OpenMP
Application Programming
Interface

Examples

Version 5.2 – April 2022

Source codes for OpenMP 5.2 Examples are available at github (https://github.com/OpenMP/Examples/tree/v5.2).

Copyright © 1997-2022 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 5.2 Specification. The additional examples and updates are referenced in the Document Revision History of the Appendix on page 529.

Text describing an example with a 5.2 feature specifically states that the feature support begins in the OpenMP 5.2 Specification. Also, an omp_5.2 keyword is included in the metadata of the source code. These distinctions are presented to remind readers that a 5.2 compliant OpenMP implementation is necessary to use these features in codes.

Examples for most of the 5.2 features are included in this document, and 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)
Kent Milfeld (TACC, Texas Advanced Computing Center)
## Contents

1 Introduction .......................... 1
   1.1 Examples Organization ...................... 2

2 OpenMP Directive Syntax ............... 3
   2.1 C/C++ Pragmas .......................... 4
   2.2 C++ Attributes ........................ 5
   2.3 Fortran Comments (Fixed Source Form) ...... 8
   2.4 Fortran Comments (Free Source Form) ...... 9

3 Parallel Execution ....................... 13
   3.1 A Simple Parallel Loop ..................... 15
   3.2 parallel Construct ....................... 16
   3.3 teams Construct on Host ................... 18
   3.4 Controlling the Number of Threads on Multiple Nesting Levels 22
   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 .......................... 29
   3.8 collapse Clause ........................ 32
   3.9 linear Clause in Loop Constructs ........... 38
   3.10 parallel sections Construct .............. 40
   3.11 firstprivate Clause and sections Construct ........... 41
   3.12 single Construct ....................... 43
   3.13 workshare Construct ..................... 45
   3.14 masked Construct ........................ 49
   3.15 loop Construct .......................... 52
   3.16 Parallel Random Access Iterator Loop ...... 53
   3.17 omp_set_dynamic and
        omp_set_num_threads Routines .............. 54
   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 ................................................. 68

4.4 Affinity Query Functions ....................................... 79

5 Tasking .................................................................. 83
  5.1 task and taskwait Constructs ................................... 84
  5.2 Task Priority ..................................................... 103
  5.3 Task Dependences .............................................. 105
    5.3.1 Flow Dependence .......................................... 105
    5.3.2 Anti-dependence ......................................... 106
    5.3.3 Output Dependence ........................................ 107
    5.3.4 Concurrent Execution with Dependences .............. 108
    5.3.5 Matrix multiplication .................................... 109
    5.3.6 taskwait with Dependences ............................. 110
    5.3.7 Mutually Exclusive Execution with Dependences ...... 117
    5.3.8 Multidependences Using Iterators ..................... 120
    5.3.9 Dependence for Undeferred Tasks ..................... 122
  5.4 Task Detachment ................................................ 126
  5.5 taskgroup Construct ............................................ 131
  5.6 taskyield Construct ............................................ 134
  5.7 taskloop Construct ............................................. 136
  5.8 Combined parallel masked and taskloop Constructs ...... 140

6 Devices .................................................................. 143
  6.1 target Construct ................................................ 144
    6.1.1 target Construct on parallel Construct .............. 144
    6.1.2 target Construct with map Clause .................... 145
    6.1.3 map Clause with to/from map-types ................. 146
    6.1.4 map Clause with Array Sections ...................... 147
    6.1.5 target Construct with if Clause ................. 149
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1.6 Target Reverse Offload</td>
<td>152</td>
</tr>
<tr>
<td>6.2 defaultmap Clause</td>
<td>155</td>
</tr>
<tr>
<td>6.3 Pointer Mapping</td>
<td>160</td>
</tr>
<tr>
<td>6.4 Structure Mapping</td>
<td>169</td>
</tr>
<tr>
<td>6.5 Fortran Allocatable Array Mapping</td>
<td>176</td>
</tr>
<tr>
<td>6.6 Array Sections in Device Constructs</td>
<td>180</td>
</tr>
<tr>
<td>6.7 C++ Virtual Functions</td>
<td>184</td>
</tr>
<tr>
<td>6.8 Array Shaping</td>
<td>185</td>
</tr>
<tr>
<td>6.9 declare mapper Directive</td>
<td>189</td>
</tr>
<tr>
<td>6.10 target data Construct</td>
<td>196</td>
</tr>
<tr>
<td>6.10.1 Simple target data Construct</td>
<td>196</td>
</tr>
<tr>
<td>6.10.2 target data Region Enclosing Multiple target Regions</td>
<td>197</td>
</tr>
<tr>
<td>6.10.3 target data Construct with Orphaned Call</td>
<td>201</td>
</tr>
<tr>
<td>6.10.4 target data Construct with if Clause</td>
<td>204</td>
</tr>
<tr>
<td>6.11 target enter data and target exit data Constructs</td>
<td>209</td>
</tr>
<tr>
<td>6.12 target update Construct</td>
<td>212</td>
</tr>
<tr>
<td>6.12.1 Simple target data and target update Constructs</td>
<td>212</td>
</tr>
<tr>
<td>6.12.2 target update Construct with if Clause</td>
<td>214</td>
</tr>
<tr>
<td>6.13 Declare Target Directive</td>
<td>216</td>
</tr>
<tr>
<td>6.13.1 Declare Target Directive for a Procedure</td>
<td>216</td>
</tr>
<tr>
<td>6.13.2 Declare Target Directive for Class Type</td>
<td>218</td>
</tr>
<tr>
<td>6.13.3 Declare Target Directive for Variables</td>
<td>221</td>
</tr>
<tr>
<td>6.13.4 Declare Target Directive with declare simd</td>
<td>224</td>
</tr>
<tr>
<td>6.13.5 Declare Target Directive with link Clause</td>
<td>227</td>
</tr>
<tr>
<td>6.14 Lambda Expressions</td>
<td>230</td>
</tr>
<tr>
<td>6.15 teams Construct and Related Combined Constructs</td>
<td>233</td>
</tr>
<tr>
<td>6.15.1 target and teams Constructs with omp_get_num_teams and omp_get_team_num Routines</td>
<td>233</td>
</tr>
<tr>
<td>6.15.2 target, teams, and distribute Constructs</td>
<td>235</td>
</tr>
<tr>
<td>6.15.3 target teams, and Distribute Parallel Loop Constructs</td>
<td>236</td>
</tr>
<tr>
<td>6.15.4 target teams and Distribute Parallel Loop Constructs with Scheduling Clauses</td>
<td>238</td>
</tr>
<tr>
<td>6.15.5 target teams and distribute simd Constructs</td>
<td>239</td>
</tr>
</tbody>
</table>
9.11 Lock Routines ........................................... 350
  9.11.1 omp_init_lock Routine ............................ 350
  9.11.2 omp_init_lock_with_hint Routine .................. 351
  9.11.3 Ownership of Locks .............................. 352
  9.11.4 Simple Lock Routines ............................ 353
  9.11.5 Nestable Lock Routines ........................... 356

10 Data Environment ........................................ 359
  10.1 threadprivate Directive ............................. 361
  10.2 default (none) Clause ................................ 367
  10.3 private Clause ...................................... 369
  10.4 Fortran Private Loop Iteration Variables ............... 373
  10.5 Fortran Restrictions on shared and private Clauses with Common Blocks .... 375
  10.6 Fortran Restrictions on Storage Association with the private Clause .......... 377
  10.7 C/C++ Arrays in a firstprivate Clause ................ 380
  10.8 lastprivate Clause .................................. 382
  10.9 Reduction ........................................... 384
    10.9.1 reduction Clause ................................ 384
    10.9.2 Task Reduction .................................. 391
    10.9.3 Reduction on Combined Target Constructs ............... 396
    10.9.4 Task Reduction with Target Constructs ................ 400
    10.9.5 Taskloop Reduction ............................... 405
    10.9.6 Reduction with the scope Construct ................. 412
    10.9.7 User-Defined Reduction ......................... 414
  10.10 scan Directive .................................... 425
  10.11 copyin Clause ..................................... 428
  10.12 copyprivate Clause ................................ 430
  10.13 C++ Reference in Data-Sharing Clauses ................ 434
  10.14 Fortran ASSOCIATE Construct ....................... 435

11 Memory Model ............................................ 439
  11.1 OpenMP Memory Model ................................ 440
  11.2 Memory Allocators ................................... 451
  11.3 Race Conditions Caused by Implied Copies of Shared Variables in Fortran .... 458
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Program Control</td>
<td></td>
</tr>
<tr>
<td>12.1</td>
<td>Conditional Compilation</td>
<td>461</td>
</tr>
<tr>
<td>12.2</td>
<td>Internal Control Variables (ICVs)</td>
<td>462</td>
</tr>
<tr>
<td>12.3</td>
<td>Placement of <code>flush</code>, <code>barrier</code>, <code>taskwait</code> and <code>taskyield</code> Directives</td>
<td>465</td>
</tr>
<tr>
<td>12.4</td>
<td>Cancellation Constructs</td>
<td>469</td>
</tr>
<tr>
<td>12.5</td>
<td><code>requires</code> Directive</td>
<td>474</td>
</tr>
<tr>
<td>12.6</td>
<td><code>declare variant</code> Directive</td>
<td>476</td>
</tr>
<tr>
<td>12.7</td>
<td>Metadirectives</td>
<td>482</td>
</tr>
<tr>
<td>12.8</td>
<td>Nested Loop Constructs</td>
<td>492</td>
</tr>
<tr>
<td>12.9</td>
<td>Restrictions on Nesting of Regions</td>
<td>495</td>
</tr>
<tr>
<td>12.10</td>
<td>Target Offload</td>
<td>502</td>
</tr>
<tr>
<td>12.11</td>
<td>Controlling Concurrency and Reproducibility with the <code>order</code> Clause</td>
<td>506</td>
</tr>
<tr>
<td>12.12</td>
<td><code>interop</code> Construct</td>
<td>512</td>
</tr>
<tr>
<td>12.13</td>
<td>Utilities</td>
<td>515</td>
</tr>
<tr>
<td>12.13.1</td>
<td>Timing Routines</td>
<td>515</td>
</tr>
<tr>
<td>12.13.2</td>
<td>Environment Display</td>
<td>516</td>
</tr>
<tr>
<td>12.13.3</td>
<td><code>error</code> Directive</td>
<td>518</td>
</tr>
<tr>
<td>13</td>
<td>OMPT Interface</td>
<td>521</td>
</tr>
<tr>
<td>13.1</td>
<td>OMPT Start</td>
<td>522</td>
</tr>
<tr>
<td>A</td>
<td>Feature Deprecations and Updates in Examples</td>
<td>525</td>
</tr>
<tr>
<td>A.1</td>
<td>Updated Examples for Different Versions</td>
<td>526</td>
</tr>
<tr>
<td>B</td>
<td>Document Revision History</td>
<td>529</td>
</tr>
<tr>
<td>B.1</td>
<td>Changes from 5.1 to 5.2</td>
<td>529</td>
</tr>
<tr>
<td>B.2</td>
<td>Changes from 5.0.1 to 5.1</td>
<td>530</td>
</tr>
<tr>
<td>B.3</td>
<td>Changes from 5.0.0 to 5.0.1</td>
<td>531</td>
</tr>
<tr>
<td>B.4</td>
<td>Changes from 4.5.0 to 5.0.0</td>
<td>532</td>
</tr>
<tr>
<td>B.5</td>
<td>Changes from 4.0.2 to 4.5.0</td>
<td>533</td>
</tr>
<tr>
<td>B.6</td>
<td>Changes from 4.0.1 to 4.0.1</td>
<td>534</td>
</tr>
<tr>
<td>B.7</td>
<td>Changes from 4.0 to 4.0.1</td>
<td>534</td>
</tr>
<tr>
<td>B.8</td>
<td>Changes from 3.1 to 4.0</td>
<td>534</td>
</tr>
</tbody>
</table>

Index | 535 |
List of Figures

4.1 A machine architecture with two quad-core processors .......................... 60
8.1 Tiling illustrations ................................................................................. 299

List of Tables

A.1 Deprecated Features and Their Replacements ........................................ 525
A.2 Updated Examples for Version 5.2 ......................................................... 526
A.3 Updated Examples for Version 5.1 ......................................................... 527
A.4 Updated Examples for Version 5.0 ......................................................... 528
This page intentionally left blank
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. The codes for this OpenMP 5.2 Examples document have the tag v5.2.

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

https://www.openmp.org
1.1 Examples Organization

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

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

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

Some of the example labels may include version information (omp_verno) to indicate features that are illustrated by an example for a specific OpenMP version, such as “scan.1.c (omp_5.0).”

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

OpenMP *directives* use base-language mechanisms to specify OpenMP program behavior. In C code, the directives are formed exclusively with pragmas, whereas in C++ code, directives are formed from 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++ attributes

```
[[omp :: directive(  directive-specification  )]]
[[using omp : directive(  directive-specification  )]]
```

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.
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_syntaxPragma.1.c

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

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

    #pragma omp parallel num_threads(NT) // PRAG 3-4
    #pragma omp for
    for(int i=0; i<NT; i++) printf("thrd no %d\n",thrd_no());

    #pragma omp parallel num_threads(NT) // PRAG 5
    {
        int no = thrd_no();
        if (no%2) { printf("thrd no %d is Odd \n",no);}
        else { printf("thrd no %d is Even\n",no);}

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

    /* repeated 4 times, any order
    OUTPUT: thrd no 0

```
2.2 C++ Attributes

OpenMP directives for C++ can also be specified with the directive extension for the C++11 standard attributes.

The C++ 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 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.

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 
\texttt{simdlen(4)} and the other with \texttt{simdlen(8)}), as prescribed by the two different 
\texttt{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 \texttt{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 \texttt{simd} directive is used for loops calling 
the $Q$ function, in combination with the attribute form of the \texttt{declare simd} directives declaring 
the variants for $Q$.

---

\textbf{C++}

\textit{Example directive\_syntax\_attribute.1.cpp (omp\_5.0)}

```c++
#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 \ast (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 \ast (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());

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

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

// INVALID-- attribute and non-attribute on same statement
[[omp :: directive( parallel num_threads(NT) ) ]] ATTR 3
#pragma omp for PRAG 4
for(int i=0; i<NT; i++) printf("thrd no %d\n",thrd_no());
```

---

6 OpenMP Examples Version 5.2 - April 2022
// INVALID-- directive order not guaranteed

// [[ omp :: directive( parallel num_threads(NT) ) ]] ATTR 4
// [[ omp :: directive( for ) ]] ATTR 5

for(int i=0; i<NT; i++) printf("thrd no %d\n", thrd_no());

// ATTR 6
[[omp::sequence(directive(parallel num_threads(NT)), directive(for))]]
for(int i=0; i<NT; i++) printf("thrd no %d\n", thrd_no());

double tmp=0.0f;
#pragma omp simd reduction(+:tmp) simdlen(4)
for(int i=0; i<100; i++) tmp += P(i);  // PRAG DECL 1
#pragma omp simd reduction(+:tmp) simdlen(8)
for(int i=0; i<100; i++) tmp += P(i);  // PRAG DECL 2
printf("%f\n", tmp);

tmp=0.0f;
#pragma omp simd reduction(+:tmp) simdlen(4)
for(int i=0; i<100; i++) tmp += Q(i);  // ATTR DECL 3
#pragma omp simd reduction(+:tmp) simdlen(8)
for(int i=0; i<100; i++) tmp += Q(i);  // ATTR DECL 4
printf("%f\n", tmp);

tmp=0.0f;
[[omp :: directive(simd reduction(+:tmp) simdlen(4))]]
for(int i=0; i<100; i++) tmp += Q(i);  // ATTR DECL 5
[[omp :: directive(simd reduction(+:tmp) simdlen(8))]]
for(int i=0; i<100; i++) tmp += Q(i);  // ATTR DECL 6
printf("%f\n", tmp);

}  // repeated 5 times, any order:

// OUTPUT: thrd no 0
// OUTPUT: thrd no 1
// OUTPUT: thrd no 2
// OUTPUT: thrd no 3

// repeated 3 time:
// OUTPUT: 656700.000000

C++
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

```fortran
program main
  include 'omp_lib.h'
  integer NT

  NT = 4

  c sentinel c$omp or *$omp can also be used

  c$omp parallel do num_threads(NT) !comments allowed here  DIR 1
  do i = 1,NT
    write(*,'("thrd no", i2)') omp_get_thread_num()
  end do

  !$omp parallel do
  !$omp+ num_threads(NT) !cont. w. char in col. 6  DIR 2
  do i = 1,NT
    write(*,'("thrd no", i2)') omp_get_thread_num()
  end do

  *$omp parallel num_threads(NT) !multi-directive form  DIR 3
  *$omp do
  do i = 1,NT
    write(*,'("thrd no", i2)') omp_get_thread_num()
  end do
  *$omp end parallel

  end

! repeated 3 times, any order
! OUTPUT: thrd no 0
! OUTPUT: thrd no 1
! OUTPUT: thrd no 2
! OUTPUT: thrd no 3
```
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

