwise, 541, 600
partial, 317
priority, 102
private, 287, 399, 404, 406
proc_bind, 60
read, 351, 363
reduction, 287, 415
release, 361
reverse_offload, 158
safelen, 288
schedule, 249
seq_cst, 363
severity, 591
shared, 404, 409
sizes, 305
task_reduction, 422
to, 217, 600
unified_shared_memory, 189, 531
uniform, 284
untied, 90
update, 349, 372
uses_allocators, 499
when, 541
write, 351, 363

close policy, 63
collapse clause, 31, 33, 289
combined constructs
parallel masked taskloop, 141
parallel masked taskloop simd, 141
parallel sections, 39
parallel worksharing-loop, 17
target teams, 243
taskloop simd, 440
combiner, 450
combiner clause, 600
compare clause, 354
conditional compilation
_OPENMP macro, 515
sentinel, 515
conditional modifier, 413
construct
dispatch, 560
constructs
atomic, 349, 354, 357, 360, 363, 365, 482
barrier, 523
cancel, 526
cancellation point, 527
critical, 342, 345, 361
depobj, 372
distribute, 245
do, 17, 27
flush, 484, 523
for, 17
interop, 585
loop, 49
masked, 47, 141, 473, 602
master, 602
ordered, 369, 376
parallel, 17, 18, 141
scope, 443
section, 40
sections, 40
simd, 141, 283
single, 41, 472
target, 150, 185, 243, 427, 431, 434
target data, 185, 203, 217
target enter data, 215
target exit data, 215
target update, 194, 217, 269
task, 84, 103
taskgroup, 134
taskloop, 138, 141, 144, 436
taskwait, 84, 112, 523
taskyield, 137, 523
teams, 20, 243
tile, 305
unroll, 317
workshare, 43
worksharing, 345
context selector
condition selector, 548
construct, 541, 545
device, 542
implementation, 544
user, 548
context selector, 533
context selector scoring, 552
copyin clause, 470
copyprivate clause, 472
critical construct, 342, 345, 361
hint clause, 342

D

data-sharing clauses, C++ reference in, 476
declare induction directive, 463
declare mapper directive, 197
declare reduction directive, 450
combiner, 450
initializer clause, 450
OpenMP variable identifiers, 450
declare simd directive, 234, 284
declare target directive, 158, 194, 223, 231, 234, 237, 239, 255, 256, 265, 545
device_type clause, 158
declare variant directive, 534
match clause, 534
default clause, 600
default(none) clause, 398

defaultmap clause, 160
implicit behavior, 160
variable category, 160
delete map-type, 215
depend clause, 103, 112, 144, 258, 261, 372, 585, 600
iterator modifier, 121
dependences
doacross loop nest, 376
loop-carried lexical forward, 294
task dependences, 103
taskloop dependences, 144
depobj construct, 372
depend clause, 372
destroy clause, 372
update clause, 372
depobj directive, 279
deprecated features, 599
destroy clause, 372, 585, 600
detach clause, 130
device clause
ancestor modifier, 158
device_type clause, 158, 239
directive syntax, 3
attribute, C/C++, 5
fixed form, Fortran, 9
free form, Fortran, 10
pragma, C/C++, 4
directives
allocate, 497
allocators, 492
assume, 510
assumes, 510
begin assumes, 510
begin declare target, 168, 169, 175, 194, 223, 227, 231, 234, 237, 239, 255, 265, 545, 600
begin declare variant, 540
declare induction, 463
declare mapper, 197
declare reduction, 450
declare simd, 234, 284
declare target, 158, 194, 223, 231, 234, 237, 239, 255, 256, 265, 545
declare variant, 533, 534
depobj, 279
error, 591
interchange, 329
metadirective, 533, 541
nothing, 329
requires, 158, 189, 531
reverse, 334
scan, 465
task_iteration, 144
threadprivate, 392, 470
dispatch construct, 560
nocontext clause, 560
novariants clause, 560
dist_schedule clause, 249
distribute construct, 245
dist_schedule clause, 249
do construct, 17, 27
doacross clause, 376, 600
doacross loop nest
doacross clause, 376
ordered construct, 376
dynamic_allocators clause, 503

E

enter clause, 158, 431, 600
environment display
OMP_DISPLAY_ENV, 589
omp_display_env routine, 589
environment variables
OMP_AFFINITY_FORMAT, 67
OMP_AVAILABLE_DEVICES, 282
OMP_DEFAULT_DEVICE, 282
OMP_DISPLAY_AFFINITY, 67
OMP_DISPLAY_ENV, 589
OMP_NUM_THREADS, 23
OMP_PLACES, 71, 79
OMP_TARGET_OFFLOAD, 572
error directive, 591
at clause, 591
severity clause, 591
example label, 2

omp_*verno*, 2
exclusive clause, 465

F
filter clause, 47
final clause, 98
firstprivate clause, 40
C/C++ arrays in, 411
fixed form syntax, Fortran, 9
flush construct, 361, 365, 484, 523
flushes
acquire, 361
flush construct, 484
flush with a list, 486
implicit, 361, 482
release, 361
for construct, 17
free form syntax, Fortran, 10
full clause, 317