```
program main
  use omp_lib
  integer,parameter :: NT = 4
  !$omp parallel do num_threads(NT) !DIR 1
  do i = 1,NT
    write(*,'("thrd no", i2)') omp_get_thread_num()
  end do
  !$omp parallel do & !continue line !DIR 2
  !$omp num_threads(NT) !or !$omp&
  do i = 1,NT
    write(*,'("thrd no", i2)') omp_get_thread_num()
  end do
  !$omp parallel num_threads(NT) !DIR 3
  !$omp do !DIR 4
  do i = 1,NT
    write(*,'("thrd no", i2)') omp_get_thread_num()
  end do
  !$omp end parallel
  end program
```

CHAPTER 2. OPENMP DIRECTIVE SYNTAX
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 \( \text{thrd\_no} \) are declared within the block structure and are hence private. It was necessary to explicitly declare the \( i \) variable, due to the \texttt{implicit none} statement; it could have also been declared outside the structured block.

\begin{minted}{fortran}
program main
  use omp_lib
  implicit none
  integer, parameter :: NT = 2, chunks=3

  !$omp parallel num_threads(NT) ! Fortran 2008 OMP 5.1
  block
    integer :: thrd_no, i
    thrd_no = omp_get_thread_num()
    !$omp do schedule(static,chunks)
    do i = 1, NT*chunks
      write(*,'("ndx=",i0.2," thrd_no=", i0.2)') i, thrd_no
    end do
  end block
end program

! any order
! OUTPUT: ndx=01 thrd_no=00
! OUTPUT: ndx=02 thrd_no=00
! OUTPUT: ndx=03 thrd_no=00
! OUTPUT: ndx=04 thrd_no=01
! OUTPUT: ndx=05 thrd_no=01
! OUTPUT: ndx=06 thrd_no=01
\end{minted}

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 \texttt{end parallel} directive is required to terminate the outer \texttt{parallel} construct.

\begin{verbatim}
Example directive_syntax_F_block.2f90 (omp_5.1)

program main
use omp_lib
implicit none
!
$omp parallel num_threads(2)
  if( omp_get_thread_num() == 0 ) &
    print*, "Loosely structured block -- end required."
  block
    if( omp_get_thread_num() == 0 ) &
      print*, " --"
  end block
$omp end parallel
!
$omp parallel num_threads(2)
  if( omp_get_thread_num() == 0 ) &
    print*, "Stricly structured block -- end not required."
  end block
!!$omp end parallel !is optional for strictly structured block
!
print*, "Sequential !is optional for strictly structured block"

!outer parallel
!
$omp parallel num_threads(2)
  if( omp_get_thread_num() == 0 ) &
    print*, "Outer, loosely structured block."
$omp parallel num_threads(2)
  block
    if( omp_get_thread_num() == 0 ) &
      print*, "Inner, strictly structured block."
  end block
$omp end parallel
!
$omp end parallel
! Two end directives are required here.
!
end program
\end{verbatim}
!OUTPUT, in order:
! Loosely structured block -- end required.
! Strictly structured block -- end not required.
! Sequential part
! Outer, loosely structured block.
! Inner, strictly structured block.
! Inner, strictly structured block.
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
#include <stdio.h>

void simple(int n, float *a, float *b)
{
  int i;

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

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

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
#include <omp.h>

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

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

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

Example `parallel.1.c`
Example parallel.1.f

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

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

PROGRAM PAREXAMPLE
  REAL ARRAY(10000)
  CALL SUB(ARRAY, 10000)
END PROGRAM PAREXAMPLE
```
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.

Example host_teams.1.c (omp_5.0)

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <omp.h>
#define N 1000

int main(){
    int nteams_required=2, max_thrds, tm_id;
    float sp_x[N], sp_y[N], sp_a=0.0001e0;
    double dp_x[N], dp_y[N], dp_a=0.0001e0;
    max_thrds = omp_get_num_procs()/nteams_required;
    // Create 2 teams, each team works in a different precision
    #pragma omp teams num_teams(nteams_required) thread_limit(max_thrds) private(tm_id)
    {
        tm_id = omp_get_team_num();
        if( omp_get_num_teams() != 2 ) //if only getting 1, quit
            { printf("error: Insufficient teams on host, 2 required\n");
                exit(0);
            }
    }
    if(tm_id == 0) // Do Single Precision Work (SAXPY) with this team
    {
        #pragma omp parallel
        {
```
```c
#pragma omp for //init
for(int i=0; i<N; i++) {sp_x[i] = i*0.0001; sp_y[i]=i; }

#pragma omp for simd simdlen(8)
for(int i=0; i<N; i++) {sp_x[i] = sp_a*sp_x[i] + sp_y[i];}

if(tm_id == 1) // Do Double Precision Work (DAXPY) with this team
{
  #pragma omp parallel
  {
    #pragma omp for //init
    for(int i=0; i<N; i++) {dp_x[i] = i*0.0001; dp_y[i]=i; }

    #pragma omp for simd simdlen(4)
    for(int i=0; i<N; i++) {dp_x[i] = dp_a*dp_x[i] + dp_y[i];}
  }
}
printf("i=%d sp|dp %f %f \n",N-1, sp_x[N-1], dp_x[N-1]);
printf("i=%d sp|dp %f %f \n",N/2, sp_x[N/2], dp_x[N/2]);

//OUTPUT1:i=999 sp|dp 999.000000 999.000010
//OUTPUT2:i=500 sp|dp 500.000000 500.000005
return 0;
}
```

Example host_teams.1.f90 (omp_5.0)
tm_id = omp_get_team_num()

if( omp_get_num_teams() /= 2 ) then  !! if only getting 1, quit
  stop "error: Insufficient teams on host, 2 required."
endif

!! Do Single Precision Work (SAXPY) with this team
if(tm_id == 0) then
  !$omp parallel
  !$omp do  !! init
  do i = 1,N
    sp_x(i) = i*0.0001e0
    sp_y(i) = i
  end do
  !$omp do simd simdlen(8)
  do i = 1,N
    sp_x(i) = sp_a*sp_x(i) + sp_y(i)
  end do
  !$omp end parallel
endif

!! Do Double Precision Work (DAXPY) with this team
if(tm_id == 1) then
  !$omp parallel
  !$omp do  !! init
  do i = 1,N
    dp_x(i) = i*0.0001d0
    dp_y(i) = i
  end do
  !$omp do simd simdlen(4)
  do i = 1,N
    dp_x(i) = dp_a*dp_x(i) + dp_y(i)
  end do
  !$omp end parallel
endif

!!omp end teams
write(*,'( "i=",i4," sp|dp= ", e15.7, d25.16 )') &
N, sp_x(N), dp_x(N)
write(*,'( "i=",i4," sp|dp= ", e15.7, d25.16 )') &
N/2, sp_x(N/2), dp_x(N/2)
end program
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:

---

**Example nthrs_nesting.1.c**

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

int main (void)
{
  omp_set_nested(1);
  omp_set_dynamic(0);
  #pragma omp parallel
  {
    #pragma omp parallel
    {
      #pragma omp single
      {
        /* If OMP_NUM_THREADS=2,3 was set, the following should print:
         * Inner: num_thds=3
         * Inner: num_thds=3
         *
         * If nesting is not supported, the following should print:
         * Inner: num_thds=1
         * Inner: num_thds=1
         */
        printf("Inner: num_thds=%d\n", omp_get_num_threads());
      }
    }
    #pragma omp barrier
  }
  omp_set_nested(0);
  #pragma omp parallel
  {
    #pragma omp single
    {
      /* Even if OMP_NUM_THREADS=2,3 was set, the following should print,
         because nesting is disabled:
         * Inner: num_thds=1
         * Inner: num_thds=1
         */
      printf("Inner: num_thds=%d\n", omp_get_num_threads());
    }
  }
```

---

OpenMP Examples Version 5.2 - April 2022
Example nthrs_nesting.1.f

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

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

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

```c
/*
 * 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;
```
! If OMP_NUM_THREADS=2,3 was set, the following should print:
! Outer: num_thds= 2
print *, "Outer: num_thds=", omp_get_num_threads()
!omp end single
!omp end parallel
end
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.

---

**Example nthrs_dynamic.1.c**

```
#include <omp.h>

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

---

**Example nthrs_dynamic.1.f**

```
PROGRAM EXAMPLE
    INCLUDE "omp_lib.h"      ! or USE OMP_LIB
    CALL OMP_SET_DYNAMIC(.FALSE.)
!$OMP PARALLEL NUM_THREADS(10)
    ! do work here
!$OMP END PARALLEL
END PROGRAM EXAMPLE
```
The call to the `omp_set_dynamic` routine with a non-zero argument in C/C++, or `.TRUE.` in Fortran, allows the OpenMP implementation to choose any number of threads between 1 and 10.

```
#include <omp.h>

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

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

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

Example fort_do.1.f

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

SUBROUTINE DO_GOOD()
  INTEGER I, J
  REAL A(1000)
  DO 100 I = 1,10
    !$OMP DO
    DO 100 J = 1,10
      CALL WORK(I,J)
    100 CONTINUE ! !$OMP ENDDO implied here
  !$OMP DO
  DO 200 J = 1,10
    200 A(I) = I + 1
  !$OMP ENDDO
  !$OMP DO
  DO 300 I = 1,10
    DO 300 J = 1,10
      CALL WORK(I,J)
    300 CONTINUE
  !$OMP ENDDO
END SUBROUTINE DO_GOOD
```

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

Example fort_do.2.f

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

SUBROUTINE DO_WRONG
  INTEGER I, J
  REAL A(1000)
  DO 100 I = 1,10
    !$OMP DO
    DO 100 J = 1,10
      CALL WORK(I,J)
    100 CONTINUE ! !$OMP ENDDO implied here
  !$OMP DO
  DO 200 J = 1,10
    200 A(I) = I + 1
  !$OMP ENDDO
  !$OMP DO
  DO 300 I = 1,10
    DO 300 J = 1,10
      CALL WORK(I,J)
    300 CONTINUE
  !$OMP ENDDO
END SUBROUTINE DO_WRONG
```
DO 100 I = 1,10

!$OMP DO
DO 100 J = 1,10
CALL WORK(I,J)
100 CONTINUE

!$OMP ENDDO
END SUBROUTINE DO_WRONG
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 loop construct, as follows:

```c
#include <math.h>

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

    #pragma omp for nowait
    for (i=0; i<m; i++)
      y[i] = sqrt(z[i]);
  }
}
```

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

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

CHAPTER 3. PARALLEL EXECUTION 29
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
#include <math.h>

void nowait_example2(int n, float *a, float *b, float *c, float *y, float *z)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for schedule(static) nowait
        for (i=0; i<n; i++)
            c[i] = (a[i] + b[i]) / 2.0f;
        #pragma omp for schedule(static) nowait
        for (i=0; i<n; i++)
            z[i] = sqrtf(c[i]);
        #pragma omp for schedule(static) nowait
        for (i=1; i<=n; i++)
            y[i] = z[i-1] + a[i];
    }
}
```
Example nowait.2.f90

SUBROUTINE NOWAIT_EXAMPLE2(N, A, B, C, Y, Z)
    INTEGER N
    REAL A(*), B(*), C(*), Y(*), Z(*)
    INTEGER I
    !$OMP PARALLEL
    !$OMP DO SCHEDULE(STATIC)
    DO I=1,N
      C(I) = (A(I) + B(I)) / 2.0
    ENDDO
    !$OMP END DO NOWAIT
    !$OMP DO SCHEDULE(STATIC)
    DO I=1,N
      Z(I) = SQRT(C(I))
    ENDDO
    !$OMP END DO NOWAIT
    !$OMP DO SCHEDULE(STATIC)
    DO I=2,N+1
      Y(I) = Z(I-1) + A(I)
    ENDDO
    !$OMP END DO NOWAIT
    !$OMP END PARALLEL
END SUBROUTINE NOWAIT_EXAMPLE2
### 3.8 collapse Clause

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

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

---

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

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

---

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

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

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

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

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

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

The code prints

```
0 1 1
0 1 2
0 2 1
1 2 2
1 3 1
1 3 2
```
Example collapse.3.c (omp_3.0)

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

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

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

Example collapse.3.f (omp_3.0)

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

Example collapse.4.c (omp_5.0)

```c
#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];
        }
    printf("b[0][0] = %f, b[M-1][M-1] = %f\n", b[0][0], b[M-1][M-1]);
    return 0;
}
```

Example collapsed.4.f90 (omp_5.0)

module calc_m
  interface
  subroutine calc_a(n, m, a)
    integer n, m
    real a(n+2,m)
    ! routine to calculate a
    ! For variable a(*,j):
    ! a(1,j),...,a(n,j) contains the n samples
    ! a(n+1,j) contains the sample mean
    ! a(n+2,j) contains the standard deviation
  end subroutine
  end interface
end module

program main
  use calc_m
  integer, parameter :: N=20, M=10
  real a(N+2,M), b(M,M)
  real temp
  integer i, j, k

  call calc_a(N,M,a)

  !$omp parallel do collapse(2) private(k,temp)
  do i = 1, M
    do j = i, M
      temp = 0.0
      do k = 1, N
        temp = temp + (a(k,i)-a(N+1,i))*(a(k,j)-a(N+1,j))
      end do
      end do
      b(i,j) = temp / (a(N+2,i) * a(N+2,j) * (N - 1))
      b(j,i) = b(i,j)
  end do
end do
  print *,"b(1,1) = ",b(1,1),", b(M,M) = ",b(M,M)
end program
3.9 linear Clause in Loop Constructs

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

---

```
#include <stdio.h>

#define N 100

int main(void)
{
    float a[N], b[N/2];
    int i, j;

    for ( i = 0; i < N; i++ )
        a[i] = i + 1;

    j = 0;

    #pragma omp parallel
    #pragma omp for linear(j:1)
    for ( i = 0; i < N; i += 2 ) {
        b[j] = a[i] * 2.0f;
        j++;
    }

    printf( "%d %f %f\n", j, b[0], b[j-1] );
    /* print out: 50 2.0 198.0 */

    return 0;
}
```

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

program linear_loop
    implicit none
    integer, parameter :: N = 100
    real :: a(N), b(N/2)
    integer :: i, j
    do i = 1, N
        a(i) = i
    end do
    j = 0
    !$omp parallel
    !$omp do linear(j:1)
    do i = 1, N, 2
        j = j + 1
        b(j) = a(i) * 2.0
    end do
    !$omp end parallel
    print *, j, b(1), b(j)
! print out: 50 2.0 198.0
end program
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.

---

**Example psections.1.c**

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

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

---

**Example psections.1.f**

```fortran
SUBROUTINE SECT_EXAMPLE()
!$OMP PARALLEL SECTIONS
!$OMP SECTION
CALL XAXIS()
!$OMP SECTION
CALL YAXIS()
!$OMP SECTION
CALL ZAXIS()
!$OMP END PARALLEL SECTIONS
END SUBROUTINE SECT_EXAMPLE
```
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
#include <omp.h>
#include <stdio.h>
#define NT 4
int main( ) {
  int section_count = 0;
  omp_set_dynamic(0);
  omp_set_num_threads(NT);
  #pragma omp parallel
  #pragma omp sections firstprivate( section_count )
  {
    #pragma omp section
    {
      section_count++;
      /* may print the number one or two */
      printf( "section_count %d\n", section_count );
    }
    #pragma omp section
    {
      section_count++;
      /* may print the number one or two */
      printf( "section_count %d\n", section_count );
    }
  }
  return 0;
}
```

CHAPTER 3. PARALLEL EXECUTION 41
Example fpriv_sections.1.f90

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

```c
#include <stdio.h>

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

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

---

C / C++
Example single.1.f

```fortran
SUBROUTINE WORK1()
END SUBROUTINE WORK1

SUBROUTINE WORK2()
END SUBROUTINE WORK2

PROGRAM SINGLE_EXAMPLE
!
$OMP PARALLEL
!
$OMP SINGLE
print *, "Beginning work1."
$OMP END SINGLE
CALL WORK1()

$OMP SINGLE
print *, "Finishing work1."
$OMP END SINGLE
!
$OMP SINGLE
print *, "Finished work1 and beginning work2."
$OMP END SINGLE NOWAIT
CALL WORK2()
!
$OMP END PARALLEL

END PROGRAM SINGLE_EXAMPLE
```
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

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

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

Example workshare.2.f

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

**Example workshare.3.f**

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

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

Each task gets worked on in order by the threads:

**Example workshare.4.f**

```fortran
SUBROUTINE WSHARE4(AA, BB, CC, DD, EE, FF, GG, HH, N)
INTEGER N
REAL AA(N,N), BB(N,N), CC(N,N)
REAL DD(N,N), EE(N,N), FF(N,N)
REAL GG(N,N), HH(N,N)

!$OMP PARALLEL
!$OMP WORKSHARE
AA = BB
CC = DD
WHERE (EE .ne. 0) FF = 1 / EE
CC = DD
!$OMP END WORKSHARE
!$OMP END PARALLEL
END SUBROUTINE WSHARE4
```
In the following example, an assignment to a shared scalar variable is performed by one thread in a workshare while all other threads in the team wait.

Example workshare.5.f

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

Example workshare.6.f
Fortran execution rules must be enforced inside a workshare construct. In the following example, the same result is produced in the following program fragment regardless of whether the code is executed sequentially or inside an OpenMP program with multiple threads:

Example workshare.7.f

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

INTEGER N
REAL AA(N), BB(N), CC(N)

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

END SUBROUTINE WSHARE7
```

Fortran
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.

```
#include <stdio.h>
extern float average(float, float, float);

void masked_example( float* x, float* xold, int n, float tol )
{
  int c, i, toobig;
  float error, y;
  c = 0;
  #pragma omp parallel
  {
    do {
      #pragma omp for private(i)
      for( i = 1; i < n-1; ++i ){
        xold[i] = x[i];
      }
    } while( toobig > 0 );
    #pragma omp barrier
  }
  #pragma omp masked // primary thread (thread 0)
  {
    ++c;
    printf( "iteration %d, toobig=%d\n", c, toobig );
  }
  } while( toobig > 0 );
```

Example masked.1.c (omp_5.1)
#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);
}

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
    !$OMP END MASKED
ENDO
!$OMP BARRIER
!$OMP MASKED FILTER(1) ! thread 1
    // The printf statement will not be executed
    // if the number of threads is less than 2.
    PRINT *, 'Total number of iterations =', C
S-35  !$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.

---

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

```c
#include <stdio.h>
#define N 100
int main()
{
    float x[N], y[N];
    float a = 2.0;
    for(int i=0; i<N; i++) { x[i]=i; y[i]=0; } // initialize
    #pragma omp parallel
    {
        #pragma omp loop
        for(int i = 0; i < N; ++i) y[i] = a*x[i] + y[i];
    }
    if(y[N-1] != (N-1)*2.0) printf("Error: 2*(N-1) != y[N-1]=%f", y[N-1]);
}
```

---

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

```fortran
program main
    integer, parameter :: N=100
    real :: x(N), y(N)
    real :: a = 2.0e0
    x=(/ (i,i=1,N) /); y=1.0e0 !! initialize
    !$omp parallel
    !$omp loop
    do i=1,N; y(i) = a*x(i) + y(i); enddo
    !$omp end parallel
    if(y(N) /= N*2.0e0) print*,"Error: 2*N /= y(N); y(N)=" , y(N)
end program
```

---
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)

```cpp
#include <vector>

void iterator_example()
{
    std::vector<int> vec(23);
    std::vector<int>::iterator it;

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

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

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

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

void dynthreads(float *x, int npoints)
{
    int iam, ipoints;

    omp_set_dynamic(0);
    omp_set_num_threads(16);

    #pragma omp parallel shared(x, npoints) private(iam, ipoints)
    {
        if (omp_get_num_threads() != 16)
            abort();

        iam = omp_get_thread_num();
        ipoints = npoints/16;
        do_by_16(x, iam, ipoints);
    }
```
Example set_dynamic_nthrs_1.f

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

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

---

**Example get_nthrs.1.c**

```c
#include <omp.h>

void work(int i);

void incorrect() {
    int np, i;
    np = omp_get_num_threads(); /* misplaced */
    #pragma omp parallel for schedule(static)
    for (i=0; i < np; i++)
        work(i);
}
```

---

**Example get_nthrs.1.f**

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

SUBROUTINE INCORRECT()
    INCLUDE "omp_lib.h" ! or USE OMP_LIB
    INTEGER I, NP
    NP = OMP_GET_NUM_THREADS() ! misplaced: will return 1
    !$OMP PARALLEL DO SCHEDULE(STATIC)
    DO I = 0, NP-1
        CALL WORK(I)
    ENDDO
    !$OMP END PARALLEL DO
END SUBROUTINE INCORRECT
```

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

### Example get_nths.2.c

```c
#include <omp.h>

void work(int i);

void correct()
{
  int i;

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

### Example get_nths.2.f

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

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

  !$OMP PARALLEL PRIVATE(I)
  I = OMP_GET_THREAD_NUM()
  CALL WORK(I)
  !$OMP END PARALLEL
END SUBROUTINE CORRECT
```
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.

![Machine Architecture Diagram](image)

**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):

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

or

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

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.

```
S-1 #pragma omp parallel proc_bind(spread) num_threads(4)
```

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

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

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

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

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

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

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

---

OpenMP Examples Version 5.2 - April 2022
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.

---

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

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

---

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

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

It is unspecified on which place the 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.

Example affinity.4.c (omp_4.0)

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

Example affinity.4.f90 (omp_4.0)

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

It is unspecified on which place the 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.

```
Example affinity.5.c (omp_4.0)
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 }
```

```
Example affinity.5.f (omp_4.0)
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
```
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).

---

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

```c
void task_affinity(double *A, int N)
{
    double * B;
    #pragma omp task depend(out:B) shared(B) affinity(A[0:N])
    {
        B = alloc_init_B(A,N);
    }
    #pragma omp task depend( in:B) shared(B) affinity(A[0:N])
    {
        compute_on_B(B,N);
    }
}
```

---

OpenMP Examples Version 5.2 - April 2022
Example affinity.6.f90 (omp_5.0)

```fortran
subroutine task_affinity(A, N)
    external alloc_init_B
    external compute_on_B
    double precision, allocatable :: B(:)

    !$omp task depend(out:B) shared(B) affinity(A)
    call alloc_init_B(B,A)
    !$omp end task

    !$omp task depend(in:B) shared(B) affinity(A)
    call compute_on_B(B)
    !$omp end task

    !$omp taskwait

end subroutine
```

CHAPTER 4. OPENMP AFFINITY  67
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.

```
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
```
// doing work here

#pragma omp parallel num_threads( omp_get_num_procs() )
{
  if(omp_get_thread_num()==0)
    printf("%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
}

// Report Affinity for 1/2 number of threads
#pragma omp parallel num_threads( omp_get_num_procs()/2 )
{
  if(omp_get_thread_num()==0)
    printf("Report Affinity for using 1/2 of max threads.\n");

  // DISPLAY OUTPUT (default format) has been sorted:
  // team_num= 0, nesting_level= 1, thread_num= 0, thread_affinity= 0,1
  // team_num= 0, nesting_level= 1, thread_num= 1, thread_affinity= 2,3
  // team_num= 0, nesting_level= 1, thread_num= 2, thread_affinity= 4,5
  // team_num= 0, nesting_level= 1, thread_num= 3, thread_affinity= 6,7

  // do work
}

return 0;

---

Example affinity_display.1.f90 (omp_5.0)

```
program affinity_display ! MAX threads = 8, single socket system
use omp_lib
implicit none
character(len=0) :: null

! API call - Displays Affinity of Primary Thread
```
call 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

!$omp parallel num_threads(omp_get_num_procs())
if(omp_get_thread_num()==0) then
  print*, "1st Parallel Region -- Affinity Reported"
endif

! 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
!$omp end parallel

!$omp parallel num_threads( omp_get_num_procs() )
if(omp_get_thread_num()==0) then
  print*, "Same Affinity in Parallel Region -- no Affinity Reported"
endif

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

! doing more work here
!$omp end parallel

! Report Affinity for 1/2 number of threads
!$omp parallel num_threads( omp_get_num_procs()/2 )
if(omp_get_thread_num()==0) then
  print*, "Altered Affinity in Parallel Region -- Affinity Reported"
endif

! DISPLAY OUTPUT (default format) has been sorted:
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:

- `OMP_PROC_BIND="TRUE"`
- `OMP_NUM_THREADS="2,4"`
- `OMP_PLACES="\{0,2,4,6\},\{1,3,5,7\}"`
- `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.
Example affinity_display.2.c (omp_5.0)

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

void socket_work(int socket_num, int n_thrds);

int main(void)
{
    int n_sockets, socket_num, n_thrds_on_socket;

    omp_set_nested(1); // or env var= OMP_NESTED=true
    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"

    #pragma omp parallel num_threads(n_sockets) private(socket_num)
    {
        socket_num = omp_get_place_num();

        if(socket_num==0)
            printf(" LEVEL 1 AFFINITIES 1 thread/socket, %d sockets:\n\n", n_sockets);

        // not needed if OMP_DISPLAY_AFFINITY=TRUE
        omp_display_affinity(NULL);

        // 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

        socket_work(socket_num, n_thrds_on_socket);
    }
    return 0;
}

void socket_work(int socket_num, int n_thrds)
```

```c
#pragma omp parallel num_threads(n_thrds)
{
    if(omp_get_thread_num()==0)
        printf(" LEVEL 2 AFFINITIES, %d threads on socket %d\n",
                n_thrds, socket_num);

    // not needed if OMP_DISPLAY_AFFINITY=TRUE
    omp_display_affinity(NULL);

    // OUTPUT:
    // LEVEL 2 AFFINITIES, 4 threads on socket 0
    // nest_level= 2, parent_thrd_num= 0, thrd_num= 0, thrd affinity= 0
    // nest_level= 2, parent_thrd_num= 0, thrd_num= 1, thrd_affinity= 2
    // nest_level= 2, parent_thrd_num= 0, thrd_num= 2, thrd_affinity= 4
    // nest_level= 2, parent_thrd_num= 0, thrd_num= 3, thrd_affinity= 6

    // LEVEL 2 AFFINITIES, 4 threads on socket 1
    // nest_level= 2, parent_thrd_num= 1, thrd_num= 0, thrd_affinity= 1
    // nest_level= 2, parent_thrd_num= 1, thrd_num= 1, thrd_affinity= 3
    // nest_level= 2, parent_thrd_num= 1, thrd_num= 2, thrd_affinity= 5
    // nest_level= 2, parent_thrd_num= 1, thrd_num= 3, thrd_affinity= 7

    // ... Do Some work on Socket
}
```

Example affinity_display.2.f90 (omp_5.0)

```fortran
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

    ! 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
end program

subroutine socket_work(socket_num, n_thrds)
use omp_lib
implicit none
integer :: socket_num, n_thrds
character(len=0) :: null

    !$omp parallel num_threads(n_thrds)
    if(omp_get_thread_num()==0) then
        write(*,'("LEVEL 2 AFFINITIES, ",i0," threads on socket ",i0)') &
        n_thrds,socket_num
    endif
    call omp_display_affinity(null) ! not needed

    ! OUTPUT:
    ! LEVEL 2 AFFINITIES, 4 threads on socket 0
    ! nest_level= 2, parent_thrd_num= 0, thrd_num= 0, thrd_affinity= 0
    ! nest_level= 2, parent_thrd_num= 0, thrd_num= 1, thrd_affinity= 2
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: 

```
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`).
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 // CODE SEGMENT 1 AFFINITY FORMAT
S-20 // CODE SEGMENT 1 AFFINITY FORMAT
S-21 // Get and Display Default Affinity Format
S-22
S-23 nchars = omp_get_affinity_format(default_format,(size_t)FORMAT_STORE);
S-24 printf("Default Affinity Format is: %s
", default_format);
S-25
S-26 if(nchars >= FORMAT_STORE){
S-27 printf("Caution: Reported Format is truncated. Increase
");
S-28 printf(" FORMAT_STORE to %d.\n", nchars+1);
S-29 }
S-30
S-31 // Set Affinity Format
S-32
S-33 omp_set_affinity_format(my_format);
S-34 printf("Affinity Format set to: %s\n", my_format);
S-35
S-36
S-37 // CODE SEGMENT 2 CAPTURE AFFINITY
S-38 // CODE SEGMENT 2 CAPTURE AFFINITY
S-39 // Set up buffer for affinity of n threads
S-40
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++){
buffer[i]=(char *)malloc( sizeof(char) * BUFFER_STORE);

// Capture Affinity using Affinity Format set above.
// Use max reduction to check size of buffer areas
max_req_store = 0;
#pragma omp parallel private(thrd_num,nchars) \  
  reduction(max:max_req_store)
{
  //safety: don’t exceed # of buffers
  if(omp_get_num_threads()>n) exit(1);

  thrd_num=omp_get_thread_num();
  nchars=omp_capture_affinity(buffer[thrd_num],
    (size_t)BUFFER_STORE,NULL);
  if(nchars > max_req_store) max_req_store=nchars;

  // ...

  for(i=0;i<n;i++){
    printf("thrd_num= %d, affinity: %s\n", i,buffer[i]);
  }

  // For 4 threads with OMP_PLACES='{0,1},{2,3},{4,5},{6,7}'
  // Format    host=%20H thrd_num=%0.4n binds_to=%A

  // affinity: host=hpc.cn567 thrd_num=0000 binds_to=0,1
  // affinity: host=hpc.cn567 thrd_num=0001 binds_to=2,3
  // affinity: host=hpc.cn567 thrd_num=0002 binds_to=4,5
  // affinity: host=hpc.cn567 thrd_num=0003 binds_to=6,7

  if(max_req_store>=BUFFER_STORE){
    printf("Caution: Affinity string truncated. Increase\n");
    printf("BUFFER_STORE to %d\n",max_req_store+1);
  }

  for(i=0;i<n;i++) free(buffer[i]);
  free (buffer);
  return 0;
}
Example affinity_display.3.f90 (omp_5.0)

```fortran
program affinity_display
  use omp_lib
  implicit none
  integer, parameter :: FORMAT_STORE=80
  integer, parameter :: BUFFER_STORE=80
  integer :: i, n, thrd_num, nchars, max_req_store
  character(FORMAT_STORE) :: default_format
  character(*), parameter :: my_format = 
    "host=%20H thrd_num=%0.4n binds_to=%A"
  character(:), allocatable :: buffer(:)
  character(len=0) :: null

! CODE SEGMENT 1    AFFINITY FORMAT

! Get and Display Default Affinity Format
  nchars = omp_get_affinity_format(default_format)
  print*,"Default Affinity Format: ", trim(default_format)
  if( nchars > FORMAT_STORE) then
    print*,"Caution: Reported Format is truncated. Increase"
    print*,"FORMAT_STORE to ", nchars
  endif

! Set Affinity Format
  call omp_set_affinity_format(my_format)
  print*,"Affinity Format set to: ", my_format

! CODE SEGMENT 2    CAPTURE AFFINITY

! Set up buffer for affinity of n threads
  n = omp_get_num_procs()
  allocate( character(len=BUFFER_STORE)::buffer(0:n-1) )

! Capture Affinity using Affinity Format set above.
! Use max reduction to check size of buffer areas
  max_req_store = 0
  !$omp parallel private(thrd_num,nchars) reduction(max:max_req_store)
```

OpenMP Examples Version 5.2 - April 2022
if(omp_get_num_threads()>n) stop "ERROR: increase buffer lines"

thrd_num=omp_get_thread_num()
nchars=omp_capture_affinity(buffer(thrd_num),null)
if(nchars>max_req_store) max_req_store=nchars

! ...

(!$omp end parallel

do i = 0, n-1
    print*, "thrd_num= ",i,"  affinity:", trim(buffer(i))
end do

! For 4 threads with OMP_PLACES='{0,1},{2,3},{4,5},{6,7}'
! Format:  host=%20H thrd_num=%0.4n binds_to=%A

! affinity: host=hpc.cn567   thrd_num=0000 binds_to=0,1
! affinity: host=hpc.cn567   thrd_num=0001 binds_to=2,3
! affinity: host=hpc.cn567   thrd_num=0002 binds_to=4,5
! affinity: host=hpc.cn567   thrd_num=0003 binds_to=6,7

if(max_req_store > BUFFER_STORE) then
    print*, "Caution: Affinity string truncated. Increase"
    print*, BUFFER_STORE to ",max_req_store
endif

dereallocate(buffer)
end program

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 subpartition of the original partition. Hence, the `omp_get_place_num_procs` query function returns the number of elements (here `procs = cores`) in the subpartition of the thread. After each parent thread creates its nested parallel region on the section, the socket number and thread number are reported.