G
grainsize clause, 138
grid modifier, 331

H
hint clause, 342, 360
holds clause, 510

I
if clause, 100, 155, 211, 219
in_reduction clause, 422, 431
with **target** construct, 434
inbranch clause, 290
inclusive clause, 465
indirect clause, 225
induction clause, 460
inductions
closed form, 460
collector clause, 463
declare induction directive, 463
induction clause, 460
inductor clause, 463
user-defined, 463
init clause, 585
initializer clause, 450
inscan modifier, 465
interchange directive, 329
internal control variables, 516
interop construct, 585
depend clause, 585
destroy clause, 585
init clause, 585
intratile modifier, 331
is_device_ptr clause, 273
iterator modifier, 121

L
lambda expressions, 241
lastprivate clause, 412
conditional modifier, 413
linear clause, 37, 284, 296, 600
link clause, 237
loop construct, 49
loop scheduling
static, 29
loop variables, Fortran, 403

M
map clause, 151, 185
alloc map-type, 208, 215
always modifier, 269
array sections in, 153
delete map-type, 215
from map-type, 152
mapper modifier, 198
to map-type, 152
tofrom map-type, 206
mapper modifier, 198
mapping
allocatable array, Fortran, 181
deep copy, 173, 198
pointer, 166
pointer attachment, 166
structure, 175
virtual functions, C++, 191
masked construct, 47, 141, 473, 602
filter clause, 47
master construct, 602

match clause, 534
memory allocators
allocator traits, 492
allocators directive, 492
memory space, 492
omp_alloc routine, 492
memory ordering clauses
acq_rel, 361
acquire, 361
release, 361
seq_cst, 363
mergeable clause, 97
metadirective directive, 541
otherwise clause, 541
when clause, 541
modifiers, **linear**
ref, 296
uval, 296
val, 296
motion-clause
from clause, 217
to clause, 217

N

nestable lock, 388
nested loop constructs, 563
no_parallelism clause, 510
nocontext clause, 560
nogroup clause, 138
non-rectangular loop nest, 35
nothing directive, 329
notinbranch clause, 290
novariants clause, 560
nowait clause, 28, 259, 261, 263
num_teams clause, 243
num_threads clause, 25, 519

O

OMP_AFFINITY_FORMAT, 67
omp_alloc routine, 492
OMP_AVAILABLE_DEVICES, 282
omp_capture_affinity routine, 75
OMP_DEFAULT_DEVICE, 282
OMP_DISPLAY_AFFINITY, 67
omp_display_affinity routine, 67
OMP_DISPLAY_ENV, 589
omp_display_env routine, 589
omp_fulfill_event routine, 130
omp_get_affinity_format routine, 74
omp_get_initial_device routine, 576
omp_get_interop_int routine, 585
omp_get_interop_ptr routine, 585
omp_get_max_threadsi routine, 576
omp_get_num_places routine, 79
omp_get_num_teams routine, 243
omp_get_num_threads routine, 56
omp_get_place_num routine, 79
omp_get_place_num_procs routine, 79
omp_get_team_num routine, 243
omp_get_wtick routine, 588
omp_get_wtime routine, 588
omp_in_final routine, 98
omp_init_allocator routine, 492
omp_init_lock routine, 382
omp_init_lock_with_hint routine, 383
omp_is_initial_device routine, 265
OMP_NUM_THREADS, 23
omp_pause_resource routine, 576
omp_pause_resource_all routine, 578
OMP_PLACES, 71, 79
omp_set_affinity_format routine, 74
omp_set_default_device routine, 268
omp_set_dynamic routine, 25, 53
omp_set_lock routine, 386
omp_set_num_threads routine, 53
omp_target_alloc routine, 269, 272
omp_target_associate_ptr routine, 269, 274
omp_target_disassociate_ptr routine, 269

omp_target_free routine, 269, 272
omp_target_is_accessible routine, 172
omp_target_is_present routine, 274
omp_target_memcpy routine, 272
omp_target_memcpy_async routine, 277
OMP_TARGET_OFFLOAD, 572
omp_test_lock routine, 386
omp_unset_lock routine, 384
omp_*verno*, 2
OMPT interface
  activating, 596
  ompt_set_callback routine, 596
  ompt_start_tool routine, 596
ompt_set_callback routine, 596
ompt_start_tool routine, 596
OpenMP context, 533
OpenMP variable identifiers
  omp_idx, 463
  omp_in, 450
  omp_orig, 454
  omp_out, 450
  omp_priv, 450
  omp_step, 463
  omp_var, 463
order(concurrent) clause, 579
  reproducible modifier, 582
  unconstrained modifier, 582
ordered clause, 33, 369
ordered construct, 369
  doacross loop nest, 376
otherwise clause, 541, 600