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

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

void socket_init(int socket_num)
{
    int n_procs;
    n_procs = omp_get_place_num_procs(socket_num);
    #pragma omp parallel num_threads(n_procs) proc_bind(close)
    {
        printf("Reporting in from socket num, thread num: %d %d\n",
                socket_num,omp_get_thread_num() );
    }
}

int main()
{
    int n_sockets, socket_num;
    omp_set_nested(1); // or export OMP_NESTED=true
    omp_set_max_active_levels(2); // or export OMP_MAX_ACTIVE_LEVELS=2
    n_sockets = omp_get_num_places();
    #pragma omp parallel num_threads(n_sockets) private(socket_num) \ 
        proc_bind(spread)
    {
        socket_num = omp_get_place_num();
        socket_init(socket_num);
    }
```
Example affinity_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)
  socket_num = omp_get_place_num()
  call socket_init(socket_num)
  !$omp end parallel
end program
Tasking constructs provide units of work to a thread for execution. Worksharing constructs do this, too (e.g. for, do, sections, and singles constructs); but the work units are tightly controlled by an iteration limit and limited scheduling, or a limited number of sections or single regions. Worksharing was designed with "data parallel" computing in mind. Tasking was designed for "task parallel" computing and often involves non-locality or irregularity in memory access.

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

Sibling tasks are synchronized by the taskwait construct, and tasks and their descendental 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, in the OpenMP Application Programming Interface section.
### 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.

---

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

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

extern void process(struct node *);

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

---

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

```fortran
RECURSIVE SUBROUTINE traverse ( P )
    TYPE Node
    TYPE(Node), POINTER :: left, right
END TYPE Node

TYPE(Node) :: P

IF (associated(P%left)) THEN
    !$OMP TASK ! P is firstprivate by default
    CALL traverse(P%left)
ENDIF

IF (associated(P%right)) THEN
    !$OMP END TASK
ENDIF
```

---
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.

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

```
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

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

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

---

Example tasking.3.f90 (omp_3.0)

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

-----------
CHAPTER 5. TASKING

87
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)
```

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

```c/c++
Example tasking.4.f (omp_3.0)
```

```fortran
RECURSIVE INTEGER FUNCTION fib(n) RESULT(res)
INTEGER n, i, j
IF ( n .LT. 2) THEN
    res = n
ELSE
    !$OMP TASK SHARED(i)
    i = fib(n-1)
    !$OMP END TASK
    !$OMP TASK SHARED(j)
    j = fib(n-2)
    !$OMP END TASK
    !$OMP TASKWAIT
    res = i+j
END IF
END FUNCTION
```
Note: There are more efficient algorithms for computing Fibonacci numbers. This classic recursion algorithm is for illustrative purposes.

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

```
Example tasking.5.c (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;
            for (i=0; i<LARGE_NUMBER; i++)
                #pragma omp task // i is firstprivate, item is shared
                process(item[i]);
        }
    }
}
```

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

real*8 item(10000000)
integer i

!$omp parallel
!$omp single ! loop iteration variable i is private
do i=1,10000000
    !$omp task
        ! i is firstprivate, item is shared
    call process(item(i))
!$omp end task
end do
!$omp end single
```
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
#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;
}
```
The following two examples demonstrate how the scheduling rules illustrated in Section 2.11.3 of the OpenMP 4.0 specification affect the usage of threadprivate variables in tasks. A threadprivate variable can be modified by another task that is executed by the same thread. Thus, the value of a threadprivate variable cannot be assumed to be unchanged across a task scheduling point. In untied tasks, task scheduling points may be added in any place by the implementation.

A task switch may occur at a task scheduling point. A single thread may execute both of the task regions that modify tp. The parts of these task regions in which tp is modified may be executed in any order so the resulting value of var can be either 1 or 2.
In this example, scheduling constraints prohibit a thread in the team from executing a new task that modifies `tp` while another such task region tied to the same thread is suspended. Therefore, the value written will persist across the task scheduling point.
Example tasking.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       var = tp; // Value does not change after write above
S-17     }
S-18   }
S-19 }
S-20 }
S-21 }

Example tasking.8.f (omp_3.0)

S-1 module example
S-2   integer tp
S-3 !$omp threadprivate(tp)
S-4   integer var
S-5   contains
S-6   subroutine work
S-7 !$omp parallel
S-8     ! do work here
S-9 !$omp task
S-10   tp = tp + 1
S-11     ! do work here
S-12 !$omp task
S-13     ! do work here but don’t modify tp
S-14 !$omp end task
S-15   var = tp  ! Value does not change after write above
S-16 !$omp end task
S-17 !$omp end parallel
The following two examples demonstrate how the scheduling rules illustrated in Section 2.11.3 of the OpenMP 4.0 specification affect the usage of locks and critical sections in tasks. If a lock is held across a task scheduling point, no attempt should be made to acquire the same lock in any code that may be interleaved. Otherwise, a deadlock is possible.

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

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

void work() {
    omp_lock_t lock;
    omp_init_lock(&lock);
    #pragma omp parallel
    {
        int i;
        #pragma omp for
        for (i = 0; i < 100; i++) {
            #pragma omp task
            {
                // lock is shared by default in the task
            }
            #pragma omp task
            {
                // do work here
            }
            #pragma omp task
            {
                // lock is shared by default in the task
            }
        }
    }
    #pragma omp end parallel
}
```
omp_set_lock(&lock);

// Capture data for the following task
#pragma omp task
  // Task Scheduling Point 1
  { /* do work here */ }
omp_unset_lock(&lock);
}
omp_destroy_lock(&lock);

Example tasking.10.f90 (omp_3.0)

module example
  include 'omp_lib.h'
  integer (kind=omp_lock_kind) lock
  integer i
contains
subroutine work
  call omp_init_lock(lock)
  !$omp parallel
    !$omp do
doi=1,100
    !$omp task
      !$omp task
        ! Outer task
        call omp_set_lock(lock) ! lock is shared by
        ! default in the task
        ! Capture data for the following task
        !$omp task ! Task Scheduling Point 1
        ! do work here
        !$omp end task
      !$omp end task
    call omp_unset_lock(lock)
  !$omp end task
end do
  !$omp end parallel
  call omp_destroy_lock(lock)
end subroutine
end module
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`).

```
Example tasking.11.c (omp_3.1)
#include <stdio.h>
void foo ( )
{
    int x = 2;
    #pragma omp task shared(x) mergeable
    {
        x++;
    }
    #pragma omp taskwait
    printf("%d\n",x); // prints 3
}
```

```
Example tasking.11.f90 (omp_3.1)
subroutine foo()
    integer :: x
    x = 2
    !$omp task shared(x) mergeable
    x = x + 1
    !$omp end task
    !$omp taskwait
    print *, x  ! prints 3
end subroutine
```

This second example shows an incorrect use of the `mergeable` clause. In this example, the created task will access different instances of the variable `x` if the task is not merged, as `x` is `firstprivate`, but it will access the same variable `x` if the task is merged. As a result, the behavior of the program is unspecified, and it can print two different values for `x` depending on the decisions taken by the implementation.
Example tasking.12.c (omp_3.1)

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

Example tasking.12.f90 (omp_3.1)

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

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

```c
#include <string.h>
#include <omp.h>
#define LIMIT 3 /* arbitrary limit on recursion depth */

void check_solution(char *);

void bin_search(int pos, int n, char *state)
{
    if ( pos == n ) {
        check_solution(state);
        return;
    }
    #pragma omp task final( pos > LIMIT ) mergeable
    {
        char new_state[n];
        if (!omp_in_final() ) {
            memcpy(new_state, state, pos);
            state = new_state;
        }
        state[pos] = 0;
        bin_search(pos+1, n, state);
    }
    #pragma omp task final( pos > LIMIT ) mergeable
    {
        char new_state[n];
        if (!omp_in_final() ) {
            memcpy(new_state, state, pos);
            state = new_state;
        }
        state[pos] = 1;
        bin_search(pos+1, n, state);
    }
    #pragma omp taskwait
}
```
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

The following example illustrates the difference between the **if** and the **final** clauses. The **if** clause has a local effect. In the first nest of tasks, the one that has the **if** clause will be undeferred but the task nested inside that task will not be affected by the **if** clause and will be created as usual. Alternatively, the **final** clause affects all **task** constructs in the **final** task region but not the **final** task itself. In the second nest of tasks, the nested tasks will be created as included tasks. Note also that the conditions for the **if** and **final** clauses are usually the opposite.
Example tasking.14.c (omp_3.1)

```c
void bar(void);

void foo ( )
{
    int i;
    #pragma omp task if(0) // This task is undeferred
    {
        #pragma omp task // This task is a regular task
        for (i = 0; i < 3; i++) {
            #pragma omp task // This task is a regular task
            bar();
        }
    }
    #pragma omp task final(1) // This task is a regular task
    {
        #pragma omp task // This task is included
        for (i = 0; i < 3; i++) {
            #pragma omp task // This task is also included
            bar();
        }
    }
}
```

Example tasking.14.f90 (omp_3.1)

```fortran
subroutine foo()
    integer i
    !$omp task if(.FALSE.) ! This task is undeferred
    !$omp task ! This task is a regular task
    do i = 1, 3
       !$omp task ! This task is a regular task
       call bar()
       !$omp end task
    enddo
    !$omp end task
    !$omp end task
    !$omp task final(.TRUE.) ! This task is a regular task
    !$omp task ! This task is included
    do i = 1, 3
       !$omp task ! This task is also included
       call bar()
       !$omp end task
```
S-18  enddo
S-19  !$omp end task
S-20  !$omp end task
S-21  end subroutine
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.

---

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

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

---

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

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

5.3.1 Flow Dependence

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

Example task_dep.1.c (omp_4.0)

```c
#include <stdio.h>

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

Example task_dep.1.f90 (omp_4.0)

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

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

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

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

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.

```
#include <stdio.h>

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

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.

---

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

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

---

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

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

---
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.

### 5.3.5 Matrix multiplication

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

```c
// Assume BS divides N perfectly
void matmul_depend(int N, int BS, float A[N][N], float B[N][N], float C[N][N])
{
  int i, j, k, ii, jj, kk;
  for (i = 0; i < N; i+=BS) {
    for (j = 0; j < N; j+=BS) {
      for (k = 0; k < N; k+=BS) {
        for (ii = i; ii < i+BS; ii++)
          for (jj = j; jj < j+BS; jj++)
            for (kk = k; kk < k+BS; kk++)
      }
    }
  }

  #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++)
}
```

```fortran
S-20 !$omp end task
S-21 !$omp end single
S-22 !$omp end parallel
S-23 end program
```

---

**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.

```
Example task_dep.6.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 shared(y)
    y--; // 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 will create a race condition.

    #pragma omp taskwait // 2nd taskwait
    printf("y=%d\n",y);
}

int main()
{
    #pragma omp parallel
    #pragma omp single
    foo();
    return 0;
}
```
Example task_dep.6.f90 (omp_5.0)

subroutine foo()
    implicit none
    integer :: x, y
    x = 0
    y = 2

    !$omp task depend(inout: x) shared(x)
        x = x + 1 !! 1st child task
    !$omp end task

    !$omp task shared(y)
        y = y - 1 !! 2nd child task
    !$omp end task

    !$omp taskwait depend(in: x) !! 1st taskwait

    print*, "x=", x

    !! Second task may not be finished.
    !! Accessing y here will create a race condition.

    !$omp taskwait !! 2nd taskwait

    print*, "y=", y

end subroutine foo

program p
    implicit none
    !$omp parallel
        !$omp single
            call foo()
        !$omp end single
    !$omp end parallel
end program p

In this example the first two tasks are serialized, because a dependence on the first child is produced by \(x\) with the \texttt{in} dependence type in the \texttt{depend} clause of the second task. However, the generating task at the first \texttt{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 \texttt{in} dependence type within the \texttt{depend} clause of the \texttt{taskwait} construct. The second \texttt{taskwait} (without a \texttt{depend} clause)
is included to guarantee completion of the second task before \( y \) is accessed. (While unnecessary, the \texttt{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.)

---

\textbf{Example task\_dep.7.c (omp\_5.0)}

```c++
#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
    foo();

    return 0;
}
```

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

subroutine foo()
  implicit none
  integer :: x, y
  x = 0
  y = 2
  !$omp task depend(inout: x) shared(x)
  x = x + 1 !! 1st child task
  !$omp end task
  !$omp task depend(in: x) depend(inout: y) shared(x, y)
  y = y - x !! 2nd child task
  !$omp end task
  !$omp taskwait depend(in: x) !! 1st taskwait
  print*, "x=", x
  !! Second task may not be finished.
  !! Accessing y here would create a race condition.
  !$omp taskwait !! 2nd taskwait
  print*, "y=", y
end subroutine foo

program p
  implicit none
  !$omp parallel
  !$omp single
    call foo()
  !$omp end single
  !$omp end parallel
end program p

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 taskwait depend clause 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.

---

Example task_dep.8.c (omp_5.0)

```c
#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; // 2st child task

    #pragma omp taskwait depend(in: x, y)

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

int main()
{
    #pragma omp parallel
    #pragma omp single
    foo();
    return 0;
}
```

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

```fortran
subroutine foo()
    implicit none
    integer :: x, y
    x = 0
    y = 2
    !$omp task depend(inout: x) shared(x)
    x = x + 1 !! 1st child task
    !$omp end task
    !$omp task depend(in: x) depend(inout: y) shared(x, y)
    y = y - x !! 2nd child task
    !$omp end task
    !$omp taskwait depend(in: x,y)
    print*, "x=", x
    print*, "y=", y
end subroutine foo
```

```fortran
program p
    implicit none
    !$omp parallel
    !$omp single
    call foo()
    !$omp end single
    !$omp end parallel
end program p
```

OpenMP Examples Version 5.2 - April 2022
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.

```
#include <stdio.h>

int main()
{
    int a, b, c, d;
    #pragma omp parallel
    #pragma omp single
    {
        #pragma omp task depend(out: c)
        c = 1; /* Task T1 */
        #pragma omp task depend(out: a)
        a = 2; /* Task T2 */
        #pragma omp task depend(out: b)
        b = 3; /* Task T3 */
        #pragma omp task depend(in: a) depend(mutexinoutset: c)
        c += a; /* Task T4 */
        #pragma omp task depend(in: b) depend(mutexinoutset: c)
        c += b; /* Task T5 */
        #pragma omp task depend(in: c)
        d = c; /* Task T6 */
    }
    printf("%d\n", d);
    return 0;
}
```
Fortran

Example task_dep.9.f90 (omp_5.0)

```fortran
program example
  integer :: a, b, c, d
  !$omp parallel
  !$omp single
  !$omp task depend(out: c)
  c = 1 ! Task T1
  !$omp end task
  !$omp task depend(out: a)
  a = 2 ! Task T2
  !$omp end task
  !$omp task depend(out: b)
  b = 3 ! Task T3
  !$omp end task
  !$omp task depend(in: a) depend(mutexinoutset: c)
  c = c + a ! Task T4
  !$omp end task
  !$omp task depend(in: b) depend(mutexinoutset: c)
  c = c + b ! Task T5
  !$omp end task
  !$omp task depend(in: c)
  d = c ! Task T6
  !$omp end task
  !$omp end single
  !$omp end parallel
  print *, d
end program
```

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)

```c
extern int longTaskA(), shortTaskB();
extern int shortTaskAC(int, int), longTaskBC(int, int);
void foo (void)
{
  int a, b, c;
  c = 0;
  #pragma omp parallel
  #pragma omp single
  {
```
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       }

Example task_dep.10.f90 (omp_5.0)

1

subroutine foo
integer :: a,b,c

  c = 0
  !$omp parallel
  !$omp single
  !$omp task depend(out: a)
  a = longTaskA()
  !$omp end task
  !$omp task depend(out: b)
  b = shortTaskB()
  !$omp end task
  !$omp task depend(in: a) depend(mutexinoutset: c)
  c = shortTaskAC(a,c)
  !$omp end task
  !$omp task depend(in: b) depend(mutexinoutset: c)
  c = longTaskBC(b,c)
  !$omp end task
  !$omp end single
  !$omp end parallel
end subroutine foo
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` function 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.

```
#include<stdio.h>

void set_an_element(int *p, int val) {
    *p = val;
}

void print_all_elements(int *v, int n) {
    int i;
    for (i = 0; i < n; ++i) {
        printf("%d, ", v[i]);
    }
    printf("\n");
}

void parallel_computation(int n) {
    int v[n];
    #pragma omp parallel
    #pragma omp single
    {
        int i;
        for (i = 0; i < n; ++i)
            #pragma omp task depend(out: v[i])
    }
```

Example task_dep.11.c (omp_5.0)
Example task_dep.11.f90 (omp_5.0)

subroutine set_an_element(e, val)
  implicit none
  integer :: e, val
  e = val
end subroutine

subroutine print_all_elements(v, n)
  implicit none
  integer :: n, v(n)
  print *, v
end subroutine

subroutine parallel_computation(n)
  implicit none
  integer :: n
  integer :: i, v(n)
  !$omp parallel
  !$omp single
  do i=1, n
    !$omp task depend(out: v(i))
    call set_an_element(v(i), i)
  !$omp end task
  enddo
  !$omp task depend(iterator(it = 1:n), in: v(it))
  !$omp task depend(in: v(:)) Violates Array section restriction.
  call print_all_elements(v, n)
  !$omp end task
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".

Example task_dep.12.c (omp_4.0)

```c
#include <stdio.h>

int main (int argc, char *argv[])
{
    int x = 0;
    #pragma omp parallel
    #pragma omp single
    {
        /* first explicit task */
        #pragma omp task shared(x) depend(out: x)
        x = 1;
        /* second explicit task */
        #pragma omp task shared(x) depend(inout: x) if(0)
        x = 2;
        /* statement executed by parent implicit task
        prints: x = 2 */
        printf("x = %d\n", x);
    }
    return 0;
}
```
Example task_dep.12.f90 (omp_4.0)

```fortran
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
```

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.

Example task_dep.13.c (omp_5.1)

```c
#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"); }
```
#pragma omp task depend(out: a)  // Task 2
{ a++;  
  printf("T2 a=%i\n", a); }

#pragma omp task depend(out: d)  // Task 3
{ d++;  
  printf("T3 d=%i\n", d); }

#pragma omp task depend(inout: omp_all_memory)  // Task 4
{ a++; d++;  
  printf("T4 a=%i d=%i\n", a, d); }

#pragma omp task  // Task 5
{ printf("T5\n"); }

#pragma omp task depend(in: a,d)  // Task 6
{ a++; d++;  
  printf("T6 a=%i d=%i\n", a, d); }
}

/* OUTPUT: ordered {T2,T3 any order}, {T4}, {T6}
T2 a=2
T3 d=2
T4 a=3 d=3
T6 a=4 d=4
*/

Example task_dep.13.f90 (omp_5.1)

program main
  integer :: a=1, d=1

  !$omp parallel masked  num_threads(5)
  !$omp task ! ! Task 1
  write(*,'("T1")')
  !$omp end task

  !$omp task depend(out: a) ! ! Task 2
  a=a+1
S-12 write(*,'("T2 a","i1")') a
S-13 !$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
5.4 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 `omp_fulfill_event` function 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)

```c
#include <omp.h>

void async_work(void (*)(void*), void*);

void work();

int main() {
    int async=1;
    #pragma omp parallel
    #pragma omp masked
    {
        omp_event_handle_t event;
        #pragma omp task detach(event)
        {
            if(async) {
                async_work( (void (*)(void*)) omp_fulfill_event, (void*) event );
            } else {
                work();
                omp_fulfill_event(event);
            }
        }
        // Other work
    }
}
```

OpenMP Examples Version 5.2 - April 2022
Example task_detach.1.f90 (omp_5.0)

```fortran
program main
  use omp_lib
  implicit none
  external :: async_work, work
  logical :: async=.true.
  integer(omp_event_handle_kind) :: event
  !$omp parallel
  !$omp masked
  !$omp task detach(event)
  if(async) then
    call async_work(omp_fulfill_event, event)
  else
    call work()
    call omp_fulfill_event(event)
  endif
  !$omp end task
  !! Other work
  !$omp taskwait
  !$omp end masked
  !$omp end parallel
end program
```

C / C++
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.

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>
#include <aio.h>
#include <errno.h>
#include <signal.h>
#include <omp.h>

#define IO_SIGNAL SIGUSR1

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)(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.
    char data[] = "Written Asynchronously.";
```
struct aiocb cb;
struct sigaction sa;

omp_event_handle_t event;

int fd = open("async_data", O_CREAT|O_RDWR|O_TRUNC,0664);

// Setup async io (aio) control block (cb)
cb.aio_nbytes = sizeof(data)-1;
cb.aio_fildes = fd;
cb.aio_buf = data;
cb.aio_reqprio = 0;
cb.aio_offset = 0;
cb.aio_sigevent.sigev_notify = SIGEV_SIGNAL;
cb.aio_sigevent.sigev_signo = IO_SIGNAL;

// Setup Signal Handler Callback
sigemptyset(&sa.sa_mask);
sa.sa_flags = SA_RESTART | SA_SIGINFO;
sa.sa_sigaction = callback_aioSigHandler; //callback
sigaction(IO_SIGNAL, &sa, NULL);

#pragma omp parallel num_threads(2)
#pragma omp masked
{
    #pragma omp task detach(event) if(0) // TASK1
    {
        cb.aio_sigevent.sigev_value.sival_ptr = (void *) event;
        aio_write(&cb);
    }

    #pragma omp task // TASK2
    work(1);
    #pragma omp task // TASK3
    work(2);

} // Parallel region barrier ensures completion of detachable task.

// Making sure the aio operation completed.
// With OpenMP detachable task the condition will always be false:
while(aio_error(&cb) == EINPROGRESS) {
    printf(" INPROGRESS\n");} //Safeguard

close(fd);
return 0;
/* Any Order:
OUT: I/O completion signal received.
OUT: Executing work(1)
OUT: Executing work(2)
*/
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() call) is created in the parallel region, and later a parallel tree traversal is started (the task executing the root of the recursive compute_tree() calls). While synchronizing tasks at the end of each tree traversal, using the taskgroup construct ensures that the formerly started background task does not participate in the synchronization and is left free to execute in parallel. This is opposed to the behavior of the taskwait construct, which would include the background tasks in the synchronization.

--- C / C++ ---

Example taskgroup.1.c (omp_4.0)

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

Example taskgroup.1.f90 (omp_4.0)

module tree_type_mod
    integer, parameter :: max_steps=100
    type tree_type
        type(tree_type), pointer :: left, right
    end type
contains
    subroutine compute_something(tree)
        type(tree_type), pointer :: tree
        ! some computation
    end subroutine
    recursive subroutine compute_tree(tree)
        type(tree_type), pointer :: tree
        if (associated(tree%left)) then
            !$omp task
            call compute_tree(tree%left)
        !$omp end task
        endif
        if (associated(tree%right)) then
            !$omp task
            call compute_tree(tree%right)
        !$omp end task
    end subroutine
end module tree_type_mod
endif
$omp task
   call compute_something(tree)
$omp end task
end subroutine
end module
program main
use tree_type_mod
   type(tree_type), pointer :: tree
   call init_tree(tree);
$omp parallel
   $omp single
   $omp task
   call start_background_work()
   $omp end task
   $omp taskgroup
   $omp task
   call compute_tree(tree)
$omp end task
$omp end taskgroup ! wait on tree traversal in this step
   call check_step()
edo i=1, max_steps
   $omp taskgroup ! wait on tree traversal in this step
   $omp task
   call compute_tree(tree)
   $omp end task
   $omp end taskgroup
   call check_step()
edo
$omp end single
$omp end parallel ! only now is background work required to be complete
   call print_results()
end program
5.6 taskyield Construct

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

Example taskyield.1.c (omp_3.1)
```c
#include <omp.h>

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

Example taskyield.1.f90 (omp_3.1)
```fortran
subroutine foo ( lock, n )
use omp_lib
integer (kind=omp_lock_kind) :: lock
integer n
integer i

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

!$omp end task

der do

der 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)

```c
void long_running_task(void);
void loop_body(int i, int j);

void parallel_work(void) {
    int i, j;
    #pragma omp taskgroup
    {
        #pragma omp task
        long_running_task(); // can execute concurrently
        #pragma omp taskloop private(j) grainsize(500) nogroup
        for (i = 0; i < 10000; i++) { // can execute concurrently
            for (j = 0; j < i; j++) {
                loop_body(i, j);
            }
        }
    }
}
```

--- C / C++ ---
Example taskloop.1.f90 (omp_4.5)

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

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 $x_1 = 16384 (T*N)$ and $x_2 = 1024 (N)$. 
Example taskloop.2.c (omp_4.5)

```c
#include <stdio.h>

#define T 16
#define N 1024

void parallel_work() {
    int x1 = 0, x2 = 0;

    #pragma omp parallel shared(x1,x2) num_threads(T)
    {
        #pragma omp taskloop
        for (int i = 0; i < N; ++i) {
            #pragma omp atomic
            x1++; // executed T*N times
        }
        #pragma omp single
        #pragma omp taskloop
        for (int i = 0; i < N; ++i) {
            #pragma omp atomic
            x2++; // executed N times
        }
    }
    printf("x1 = %d, x2 = %d\n", x1, x2);
}
```

Example taskloop.2.f90 (omp_4.5)

```fortran
subroutine parallel_work
    implicit none
    integer :: x1, x2
    integer :: i
    integer, parameter :: T = 16
    integer, parameter :: N = 1024

    x1 = 0
    x2 = 0

    !$omp parallel shared(x1,x2) num_threads(T)
    !$omp taskloop
    do i = 1,N
        !$omp atomic
```
x1 = x1 + 1 ! executed T*N times
end do
!$omp end taskloop

!$omp single
!$omp taskloop
do i = 1,N
  !$omp atomic
  x2 = x2 + 1 ! executed N times
  !$omp end atomic
end do
!$omp end taskloop
!$omp end single
!$omp end parallel

write (*,'(A,I0,A,I0)') 'x1 = ', x1, ', x2 = ', x2
end subroutine
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
#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
}
```

---

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

```
program main

integer, parameter :: N=100
integer :: i, a(N), b(N), c(N)

do i=1,N           !! initialize
  b(i) = i
  c(i) = i
enddo

!$omp parallel
!$omp masked
!$omp taskloop    !! taskloop 1
do i=1,N
  a(i) = b(i) + c(i)
enddo
!$omp end taskloop
!$omp end masked
!$omp end parallel

!$omp parallel masked taskloop    !! taskloop 2
do i=1,N
  b(i) = a(i) + c(i)
enddo
!$omp end parallel masked taskloop

!$omp parallel masked taskloop simd!! taskloop 3
do i=1,N
  c(i) = a(i) + b(i)
enddo
!$omp end parallel masked taskloop simd

print*, c(1), c(N)   !! 5 and 500

end program
```
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 to from 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.

---

Example target.1.c (omp_4.0)

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

---

Example target.1.f90 (omp_4.0)

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

---
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)

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

--- C / C++ ---

Example target.2.f90 (omp_4.0)

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

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

C / C++

The to and from map-types allow programmers to optimize data motion. Since data for the v1 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.
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.
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 \( v2(\cdot:N) \).

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

A more realistic situation in which an assumed-size array is passed to \texttt{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 \( v2(\cdot:N) \).

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

Example target.5.c (omp_4.0)

```c
#define THRESHOLD1 1000000
#define THRESHOLD2 1000
extern void init(float*, float*, int);
extern void output(float*, int);

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

The if clause with the target modifier applies to the target component of the combined directive, and the if clause with the parallel modifier applies to the parallel component of the combined directive.
```c
#include <omp.h>

#pragma omp target parallel for \
  if(target: N>THRESHOLD1) if(parallel: N>THRESHOLD2) \
  map(to: v1[0:N], v2[:N]) map(from: p[0:N])
for (i=0; i<N; i++)
  p[i] = v1[i] * v2[i];
```

```fortran
Example target.6.f90 (omp_4.5)
```
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)

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

#define N 100

#pragma omp requires reverse_offload

void error_handler(int wrong_value, int index)
{
    printf(" Error in offload: A[%d]=%d\n", index,wrong_value);
    printf(" Expecting: A[i ]=i\n");
    exit(1);

    // output: Error in offload: A[99]=-1
    // Expecting: A[i ]=i
}

#pragma omp declare target device_type(host) enter(error_handler)

int main()
{
    int A[N];
    for (int i=0; i<N; i++) A[i] = i;
```
Example target_reverse_offload.7.f90 (omp_5.0)

```c
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
```

```
!$omp requires reverse_offload

subroutine error_handler(wrong_value, index)
    integer :: wrong_value,index
    !$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
  use omp_lib
  integer, parameter :: N=100
  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
```

CHAPTER 6. DEVICES  153
endif
end do
!$omp end target

end program
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 introduced in OpenMP 5.0. The implicit behavior is specified as alloc, to, from, tofrom, firstprivate, none, default or present, and is applied to a variable-category, where scalar, aggregate, allocatable, and pointer are the variable categories.

In OpenMP, a “category” has a common data-mapping and data-sharing behavior for variable types within the category. 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, with intrinsic type, and excludes character type.

Also, aggregate refers to arrays and structures (C/C++) and derived types (Fortran). Fortran has the additional category of allocatable.

In the example below, the first target construct uses defaultmap clauses to set data-mapping and possibly data-sharing attributes that reproduce the default implicit mapping (data-mapping and data-sharing attributes). That is, if the defaultmap clauses were removed, the results would be identical.

In the second target construct all implicit behavior is removed by specifying the none implicit behavior in the defaultmap clause. Hence, all variables must be explicitly mapped. In the C/C++ code a scalar (s), an array (A) and a structure (S) are explicitly mapped tofrom. The Fortran code uses a derived type (D) in lieu of structure.

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

In the fourth target construct all arrays, structures (C/C++) and derived types (Fortran) are mapped with firstprivate data-sharing behavior by a defaultmap clause with an aggregate variable category. For the H allocated array in the Fortran code, the allocable category must be used in a separate defaultmap clause to acquire firstprivate data-sharing behavior (H has the Fortran allocatable attribute).

---

Example target_defaultmap.1.c (omp_5.0)

```c
#include <stdlib.h>
#include <stdio.h>
#define N 2

int main()
{
    typedef struct S_struct { int s; int A[N]; } S_struct_t;
```
int s; // scalar int variable (scalar)
int A[N]; // aggregate variable (array)
S_struct_t S; // aggregate variable (structure)
int *ptr; // scalar, pointer variable (pointer)

int s1, s2, s3;

// Initialize everything to zero;
s=2; s1=s2=s3=0;
A[0]=0; A[1]=0;
S.s=0; S.A[0]=0; S.A[1]=0;

// Target Region 1

// Uses defaultmap to set scalars, aggregates &
// pointers to normal defaults.
#pragma omp target
defaultmap(firstprivate: scalar) // could also be default
defaultmap(tofrom: aggregate) // could also be default
defaultmap(default: pointer) // must be default
map(ptr2m[:N])
{
s = 3; // SCALAR firstprivate, value not returned
A[0] = 3; A[1] = 3; // AGGREGATE array, default map tofrom
S.s = 2;
S.A[0] = 2; S.A[1] = 2;
ptr = &A[0]; // POINTER is private
ptr[0] = 2; ptr[1] = 2;
}
if(s==2 && A[0]==2 && S.s==2 && S.A[0]==2)
printf(" PASSED 1 of 4\n");

// Target Region 2

// no implicit mapping allowed.
#pragma omp target defaultmap(none) map(tofrom: s, A, S)
{
s +=5; // All variables must be explicitly mapped
A[0] +=5; A[1] +=5;
S.s +=5;
}
if(s==7 && A[0]==7 && S.s==7 && S.A[0]==7)
printf(" PASSED 2 of 4\n");

// Target Region 3
// defaultmap & explicit map with variables in same category
s1=s2=s3=1;
#pragma omp defaultmap(tofrom: scalar) map(firstprivate: s1,s2)
{
    s1 += 5;        // firstprivate (s1 value not returned to host)
s2 += 5;        // firstprivate (s2 value not returned to host)
s3 += s1 + s2;  // mapped as tofrom
}
if(s1==1 && s2==1 && s3==13 ) printf(" PASSED 3 of 4\n");

// Target Region 4
A[0]=0; A[1]=0;
S.A[0]=0; S.A[1]=0;

// arrays and structure are firstprivate, and scalars are from
#pragma omp target defaultmap(firstprivate: aggregate) \
    map(from: s1, s2)
{
    A[0]+=1; S.A[0]+=1; //Aggregate changes not returned to host
    s1 = A[0]+S.A[0];  //s1 value returned to host
    s2 = A[1]+S.A[1];  //s1 value returned to host
}
if( A[0]==0 && S.A[0]==0 && s1==2 ) printf(" PASSED 4 of 4\n");

1

Example target_defaultmap.1.f90 (omp_5.0)
integer, target :: A(N) !! AGGREGATE: Array

type(DDT_sA) :: D !! AGGREGATE: Derived Data Type (D)

integer, allocatable :: H(:) !! ALLOCATABLE: Heap allocated array

integer, pointer :: ptrA(:) !! POINTER: points to Array

!! Assign values to scalar, Array, Allocatable, and Pointers

s = 2;
s1 = 0; s2 = 0; s3 = 0
D%s = 0; D%A(1) = 0; D%A(2) = 0
A(1) = 0; A(2) = 0

allocate(H(2))

H(1) = 0; H(2) = 0

!! Target Region 1
!! Using defaultmap to set scalars, aggregates &
!! pointers and allocatables to normal defaults.

!! SCALAR firstprivate, val not returned
A(1) = 3 !! AGGREGATE array, default map tofrom
A(2) = 3

D%s = 2 !! AGGR. Derived Type, default map tofrom
D%A(1) = 2; D%A(2) = 2

H(1) = 2; H(2) = 2 !! ALLOCATABLE, default map tofrom

ptrA => A !! POINTER is private
ptrA(1) = 2; ptrA(2) = 2

!! Target Region 2
!! no implicit mapping allowed

if(s == 2 .and. A(1) == 2 .and. D%s == 2 .and. D%A(1) == 2 .and. H(1) == 2) &
print*, "PASSED 1 of 4"

!! Target Region 1
!! Using defaultmap to set scalars, aggregates &
!! pointers and allocatables to normal defaults.

!! SCALAR firstprivate, val not returned
A(1) = 3 !! AGGREGATE array, default map tofrom
A(2) = 3

D%s = 2 !! AGGR. Derived Type, default map tofrom
D%A(1) = 2; D%A(2) = 2

H(1) = 2; H(2) = 2 !! ALLOCATABLE, default map tofrom

ptrA => A !! POINTER is private
ptrA(1) = 2; ptrA(2) = 2

!! Target Region 2
!! no implicit mapping allowed

if(s == 2 .and. A(1) == 2 .and. D%s == 2 .and. D%A(1) == 2 .and. H(1) == 2) &
print*, "PASSED 1 of 4"
s=s+5 !! All variables must be explicitly mapped
A(1)=A(1)+5; A(2)=A(2)+5
D%s=D%s+5
D%A(1)=D%A(1)+5; D%A(2)=D%A(2)+5

!! Target Region 3
!!defaultmap & explicit map with variables in same category
s1=1; s2=1; s3=1
!
if(s==7 .and. A(1)==7 .and. D%s==7 .and. D%A(1)==7) &
print*," PASSED 2 of 4"
!

!! Target Region 4
A(1)=0; A(2)=0
D%A(1)=0; D%A(2)=0
H(1)=0; H(2)=0
!! non-allocated arrays & derived types are in AGGREGATE cat.
!! Allocatable Arrays are in ALLOCATABLE category
!! Scalars are explicitly mapped from
!
if(A(1)==0 .and. D%A(1)==0 .and. H(1)==0 .and. s1==3) &
print*," PASSED 4 of 4"
!
end program
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 \textit{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, \textit{ptr2} can be manipulated on the device; however, as an explicitly mapped pointer, \textit{ptr1} becomes an \textit{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 \texttt{target} region.

As a comparison, note that the \textit{array} array is automatically mapped, since the compiler knows the extent of the array.

The pointer \textit{ptr3} is used inside the \texttt{target} construct, but it does not appear in a data-mapping or data-sharing clause. Nor is there a \texttt{defaultmap} clause on the construct to indicate what its implicit data-mapping or data-sharing attribute should be. For such a case, \textit{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 \textit{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 \textit{ptr3} is private, the value to which it is assigned in the \texttt{target} region is not returned into the original \textit{ptr3} on the host.