P

parallel construct, 17, 18, 141
parallel sections construct, 39
partial clause, 317
pointer attachment, 166
pragma syntax, C/C++, 4
primary policy, 65
priority clause, 102
private clause, 287, 399
  common blocks, Fortran, 404
  storage association, Fortran, 406
proc_bind clause, 60

R

random access iterator, C++, 53
read clause, 351, 363
reduction clause, 287, 415
  inscan modifier, 465
  on private variables, 445
  on **scope** construct, 443
  on **target** construct, 427
  on **taskloop** construct, 436
  on **taskloop simd** construct, 440
  on **teams** construct, 246
  original(private) modifier, 445
  task modifier, 425, 433
reductions
  declare reduction directive, 450
  in_reduction clause, 422
  reduction clause, 287, 415
  task_reduction clause, 422
  user-defined, 450
region nesting rules, 566
release clause, 361
requires directive, 189, 531
  reverse_offload clause, 158
  unified_shared_memory clause, 531
reverse directive, 334
reverse_offload clause, 158
routines
  omp_alloc, 492
  omp_capture_affinity, 75
  omp_display_affinity, 67
  omp_display_env, 589
  omp_fulfill_event, 130
  omp_get_affinity_format, 74
  omp_get_initial_device, 576
  omp_get_interop_int, 585
  omp_get_interop_ptr, 585
  omp_get_max_threads, 576
  omp_get_num_places, 79
  omp_get_num_teams, 243
  omp_get_num_threads, 56

omp_get_place_num, 79
omp_get_place_num_procs, 79
omp_get_team_num, 243
omp_get_wtick, 588
omp_get_wtime, 588
omp_in_final, 98
omp_init_allocator, 492
omp_init_lock, 382
omp_init_lock_with_hint, 383
omp_is_initial_device, 265
omp_pause_resource, 576
omp_pause_resource_all, 578
omp_set_affinity_format, 74
omp_set_default_device, 268
omp_set_dynamic, 25, 53
omp_set_lock, 386
omp_set_num_threads, 53
omp_target_alloc, 269, 272
omp_target_associate_ptr, 269, 274
omp_target_disassociate_ptr, 269
omp_target_free, 269, 272
omp_target_is_accessible, 172
omp_target_is_present, 274
omp_target_memcpy, 272
omp_target_memcpy_async, 277
omp_test_lock, 386
omp_unset_lock, 384
ompt_set_callback, 596
ompt_start_tool, 596

S

safelen clause, 288
scan directive, 465
  exclusive clause, 465
  inclusive clause, 465
schedule clause, 249
scope construct, 443
section construct, 40
sections construct, 40
seq_cst clause, 363
severity clause, 591
shared clause
  common blocks, Fortran, 404
  storage association, Fortran, 409
shared variables
  race conditions, 508
simd construct, 141, 283
single construct, 41, 472
sizes clause, 305
spread policy, 60, 79
standalone directive placement, 523
static scheduling, 29

T

target construct, 150, 185, 243, 255, 427, 431, 434
  defaultmap clause, 160
  depend clause, 261
  device clause, 158
  if clause, 155
  implicit mapping, 150, 206
  is_device_ptr clause, 273
  map clause, 151, 206
  nowait clause, 259, 261, 263
target data construct, 185, 203, 217
  if clause, 211
  map clause, 206
target enter data construct, 215
target exit data construct, 215
target reverse offload
  requires directive, 158
  reverse_offload clause, 158
target update construct, 217, 269
  from clause, 194, 217
  if clause, 219
  mapper, 221
  motion-clause, 217
  to clause, 194, 217
task construct, 84, 103, 255
  depend clause, 103, 258
  detach clause, 130
  final clause, 98
  if clause, 100
  mergeable clause, 97
  priority clause, 102

untied clause, 90
task dependences
anti dependence, 105
concurrent execution with, 107
flow dependence, 103
matrix multiplication, 110
mutually exclusive execution, 118
output dependence, 106
taskwait construct with, 112
transparent tasks, 126
undeferred tasks, 123
using iterators, 121
task modifier, 425, 433
task scheduling point, 90
task_iteration directive, 144
depend clause, 144
task_reduction clause, 422
taskgroup construct, 134
taskloop construct, 138, 141, 144, 436
grainsize clause, 138
nogroup clause, 138
taskloop simd construct, 440
taskwait construct, 84, 112, 523
depend clause, 112
taskyield construct, 137, 523
teams construct, 20, 243
num_teams clause, 243
threadprivate directive, 392, 470
tile construct, 305
apply clause, 327, 331
sizes clause, 305
to clause, 600
trait property, 533
trait selector, 533
trait selector set, 533

U

unified_shared_memory clause, 189, 531
uniform clause, 284
unroll construct, 317
apply clause, 327
full clause, 317
partial clause, 317
untied clause, 90
update clause, 349, 372
uses_allocators clause, 499

W

when clause, 541
workshare construct, 43
worksharing-loop constructs
do, 17
for, 17
schedule clause, 249
write clause, 351, 363