---

\textbf{Example target\_ptr\_map\_l.c (omp\_5.0)}

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

int main()
{
    int *ptr1;
    int *ptr2;
    int *ptr3;
    int aray[N];
    ptr1 = (int *)malloc(sizeof(int)*N);
    ptr2 = (int *)malloc(sizeof(int)*N);
    
    #pragma omp target map(ptr1, ptr1[:N]) map(ptr2[:N] )
    {
        for (int i=0; i<N; i++)
        {
            ptr1[i] = i;
            ptr2[i] = i;
            aray[i] = i;
        }
    }
```

---

CHAPTER 6. DEVICES 161
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.)

---

Example target_ptr_map.2.c (omp_5.1)

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

#pragma omp begin declare target
  int *p;
  extern void use_arg_p(int *p, int n);
  extern void use_global_p(int n);
#pragma omp end declare target

int main()
{  
  int i;
  
  ptr3=(int *)malloc(sizeof(int)*N); // ptr3 is firstprivate
  for (int i=0; i<N; i++) ptr3[i] = 5;
  for (int i=0; i<N; i++) ptr1[i] += ptr3[i];
  free(ptr3); // explicitly free allocated storage on device
}

printf(" %d %d\n",ptr1[1],ptr2[1]);
// 6 9

free(ptr1);
free(ptr2);
return 0;
```
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 \( p_2 \) 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 \( p_2 \) 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 \( p_2 \) is pointing) on the construct, the order of this map relative to the initialization of \( p_2 \) is unspecified. Therefore, the initial value of \( p_2 \) will also be undefined.

Thus, referencing values via either \( p_1 \) or \( p_2 \) inside the **target** region would be invalid.

---

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

```c
#define N 100

int x[N], y[N];

#pragma omp begin declare target
int *p1;
#pragma omp end declare target
int *p2;

int foo()
{
    p1 = &x[0];
    p2 = &y[0];

    // Explicitly map array section x[:N]
    #pragma omp target enter data map(x[:N])

    #pragma omp target // as if .. map(p1) map(p1[:0]) map(p2[:0]) map(y)
    {
        // Accessing the mapped arrays x,y is OK here.
        x[0] = 1;
        y[1] = 2;

        // Pointer attachment for p1 does not occur here
        // because p1[:0] does not allocate a new array section and
        // array x is present on the target construct as it was mapped
        // before by the target enter data directive.
        p1[0] = 3;  // accessing p1 is undefined

        // The initial value of p2 in the target region is undefined
        // because map(y) may occur after map(p2[:0]).
    }
}
```

---
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.

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

```c
#define N 100

int x[N], y[N];
#pragma omp begin declare target
int *p1;
#pragma omp end declare target
int *p2;

int foo()
{
    p1 = &x[0];
    p2 = &y[0];

    // Explicitly map array section y[:N]
    #pragma omp target enter data map(y[:N])
    #pragma omp target map(x[:N]) map(p1[:N]) map(p2[:0])
    {
        // Accessing the mapped arrays x,y is OK here.
        x[0] = 1;
        y[1] = 2;

        // Pointer attachment for p1 occurs here when array x is mapped
        // on the target construct (as p1 = &x[0] on the device)
        p1[0] = 3; // accessing p1 is OK

        // p2 in the target region is initialized to &y[0]
        p2[1] = 4; // accessing p2 is OK
    }
```

```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 next version of the OpenMP Specification, a false presence check without the firstprivate clause will cause the pointer to retain its original value.

```c
#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]) )
  {
```
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.

Example target_ptr_map.5.c (omp_5.2)

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

typedef struct {
    int *ptr;
    int buf_size;
} T;

#pragma omp declare mapper(deep_copy: T s) map(s, s.ptr[:s.buf_size])

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

int main()
{
    const int n = 1000;
    const int buf_size = sizeof(int) * n;
    T s = { 0, buf_size };
    const int dev = omp_get_default_device();
    s.ptr = (int *)malloc(buf_size);
    const int accessible = omp_target_is_accessible(s.ptr, s.buf_size, dev);

    #pragma omp metadirective 
    when(user={condition(accessible)}: target) 
    otherwise(target map(mapper(deep_copy), tofrom:s))
    {
    do_work(s.ptr, n);
    }

    free(s.ptr);
```

```c
S-1
S-2
S-3
S-4
S-5
S-6
S-7
S-8
S-9
S-10
S-11
S-12
S-13
S-14
S-15
S-16
S-17
S-18
S-19
S-20
S-21
S-22
S-23
S-24
S-25
S-26
S-27
S-28
S-29
S-30
S-31
```
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
## 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)

```c
#include <stdio.h>
#include <stdlib.h>
#define N 100
#define BAZILLION 2000000

struct foo {
    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 S;
    int i;
    S.a = 2.0;
    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  }

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.

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  SAXPY(float arg_a, float arg_b){ a=arg_a; b=arg_b; }
S.11  void driver();
S.12  void saxpyfun(float *p);
S.13  }
S.14
S.15  #pragma omp begin declare target
S.16  void SAXPY::saxpyfun(float *q)
S.17  {
S.18      for(int i=0; i<N; i++)
S.19          buffer[i] = q[i]*a + b;
S.20  }
S.21  #pragma omp end declare target
S.22
S.23  void SAXPY::driver()
S.24  {
S.25      p = (float *) malloc(N*sizeof(float));
S.26      for(int i=0; i<N; i++) p[i]=i;
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). Consequently, 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.

---

Example target_struct_map.3.c (omp_5.1)
void saxpyfun(struct foo *S) {
    int i;
    for(i=0; i<N; i++)
        S->p[i] = S->p[i] * S->a + S->b; // S->p[i] invalid
}

int main() {
    struct foo S1, S2;
    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;
    // No pointer attachment for S1.p here
    #pragma omp target data map(S1.p[:N])
    #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;
    // No pointer attachment for S2.p here either
    #pragma omp target data map(S2.p[:N])
    #pragma omp target map(S2.p, S2.a, S2.b) // implicit map of S2
    saxpyfun(&S2);

    // These print statement may not execute because the
    // above code is invalid
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.
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);
Case 3

```c
S3.a = 2.0;
S3.b = 4.0;
S3.p = (float *)malloc(sizeof(float)*N);
for(i=0; i<N; i++) S3.p[i] = i;

// The target construct results in pointer attachment for S3.p.
// Note that S3.p is implicitly mapped due to the implicit map of S3
// (but corresponding storage is NOT created for members buffera,
// bufferb, and x).
#pragma omp target map(S3.p[:N], S3.a, S3.b) // implicit map of S3
saxpyfun(&S3);
```

```c
printf(" %4.0f %4.0f\n", S1.p[0], S1.p[N-1]); //OUT1 4 202
printf(" %4.0f %4.0f\n", S2.p[0], S2.p[N-1]); //OUT2 4 202
printf(" %4.0f %4.0f\n", S3.p[0], S3.p[N-1]); //OUT3 4 202
```

```c
free(S1.p);
free(S2.p);
free(S3.p);
return 0;
```
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 OpenMP 6.0, this restriction may be relaxed to also guarantee updates without the always modifier.

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)

```fortran
program main
  implicit none
  integer :: i

  integer, save, allocatable :: d(:)
  !$omp declare target(d)

  integer, allocatable :: a(:)
  integer, allocatable :: b(:)
  integer, allocatable :: c(:)

  allocate(a(4))
  !$omp target ! Target 1
  a(:) = 4
  !$omp end target
  print *, a ! prints 4*4
```

---
allocate(b(4))
!$omp target map(b) ! Target 2
b(:) = 4
!$omp end target
print *, b ! prints 4*4

!$omp target data map(c)
allocate(c(4), source=[1,2,3,4])
!$omp target map(always,tofrom:c) ! Target 3
c(:) = 4
!$omp end target
print *, c ! prints 4*4
deallocate(c)

!$omp end target data
allocate(d(4), source=[1,2,3,4])
!$omp target map(always,tofrom:d) ! Target 4
d(:) = d(:) + [ ( i,i=size(d),1,-1) ]
!$omp end target
print *, d ! prints 4*5
deallocate(a, b, d)

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 reshapess 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)

program main
implicit none
integer, allocatable :: a(:,,:), b(:,), c(:)
integer :: x(10,2)
allocate(a(2,10))

CHAPTER 6. DEVICES  177
S-8  !$omp target
S-9    a = x           ! Reshape (or resize) NOT ALLOWED (implicit change)
S-10
S-11  deallocate(a)   ! Allocation status change of "a" NOT ALLOWED.
S-12
S-13  allocate(b(20)) ! Allocation of b *
S-14
S-15  c = 10           ! Intrinsic assignment allocates c *
S-16
S-17  ! * Since an explicit deallocation for b and c does not occur before
S-18  ! the end of the target region, the PROGRAM BEHAVIOR IS UNSPECIFIED.
S-19  !$omp end target
S-20
S-21  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)

S-1  module corfu
S-2  contains
S-3    subroutine foo(ain,bout)
S-4      implicit none
S-5      integer, allocatable, intent( in) :: ain(:)
S-6      integer, allocatable, intent(out) :: bout(:) !"out" causes de/realloc
S-7      !$omp declare target
S-8      bout = ain
S-9    end subroutine
S-10  end module
S-11
S-12  program main
S-13    use corfu
S-14      implicit none
S-15
S-16      integer, allocatable :: a(:)
S-17      integer, allocatable :: b(:)
S-18      allocate(a(10),b(10))
S-19      a(:)=10


```fortran
S-20 b(:)=10
S-21 !omp target
S-22 call foo(a,b) !ERROR: b deallocation/reallocation not allowed
S-23 ! in target region
S-24 !$omp end target
S-25 end program
S-26
```

**Chapter 6. Devices**
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
void foo ()
{
    int A[30];
    #pragma omp target data map( A[0:4] )
    {
        /* Cannot map distinct parts of the same array */
        #pragma omp target map( A[7:20] )
        {
            A[2] = 0;
        }
    }
}
```

```fortran
subroutine foo()
integer :: A(30)
A = 1
!$omp target data map( A(1:4) )
! Cannot map distinct parts of the same array
!$omp target map( A(8:27) )
A(3) = 0
!$omp end target
!$omp end target data
end subroutine
```
This example shows the invalid usage of two separate sections of the same array inside of a target construct.

C / C++

Example array_sections.2.c (omp_4.0)

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

C / C++

Fortran

Example array_sections.2.f90 (omp_4.0)

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

---

### Example array_sections.3.c (omp_4.0)

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

---

### Example array_sections.3.f90 (omp_4.0)

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

---
This example shows the valid usage of a wholly contained array section of an already mapped array section inside of a `target` construct.

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

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

CHAPTER 6. DEVICES 183
6.7 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 example identifies problem cases in which the restrictions are not followed (for Unified Shared Memory, as prescribed by the `requires` directive).

The first section illustrates the restriction that when mapping an object for the first time, the static and dynamic types must match.

For the first target region the behavior of the implicit map of `ar` is not specified– its static type (A) doesn’t match its dynamic type (D). Hence access to the virtual functions is undefined. However, the second target region can access `D::vf()` since the object to which `ap` points is not mapped and therefore the restriction does not apply.

The second section 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."

An instantiation of a polymorphic class (A) occurs in the `target` region, and access of its virtual function is incorrectly attempted on the host (another device). However, once the object is deleted on the target device and instantiated on the host, access within the next `target` region is permitted.

```
#include <iostream>
#pragma omp requires unified_shared_memory
#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()
{
  // Section 1 --------------------------------------------------------
  D d;       // D derives from A, and A::vf() is virtual
  A &ar = d; // reference to Derived object d
```
6.8 Array Shaping

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):

\[
([s_1][s_2] \ldots [s_n])\text{pointer}
\]
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 $([\text{nx}][\text{ny}+2])$ casts pointer variable $a$ to a 2-dimentional array of size $nx \times (ny+2)$. The resulting array is then accessed as array sections (such as $[0: \text{nx}][1]$ and $[0: \text{nx}][\text{ny}]$) 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)*

```c
#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)])
    {
        // do work on the device
        #pragma omp target // map(a[0:nx*(ny+2)]) is optional here
        do_work(a, nx, ny);

        // update boundary points (two columns of 2D array) on the host
        // pointer is shaped to 2D array using the shape-operator
        #pragma omp target update from( (([[nx][ny+2]])a)[0:nx][1], \
            (([[nx][ny+2]])a)[0:nx][ny] )

        // exchange ghost points with neighbors
        exch_data(a, nx, ny);

        // update ghost points (two columns of 2D array) on the device
        // pointer is shaped to 2D array using the shape-operator
        #pragma omp target update to( (([[nx][ny+2]])a)[0:nx][0], \
            (([[nx][ny+2]])a)[0:nx][ny+1] )

        // perform other work on the device
        #pragma omp target // map(a[0:nx*(ny+2)]) is optional here
        other_work(a, nx, ny);
    }
}
```

C / C++
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.

```
module m
  interface
    subroutine do_work(a, nx, ny)
      !$omp declare target enter(do_work)
      integer, intent(in) :: nx, ny
      double precision a(0:nx+1,ny)
    end subroutine do_work
    subroutine other_work(a, nx, ny)
      !$omp declare target enter(other_work)
      integer, intent(in) :: nx, ny
      double precision a(0:nx+1,ny)
    end subroutine other_work
    subroutine exch_data(a, nx, ny)
      integer, intent(in) :: nx, ny
      double precision a(0:nx+1,ny)
    end subroutine exch_data
  end interface
end module m

subroutine array_shaping(a, nx, ny)
  use m
  implicit none
  integer, intent(in) :: nx, ny
  double precision a(0:nx+1,ny)
  !$omp target data map(a)
  !$omp target ! map(a) is optional here
  call do_work(a, nx, ny)
  !$omp end target
  ! update boundary points (two rows of 2D array) on the host.
  ! data transferred are noncontiguous
  !$omp target update from( a(1,1:ny), a(nx,1:ny) )
```

```
Example array_shaping.1.f90 (omp_5.2)
```
S-40                     ! exchange ghost points with neighbors
S-41                     call exch_data(a, nx, ny)
S-42                     ! update ghost points (two rows of 2D array) on the device.
S-43                     ! data transferred are noncontiguous
S-44                     !$omp target update to( a(0,1:ny), a(nx+1,1:ny) )
S-45                     ! perform other work on the device
S-46                     !$omp target   ! map(a) is optional here
S-47                     call other_work(a, nx, ny)
S-48                     !$omp end target
S-49                     !$omp end target data
S-50                     end subroutine
6.9 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++
#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])

void init(myvec_t *s);

int main(){
    myvec_t s;
    s.data = (double *)calloc(N,sizeof(double));
    s.len = N;
    #pragma omp target
    init(&s);
    printf("s.data[%d]=%lf
",N-1,s.data[N-1]); //s.data[99]=99.000000
}

void init(myvec_t *s)
{
    for(int i=0; i<s->len; i++) s->data[i]=i;
}
```

Example target_mapper.1.c (omp_5.0)
Example target_mapper.1.f90 (omp_5.0)

```fortran
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
  s%data = [ (i, i=1,s%len) ]
end subroutine
```

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\) \text{ and } \text{map}(\text{mapper(bottom_id)}, \text{tofrom: } a, b).\] Each target offload is allowed to execute concurrently on two different devices (0 and 1) through the \text{nowait} clause.

The Fortran code uses the \text{left_id} and \text{right_id} map identifiers in the \text{map(mapper(left_id), tofrom: a, b)} \text{ and } \text{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 \text{declare mapper} directive.

Note, the \text{is} and \text{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.

\begin{example}
\begin{verbatim}
C / C++

Example target_mapper.2.c (omp_5.1)

```
#include <stdio.h>

// N MUST BE EVEN
#define N 100

typedef struct dzmat
{
    double r_m[N][N];
    double i_m[N][N];
} dzmat_t;

#pragma omp declare mapper( top_id: dzmat_t v) \
    map(v.r_m[0:N/2][0:N], \ 
        v.i_m[0:N/2][0:N] )

#pragma omp declare mapper(bottom_id: dzmat_t v) \ 
    map(v.r_m[N/2:N/2][0:N], \ 
        v.i_m[N/2:N/2][0:N] )

//initialization
void dzmat_init(dzmat_t *z, int is, int ie, int n);

//matrix add: c=a+b
void host_add( dzmat_t *a, dzmat_t *b, dzmat_t *c, int n);

int main()
{
    dzmat_t a,b,c;
    int is,ie;
    is=0; ie=N/2-1;     //top N/2 rows on device 0
    #pragma omp target map(mapper(top_id), tofrom: a,b) device(0) \ 
    firstprivate(is,ie) nowait
    {
        dzmat_init(&a,is,ie,N);
        dzmat_init(&b,is,ie,N);
    }
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
```
module complex_mats

integer, parameter :: N=100 ! N must be even
type dzmat_t
  double precision :: r_m(N,N), i_m(N,N)
end type

!$omp declare mapper(left_id: dzmat_t :: v) map(v%r_m(N, 1:N/2), 
!$omp& v%i_m(N, 1:N/2))

!$omp declare mapper(right_id: dzmat_t :: v) map(v%r_m(N,N/2+1:N), 
!$omp& v%i_m(N,N/2+1:N))

end module

program main
  use complex_mats
type(dzmat_t) :: a,b,c
external dzmat_init, host_add ! initialization and matrix add: a=b+c

integer :: is,ie

is=1; ie=N/2 ! left N/2 columns on device 0
!$omp target map(mapper(left_id), tofrom: a,b) device(0) 
!$omp& firstprivate(is,ie) nowait
  call dzmat_init(a,is,ie)
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.
module my_structures
    type myvec_t
        integer :: len
        double precision, pointer :: data(:)
    end type

    type mypoints_t
        type(myvec_t) :: scratch
        type(myvec_t), pointer :: x(:)
        double precision :: hostonly_data[500000]
    end type
end module

program main
    mypoints_t P;

    init_mypts_array(&P, N);

    !$omp declare mapper(mypoints_t :: v) &
    !$omp& map(v%x, v%x(1)) map(alloc:v%scratch)

    eval_mypts_array(&P, N);
end program
use my_structures
external init_mypts_array, eval_mypts_array

type(mypoints_t) :: P

call init_mypts_array(P)

!$omp target map(P)
call eval_mypts_array(P)

end program
6.10 target data Construct

6.10.1 Simple target data Construct

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

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

Example target_data.1.c (omp_4.0)
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)

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

### 6.10.2 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.
Example target_data.2.c (omp_4.0)

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

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

Example target_data.2.f90 (omp_4.0)

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

---

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

```c
#include <math.h>
#define COLS 100

void gramSchmidt(float Q[][COLS], const int rows)
{
    int cols = COLS;
    #pragma omp target data map(Q[0:rows][0:cols])
    for(int k=0; k < cols; k++)
    {
        double tmp = 0.0;
        #pragma omp target map(tofrom: tmp)
        #pragma omp parallel for reduction(+:tmp)
        for(int i=0; i < rows; i++)
            tmp += (Q[i][k] * Q[i][k]);
        tmp = 1/sqrt(tmp);
        #pragma omp target
        #pragma omp parallel for
        for(int i=0; i < rows; i++)
            Q[i][k] *= tmp;
    }
}
```
subroutine gramSchmidt(Q,rows,cols)
  integer :: rows,cols, i,k
  double precision :: Q(rows,cols), tmp
  !$omp target data map(Q)
  do k=1,cols
    tmp = 0.0d0
    !$omp target map(tofrom: tmp)
    !$omp parallel do reduction(+:tmp)
      do i=1,rows
        tmp = tmp + (Q(i,k) * Q(i,k))
      end do
    !$omp end target
    tmp = 1.0d0/sqrt(tmp)
    !$omp target
    !$omp parallel do
      do i=1,rows
        Q(i,k) = Q(i,k)*tmp
      enddo
    !$omp end target
  end do
  !$omp end target data
end subroutine

Example target_data.3.f90 (omp_4.0)
6.10.3 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)

```c
void vec_mult(float*, float*, float*, int);
extern void init(float*, float*, int);
extern void output(float*, int);

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

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

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 construct’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).

Example target_data.4.f90 (omp_4.0)

module mults
contains
subroutine foo(p0,v1,v2,N)
real,pointer,dimension(:) :: p0, v1, v2
integer :: N,i

call init(v1, v2, N)
!$omp target data map(to: v1, v2) map(from: p0)
call vec_mult(p0,v1,v2,N)
!$omp end target data

call output(p0, N)
end subroutine

subroutine vec_mult(p1,v3,v4,N)
real,pointer,dimension(:) :: p1, v3, v4
integer :: N,i

!$omp target map(to: v3, v4) map(from: p1)
!$omp parallel do
do i=1,N
  p1(i) = v3(i) * v4(i)
end do
!$omp end target
end subroutine
end module

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

Example target_data.5.cpp (omp_4.0)

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 \texttt{vec\_mult}, \( p1 \), \( v3 \), and \( v4 \) are references to the \( p0 \), \( v1 \), and \( v2 \) arrays, respectively. The target construct’s device data environment inherits the arrays \( p0 \), \( v1 \), and \( v2 \) from the enclosing target data construct’s device data environment. Thus, \( p1 \), \( v3 \), and \( v4 \) are already present in the device data environment.
Example target_data.5.f90 (omp_4.0)

```fortran
module my_mult
contains
subroutine foo(p0,v1,v2,N)
  real,dimension(:) :: p0, v1, v2
  integer :: N,i
  call init(v1, v2, N)
  !$omp target data map(to: v1, v2) map(from: p0)
  call vec_mult(p0,v1,v2,N)
  !$omp end target data
  call output(p0, N)
end subroutine

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

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

### 6.10.4 target data Construct with if Clause

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

In the following example, the if clause on the **target data** construct indicates that if the variable \( N \) is smaller than a given threshold, then the **target data** construct will not create a device data environment.
The **target** constructs enclosed in the **target data** region must also use an **if** clause on the same condition, otherwise the pointer variable \( p \) is implicitly mapped with a map-type of **tofrom**, but the storage location for the array section \( p[0:N] \) will not be mapped in the device data environments of the **target** constructs.

```
```

**Example target_data.6.c (omp_4.0)**

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

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

\begin{verbatim}
module params
integer,parameter :: THRESHOLD=1000000
end module

subroutine vec_mult(p, v1, v2, N)
  use params
  real :: p(N), v1(N), v2(N)
  integer :: i
  call init(v1, v2, N)
  !$omp target data if(N>THRESHOLD) map(from: p)
  !$omp target if(N>THRESHOLD) map(to: v1, v2)
  !$omp parallel do
  do i=1,N
    p(i) = v1(i) * v2(i)
  end do
  !$omp end target
  call init_again(v1, v2, N)
  !$omp target if(N>THRESHOLD) map(to: v1, v2)
  !$omp parallel do
  do i=1,N
    p(i) = p(i) + v1(i) * v2(i)
  end do
  !$omp end target
  !$omp end target data
  call output(p, N)
end subroutine
\end{verbatim}
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 \( p[0:N] \).

---

Example target_data.7.c (omp_4.0)

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

---
The \texttt{if} clauses work the same way for the following Fortran code. When the \texttt{if} clause conditional expression on the \texttt{target} construct evaluates to \texttt{false}, the \texttt{target} region will execute on the host device. However, the \texttt{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 \texttt{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 \texttt{target data} construct, resulting in undefined values in \(p\).

\begin{verbatim}
Example target_data.7.f90 (omp_4.0)

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

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

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

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

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

```cpp
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 deconstructor is called.
        #pragma omp target exit data map(delete:v[0:len])
        delete[] v;
    }
private:
    double* v;
    int len;
};
```

CHAPTER 6. DEVICES 209
The following C code allocates and frees the data member of a Matrix structure. The
\textbf{init\_matrix} function allocates the memory used in the structure and uses the
\textbf{target enter data} directive to map it to the target device. The \textbf{free\_matrix} function
removes the mapped array from the target device and then frees the memory on the host. Note, the
stand-alone \textbf{target enter data} occurs after the host memory is allocated, and the
\textbf{target exit data} construct occurs before the host data is freed.

```
#include <stdlib.h>

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

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

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

Example \textit{target\_unstructured\_data.1.c} (\texttt{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;
}
```
The following Fortran code allocates and deallocates a module array. 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)

```fortran
module example
    real(8), allocatable :: A(:)
    contains
    subroutine initialize(N)
        integer :: N
        allocate(A(N))
        !$omp target enter data map(alloc:A)
    end subroutine initialize
    subroutine finalize()
        !$omp target exit data map(delete:A)
        deallocate(A)
    end subroutine finalize
end module example
```

---

Fortran
6.12 target update Construct

6.12.1 Simple target data and target update Constructs

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

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

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

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

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

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

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

\begin{verbatim}
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++)
\end{verbatim}
subroutine vec_mult(p, v1, v2, N)
    real :: p(N), v1(N), v2(N)
    integer :: i
    call init(v1, v2, N)
    !$omp target data map(to: v1, v2) map(from: p)
    !$omp target
    !$omp parallel do
    do i=1,N
        p(i) = v1(i) * v2(i)
    end do
    !$omp end target
    call init_again(v1, v2, N)
    !$omp target update to(v1, v2)
    !$omp target
    !$omp parallel do
    do i=1,N
        p(i) = p(i) + v1(i) * v2(i)
    end do
    !$omp end target
    !$omp end target data
    call output(p, N)
end subroutine
6.12.2 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 \( \text{maybe\_init\_again()} \) returns \( true \) if new data is written.

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

Example target_update.2.c (omp_4.0)

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

C / C++

214 OpenMP Examples Version 5.2 - April 2022
Example target_update.2.f90 (omp_4.0)

```fortran
subroutine vec_mult(p, v1, v2, N)
  !interface
  logical function maybe_init_again (v1, N)
  real :: v1(N)
  integer :: N
  end function
end interface

real :: p(N), v1(N), v2(N)
integer :: i
logical :: changed

call init(v1, v2, N)
!$omp target data map(to: v1, v2) map(from: p)
  !$omp parallel do
  do i=1, N
    p(i) = v1(i) * v2(i)
  end do
  !$omp end target
changed = maybe_init_again(v1, N)
!$omp target update if(changed) to(v1(:N))
changed = maybe_init_again(v2, N)
!$omp target update if(changed) to(v2(:N))
!$omp target
  !$omp parallel do
  do i=1, N
    p(i) = p(i) + v1(i) * v2(i)
  end do
  !$omp end target
!$omp target data
  call output(p, N)
end subroutine
```
6.13 Declare Target Directive

6.13.1 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 function that can execute on the default
target device.

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

For 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.

Example declare_target.1.c (omp_5.1)

```
#pragma omp begin declare target
extern void fib(int N);
#pragma omp end declare target

#define THRESHOLD 1000000

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

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.
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.

---

Example declare_target.2.f90 (omp_4.0)

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

---

Example declare_target.1.f90 (omp_4.0)

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

module params
  integer :: THRESHOLD=1000000
end module

program my_fib
  use params
  use module_fib
  !$omp target if( N > THRESHOLD )
  call fib(N)
  !$omp end target
end program
```
6.13.2 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)

```cpp
#include <iostream>
using namespace std;

#pragma omp begin declare target // declare target--class and function
class XOR1
{
    int a;

    public:
        XOR1(int arg): a(arg) {};
        int foo();
}

int XOR1::foo() { return a^0x01;}
#pragma omp end declare target
```
#pragma omp begin declare target // declare target--class, not function
class XOR2
{
    int a;
    public:
        XOR2(int arg): a(arg) {};
        int foo();
    }
}
#pragma omp end declare target

int XOR2::foo() { return a^0x01;}

class XOR3 // declare target--neither class nor function
{
    int a;
    public:
        XOR3(int arg): a(arg) {};
        int foo();
    }
}
int XOR3::foo() { return a^0x01;}

int main(){
XOR1 my_XOR1(3);
XOR2 my_XOR2(3);
XOR3 my_XOR3(3);
int res1, res2, res3;

#pragma omp target map(tofrom:res1)
res1=my_XOR1.foo();

#pragma omp target map(tofrom:res2)
res2=my_XOR2.foo();

#pragma omp target map(tofrom:res3)
res3=my_XOR3.foo();

cout << res1 << endl; // OUT1: 2
cout << res2 << endl; // OUT2: 2
cout << res3 << endl; // OUT3: 2
}
necessary to specify in 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)

```cpp
#pragma omp begin declare target
class XOR1
{
  int a;
  public:
    XOR1(int arg): a(arg) {};
    int foo();
};
#pragma omp end declare target
```

Example declare_target.2b_functions.cpp (omp_5.1)

```cpp
#include "classes.hpp"
int XOR1::foo() { return a^0x01;}
```

Example declare_target.2b_main.cpp (omp_5.1)

```cpp
#include <iostream>
using namespace std;
#include "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 \texttt{typeY::foo()} cannot be accessed on a target device because its declaration
does not appear between \texttt{begin declare target} and \texttt{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.

\textit{Example declare\_target.2c.cpp} (omp\_5.1)

\begin{verbatim}
#pragma omp begin declare target
struct typeX
{
  int a;
};
class typeY
{
  int a;
  public:
  int foo() { return a^0x01;}
};
#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
  }
}
\end{verbatim}

\subsection{6.13.3 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 \texttt{p}, \texttt{v1}, and \texttt{v2} appear between
\texttt{begin declare target} and \texttt{end declare target} directives indicating that the variables
are mapped to the implicit device data environment of each target device. The \texttt{target update}
directive is then used to manage the consistency of the variables \texttt{p}, \texttt{v1}, and \texttt{v2} between the data
environment of the encountering host device task and the implicit device data environment of the
default target device.
The Fortran version of the above C code uses a different syntax. Fortran modules use a list syntax on the `declare target` directive to declare mapped variables.

```
Example declare_target.3.f90 (omp_4.0)
S-1 module my_arrays
S-2 !$omp declare target (N, p, v1, v2)
S-3 integer, parameter :: N=1000
S-4 real :: p(N), v1(N), v2(N)
S-5 end module
S-6 subroutine vec_mult()
S-7 use my_arrays
S-8 integer :: i
S-9 call init(v1, v2, N);
S-10 !$omp target update to(v1, v2)
S-11 !$omp target
S-12 !$omp parallel do
S-13 do i = 1,N
S-14 p(i) = v1(i) * v2(i)
S-15 end do
```
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.

The Fortran version of the above C code uses a different syntax. In Fortran modules a list syntax on the declare target directive is used to declare mapped variables and procedures. The \( N \) and \( Q \) variables are declared as a comma separated list. When the declare target directive is used to declare just the procedure, the procedure name need not be listed – it is implicitly assumed, as illustrated in the \( Pfun() \) function.
Example declare_target.4.f90 (omp_4.0)

```fortran
module my_global_array
!
integer, parameter :: N=10
real :: Q(N,N)
contains
function Pfun(i,k)
real :: Pfun
integer,intent(in) :: i,k
Pfun=(Q(i,k) * Q(k,i))
end function
end module
```

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

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

### 6.13.4 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.
Example declare_target.5.c (omp_5.1)

```c
#define N 10000
#define M 1024

#pragma omp begin declare target
float Q[N][N];
#endif

#pragma omp declare simd uniform(i) linear(k) notinbranch
float P(const int i, const int k)
{
    return Q[i][k] * Q[k][i];
}
#endif

#pragma omp end declare target

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

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

The Fortran version of the above C code uses a different syntax. Fortran modules use a list syntax of the `declare target` declaration for the mapping. Here the \( N \) and \( Q \) variables are declared in the list form as a comma separated list. The function declaration does not use a list and implicitly assumes the function name. In this Fortran example row and column indices are reversed relative to the C/C++ example, as is usual for codes optimized for memory access.
Example declare_target.5.f90 (omp_4.0)

module my_global_array
!
integer, parameter :: N=10000, M=1024
real :: Q(N,N)
contains
function P(k,i)
!
real :: P
integer,intent(in) :: k,i
P=(Q(k,i) * Q(i,k))
end function
end module

function accum() result(tmp)
use my_global_array
real :: tmp, tmp1
integer :: i
!
tmp = 0.0e0
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
!
! Note: The variable tmp is now mapped with tofrom, for correct
! execution with 4.5 (and pre-4.5) compliant compilers. See Devices Intro.
6.13.5 Declare Target Directive with link Clause

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

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

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

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

C / C++

Example declare_target.6.c (omp_5.1)

```c
#define N 100000000

float sp[N], sv1[N], sv2[N];
double dp[N], dv1[N], dv2[N];
#pragma omp declare target link(sp,sv1,sv2) link(dp,dv1,dv2)

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

#pragma omp begin declare target

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

void d_vec_mult_accum()
{
    int i;
```

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

int main()
{
    s_init(sv1, sv2, N);
    #pragma omp target map(to:sv1,sv2) map(from:sp)
    s_vec_mult_accum();
    s_output(sp, N);
    
    d_init(dv1, dv2, N);
    #pragma omp target map(to:dv1,dv2) map(from:dp)
    d_vec_mult_accum();
    d_output(dp, N);
    
    return 0;
}
```

---

**Example declare_target.6.f90 (omp_4.5)**

```fortran
module m_dat
    integer, parameter :: N=100000000
    !$omp declare target link(sp,sv1,sv2)
    real :: sp(N), sv1(N), sv2(N)
    !$omp declare target link(dp,dv1,dv2)
    double precision :: dp(N), dv1(N), dv2(N)

    contains
    subroutine s_vec_mult_accum()
        !$omp declare target
        integer :: i
        !$omp parallel do
        do i = 1,N
            sp(i) = sv1(i) * sv2(i)
        end do
    end subroutine s_vec_mult_accum
    end subroutine d_vec_mult_accum()
```

---
integer :: i

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

end subroutine

end module m_dat

program prec_vec_mult
  use m_dat

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

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

end program
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 $\text{lambda1}$. When the call operator is invoked on $\text{lambda1}$, 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 $\text{lambda2}$, 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 $\text{lambda2}$ 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 $\text{lambda3}$. The effect of the map is to make the the call for $\text{lambda3}$ 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 $\text{lambda4}$ 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 $\text{lambda4}()$ 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)

```cpp
#include <iostream>
using namespace std;

struct S { int a; int b; };

int main()
{
    // CASE 1 Lambda defined in target region
    S s = S {0,1};
```
```cpp
#pragma omp target
{
    auto lambda1 = [&s]() { s.a = s.b * 2; };
    s.b += 2;
    lambda1(); // s.a = 3 * 2
}
cout << s.a << " " << s.b << endl; //OUT 6 3

// CASE 2 Host defined lambda, Capture pointer to s
s = {0,1};
S *sp = &s;
auto lambda2 = [sp]() {sp->a = sp->b * 2; };
// closure object's sp attaches to corresponding s on target
// construct
#pragma omp target data map(sp[0])
#pragma omp target
{
    sp->b += 2;
    lambda2();
}
cout << s.a << " " << s.b << endl; //OUT 6 3

// CASE 3 Host defined lambda, Capture s by reference
s = {0,1};
auto lambda3 = [&s]() {s.a = s.b * 2; };
// closure object's s refers to corresponding s in target
// construct
#pragma omp target data map(s)
#pragma omp target
{
    s.b += 2;
    lambda3();
}
cout << s.a << " " << s.b << endl; //OUT 6 3

// CASE 4 Host defined lambda, references static variable
static S ss = {0,1};
#pragma omp declare target enter(ss)
auto lambda4 = [&]() {ss.a = ss.b * 2; };
#pragma omp target map(always,from:ss)
```
{  
  ss.b += 2;
  lambda4();
}

cout << ss.a << " " << ss.b << endl; //OUT 6 3
return 0;
6.15 teams Construct and Related Combined Constructs

6.15.1 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.

Example teams.1.c (omp_4.0)

```c
#include <stdlib.h>
#include <omp.h>
float dotprod(float B[], float C[], int N)
{
    float sum0 = 0.0;
    float sum1 = 0.0;
    #pragma omp target map(to: B[:N], C[:N]) map(tofrom: sum0, sum1)
    #pragma omp teams num_teams(2)
    {
        int i;
        if (omp_get_num_teams() != 2)
            abort();
        if (omp_get_team_num() == 0)
        {
            #pragma omp parallel for reduction(+:sum0)
            for (i=0; i<N/2; i++)
                sum0 += B[i] * C[i];
        }
        else if (omp_get_team_num() == 1)
        {
            #pragma omp parallel for reduction(+:sum1)
            for (i=N/2; i<N; i++)
                sum1 += B[i] * C[i];
        }
        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
S-1
S-2
S-3
S-4
S-5
S-6
S-7
S-8
S-9
S-10
S-11
S-12
S-13
S-14
S-15
S-16
S-17
S-18
S-19
S-20
S-21
S-22
S-23
S-24
S-25
S-26
S-27

C / C++

Fortran

Example teams.1.f90 (omp_4.0)

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

use omp_lib, ONLY : omp_get_num_teams, omp_get_team_num

real :: B(N), C(N), sum,sum0, sum1
integer :: N, i
sum0 = 0.0e0
sum1 = 0.0e0

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

Fortran

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.
6.15.2 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)

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

CHAPTER 6. DEVICES 235
function dotprod(B,C,N, block_size, num_teams, block_threads) result(sum)

 implicit none

 real :: B(N), C(N), sum
 integer :: N, block_size, num_teams, block_threads, i, i0

 sum = 0.0e0

 !$omp target map(to: B, C) map(tofrom: sum)
 !$omp teams num_teams(num_teams) thread_limit(block_threads) &
 !$omp& reduction(+:sum)
 !$omp distribute
 do i0=1,N, block_size
 !$omp parallel do reduction(+:sum)
 do i = i0, min(i0+block_size,N)
 sum = sum + B(i) * C(i)
 end do
 end do
 !$omp end teams
 !$omp end target

 end function

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

6.15.3 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.
Example teams.3.c (omp_4.5)

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

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

Example teams.3.f90 (omp_4.5)

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

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

The following example shows how the `target teams` and distribute parallel loop constructs are used to execute a `target` region. The `teams` construct creates a league of at most eight teams where the 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.

---

```
Example teams.4.c (omp_4.0)
```

```c
#define N 1024*1024
float dotprod(float B[], float C[])
{
    float sum = 0.0;
    int i;
    #pragma omp target map(to: B[0:N], C[0:N]) map(tofrom: sum)
    #pragma omp teams num_teams(8) thread_limit(16) reduction(+:sum)
    #pragma omp distribute parallel for reduction(+:sum) \
    dist_schedule(static, 1024) schedule(static, 64)
    for (i=0; i<N; i++)
    {
        sum += B[i] * C[i];
    }
    return sum;
}

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

---
Example teams.4.f90 (omp_4.0)

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

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

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

6.15.5 target teams and distribute simd

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.
Example teams.5.c (omp_4.0)

```c
extern void init(float *, float *, int);
extern void output(float *, int);

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

Example teams.5.f90 (omp_4.0)

```fortran
subroutine vec_mult(p, v1, v2, N)
    real :: p(N), v1(N), v2(N)
    integer :: i
    call init(v1, v2, N)
    !$omp target teams map(to: v1, v2) map(from: p)
    !$omp distribute simd
    do i=1,N
        p(i) = v1(i) * v2(i)
    end do
    !$omp end target teams
    call output(p, N)
end subroutine
```
6.15.6 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.

---

**Example teams.6.c (omp_4.0)**

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

---

**Example teams.6.f90 (omp_4.0)**

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

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

6.16.1 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.

```
#include <omp.h>

#pragma omp declare target float F(float);

#define N 1000000000
#define CHUNKSZ 1000000

void init(float *, int);

float Z[N];

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

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).

```
Example async_target.1.f90 (omp_4.0)
```

```fortran
module parameters
integer, parameter :: N=1000000000, CHUNKSZ=1000000
end module

subroutine pipedF()
use parameters, ONLY: N, CHUNKSZ
integer :: C, i
real :: z(N)

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

call init(z,N)
do C=1,N,CHUNKSZ

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

end subroutine pipedF
```

The following example shows how the `task` and `target` constructs are used to execute multiple `target` regions asynchronously. The task dependence ensures that the storage is allocated and initialized on the device before it is accessed.
Example async_target.2.c (omp_5.1)

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

#pragma omp begin declare target
extern void init(float *, float *, int);
#pragma omp end declare target

extern void foo();
extern void output(float *, int);

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

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 \( v_1 \) and \( v_2 \) arrays will be in a non-associated state on the device. When space for \( v_1 \) and \( v_2 \) 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 \( v_1 \) and \( v_2 \) 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 \( v_1 \) and \( v_2 \) 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 \( v_1 \) and \( v_2 \) in the two tasks. It would be noncompliant to use \( v_1 \) and/or \( v_2 \) 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 \( v_1 \) and \( v_2 \) 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)**

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

  !$omp target data map(v1,v2)

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

  call foo() ! execute other work asychronously

  !$omp task shared(v1,v2,p) depend(in: N)
  !$omp target device(idev) map(from: p)
```
The following example shows how to execute code asynchronously on a device without an explicit task. The `nowait` clause on a `target` construct allows the thread of the `target task` to perform other work while waiting for the `target` region execution to complete. Hence, the `target` region can execute asynchronously on the device (without requiring a host thread to idle while waiting for the `target task` execution to complete).

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

After a team of threads is formed the 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.
Example async_target.3.c (omp_5.1)

```c
#include <stdio.h>

#define N 1000000 //N must be even

void init(int n, float *v1, float *v2);

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

    init(n, v1, v2);

    #pragma omp parallel
    {
        #pragma omp masked
        #pragma omp target teams distribute parallel for nowait

        for(i=0; i<n/2; i++){ vxv[i] = v1[i]*v2[i]; }
        for(i=n/2; i<n; i++){ vxv[i] = v1[i]*v2[i]; }
    }

    printf(" vxv[0] vxv[n-1] %f %f\n", vxv[0], vxv[n-1]);
    return 0;
}
```

Example async_target.3.f90 (omp_5.1)

```fortran
program concurrent_async
    use omp_lib

    integer,parameter :: n=1000000 !!n must be even
    integer :: i, chunk=1000
    real :: v1(n), v2(n), vxv(n)

    call init(n, v1, v2)

    !$omp parallel
```
6.16.3 Asynchronous target with nowait and depend Clauses

More details on dependences can be found in Section 5.3 on page 105, Task Dependences. In this example, there are three flow dependences. In the first two dependences the target task does not execute until the preceding explicit tasks have finished. These dependences are produced by arrays $v1$ and $v2$ with the out dependence type in the first two tasks, and the in dependence type in the target task.

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

The nowait clause on the target construct creates a deferrable target task, allowing the encountering task to continue execution without waiting for the completion of the target task.
Example async_target.4.f90 (omp_4.5)

```c
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)
  !$omp parallel for private(i)
  for (i=0; i<N; i++)
    p[i] = v1[i] * v2[i];
  !$omp task depend(in:p)
  output(p, N);
```

CHAPTER 6. DEVICES

249
S-22       !$omp parallel do
S-23       do i=1,N
S-24       p(i) = v1(i) * v2(i)
S-25       end do
S-26       !$omp end target
S-27
S-28
S-29       !$omp task depend(in:p)
S-30       call output(p, N)
S-31       !$omp end task
S-32
S-33       !$omp end single
S-34       !$omp end parallel
S-35
S-36       deallocate(p, v1, v2 )
S-37
S-38       end subroutine
6.17 Device Routines

6.17.1 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.

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

#pragma omp begin declare target
void vec_mult(float *p, float *v1, float *v2, int N);
extern float *p, *v1, *v2;
extern int N;
#pragma omp end declare target

extern void init_vars(float *, float *, int);
extern void output(float *, int);

void foo()
{
    init_vars(v1, v2, N);
    #pragma omp target device(42) map(p[:N], v1[:N], v2[:N])
    {
        vec_mult(p, v1, v2, N);
    }
    output(p, N);
}

void vec_mult(float *p, float *v1, float *v2, int N)
{
    int i;
    int nthreads;
    if (!omp_is_initial_device())
    {
        printf("1024 threads on target device\n");
        nthreads = 1024;
    }
    else
    {
        printf("8 threads on initial device\n");
        nthreads = 8;
    }
```
#pragma omp parallel for private(i) num_threads(nthreads)
for (i=0; i<N; i++)
p[i] = v1[i] * v2[i];
}

Example device.1.f90 (omp_4.0)

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

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

program prog_vec_mult
  use params
  use vmult
  real :: p(N), v1(N), v2(N)
  call init(v1,v2,N)
  !$omp target device(42) map(p, v1, v2)
  call vec_mult(p, v1, v2, N)
  !$omp end target
  call output(p, N)
end program
6.17.2 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
#include <omp.h>
extern void init(float *, float *, int);
extern void output(float *, int);
void vec_mult(float *p, float *v1, float *v2, int N)
{
    int i;
    init(v1, v2, N);
    int ndev = omp_get_num_devices();
    int do_offload = (ndev>0 && N>1000000);
    #pragma omp target if(do_offload) \
       map(to: v1[0:N], v2[:N]) \
       map(from: p[0:N])
    #pragma omp parallel for if(N>1000) private(i)
    for (i=0; i<N; i++)
        p[i] = v1[i] * v2[i];
    output(p, N);
}
```

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

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

---

### Example device.3.c (omp_4.0)

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

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

---

### Example device.3.f90 (omp_4.0)

```fortran
program foo
    use omp_lib, ONLY : omp_get_default_device, omp_set_default_device
    integer :: old_default_device, new_default_device
    old_default_device = omp_get_default_device()
    print*, "Default device = ", old_default_device
    new_default_device = old_default_device + 1
    call omp_set_default_device(new_default_device)
    if (omp_get_default_device() == old_default_device)
        print*, "Default device is STILL = ", old_default_device
end program
```
6.17.4 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.

---

Example `target_associate_ptr.1.c` (omp_4.5)

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

#define CS 50
#define N (CS*2)

int main() {
    int arr[N];
    int *dev_ptr;
    int dev;

    for (int i = 0; i < N; i++)
        arr[i] = i;

    dev = omp_get_default_device();

    // Allocate device memory
    dev_ptr = (int *)omp_target_alloc(sizeof(int) * CS, dev);

    // Loop over chunks
    for (int ioff = 0; ioff < N; ioff += CS) {
        // Associate device memory with one chunk of host memory
        omp_target_associate_ptr(&arr[ioff], dev_ptr,
```

```c
```
sizeof(int) * CS, 0, dev);

printf("before: arr[%d]=%d\n", ioff, arr[ioff]);

// Update the device data
#pragma omp target update to(arr[ioff:CS]) device(dev)

// Explicit mapping of arr to make sure that we use the allocated
// and associated memory. No host-device data update here.
#pragma omp target map(tofrom : arr[ioff:CS]) device(dev)
for (int i = 0; i < CS; i++) {
    arr[i+ioff]++;
}

// Update the host data
#pragma omp target update from(arr[ioff:CS]) device(dev)

printf("after: arr[%d]=%d\n", ioff, arr[ioff]);

// Disassociate device pointer from the current chunk of host memory
// before next use
omp_target_disassociate_ptr(&arr[ioff], dev);

// Free device memory
omp_target_free(dev_ptr, dev);

return 0;

/* Outputs:
before: arr[0]=0
after: arr[0]=1
before: arr[50]=50
after: arr[50]=51
*/

Example target_associate_ptr.f90 (omp_5.1)
integer, target :: arr(N)
type(c_ptr) :: h_ptr, dev_ptr
integer(c_size_t) :: csize, dev_off
integer(c_int) :: dev
integer :: i, ioff, s

doi = 1, N
   arr(i) = i
end do

dev = omp_get_default_device()
csize = c_sizeof(arr(1)) * CS

! Allocate device memory
dev_ptr = omp_target_alloc(csize, dev)
dev_off = 0

! Loop over chunks
do ioff = 1, N, CS
   ! 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

CHAPTER 6. DEVICES
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.
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);

omp_target_free(mem_dev_cpy, t);

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
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

```fortran
program device_mem
  use omp_lib
  use, intrinsic :: iso_c_binding

  integer(kind=4),parameter :: N = 10
  type(c_ptr) :: cp
  integer(c_int), pointer :: fp(:)
  integer(c_int) :: rc, host_dev, targ_dev
  integer(c_size_t) :: int_bytes

  integer, pointer :: fp_src(:), fp_dst(:) ! Section 2 vars
  type(c_ptr) :: cp_src, cp_dst ! Section 2 vars
  integer, pointer :: h_fp(:) ! Section 3 vars
  type(c_ptr) :: h_cp, d_cp ! Section 3 vars
```

---

OpenMP Examples Version 5.2 - April 2022
integer :: i

host_dev = omp_get_initial_device()
targ_dev = omp_get_default_device()
int_bytes = C_SIZEOF(rc)

!-------------------------------------------------------------Section 1 vv--------

cp = omp_target_alloc(N*int_bytes, targ_dev)

!$omp target is_device_ptr(cp) device(targ_dev) !fp implicit map
  call c_f_pointer(cp, fp, [ N ])
  fp(:) = 4
  if( all(fp == 4) ) print*,"PASSED 1 of 5"
  nullify(fp)
!$omp end target

call omp_target_free(cp, targ_dev)
cp = c_null_ptr

!-------------------------------------------------------------Section 2 vv--------


cp_src = omp_target_alloc((N+1)*int_bytes, host_dev)
cp_dst = omp_target_alloc( N *int_bytes, targ_dev)

! Initialize host array (src)
call c_f_pointer(cp_src, fp_src, [N+1])
fp_src = [(i,i=1,N+1)]

!$omp target device(targ_dev) is_device_ptr(cp_dst)
call c_f_pointer(cp_dst, fp_dst, [N]) ! fp_dst becomes associated
fp_dst(:) = -1 ! Initial device storage
nullify(fp_dst) ! return as disassociated
!$omp end target

! Copy subset of host (src) array to device (dst) array
rc = omp_target_memcpy(
  dst=cp_dst, src=cp_src, length=N*int_bytes, &
  dst_offset=0, src_offset=int_bytes, &
  dst_device_num=targ_dev, src_device_num=host_dev)

! Check dst array on device
!$omp target device(targ_dev) is_device_ptr(cp_dst)
call c_f_pointer(cp_dst, fp_dst, [N])
if ( all(fp_dst == [(i,i=1,N)]) ) print*,"PASSED 2 of 5"
nullify(fp_dst)
!$omp end target
allocate host memory and initialize.
allocate(h_fp(N), source=[(i,i=1,N)])

h_cp = c_loc(h_fp)
  ! Device is not aware of allocation on host
  if(omp_target_is_present(h_cp, targ_dev) == 0) &
    print*, "PASSED 3 of 5"

  ! Allocate device memory
  d_cp = omp_target_alloc(c_sizeof(h_fp(1))*size(h_fp), targ_dev)

  ! now associate host and device storage
  rc=omp_target_associate_ptr(h_cp,d_cp,c_sizeof(h_fp(1))*size(h_fp), &
    0_c_size_t,targ_dev)

  ! check presence of device data, associated w. host pointer
  if(omp_target_is_present(h_cp, targ_dev) /= 0) &
    print*,"PASSED 4 of 5"

  ! copy from host to device via C pointers
  rc=omp_target_memcpy(d_cp, h_cp,c_sizeof(h_fp(1))*size(h_fp), &
    0_c_size_t, 0_c_size_t, &
    targ_dev, host_dev)

  ! validate the device data in the target region
  ! no data copy here since the reference count is infinity
  !$omp target device(targ_dev) map(h_fp)
  if ( all(h_fp == [(i,i=1,N)]) ) print*, "PASSED 5 of 5"
  !$omp end target

  call omp_target_free(d_cp,targ_dev)
deallocate(h_fp)
end program
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 dependency (or with dependency preserved using ordered simd) are candidates for vectorization by the compiler for execution with SIMD units. In addition, with state-of-the-art vectorization technology and declare simd 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.
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)

```c
#include <stdio.h>

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

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

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

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

int main(){
    int i;
    const int N=32;
    ```
```c
void work(double a[N], double b[N], int N)
{
    for ( int i=0; i<N; i++ ) {
        a[i] = i; b[i] = N-i;
    }
}

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

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

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

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

function add2(a,b,i, fact) result(c)
    implicit none
    !omp declare simd(add2) uniform(a,b,fact) linear(i:1)
    integer :: i
    double precision :: a(*),b(*),fact, c
    c = a(i) + b(i) + fact
end function
```
A thread that encounters a SIMD construct executes a vectorized code of the iterations. Similar to the concerns of a worksharing loop, a loop vectorized with a SIMD construct must assure that temporary and reduction variables are privatized and declared as reductions with clauses. The example below illustrates the use of `private` and `reduction` clauses in a SIMD construct.

### Example SIMD.3.c (omp_4.0)

```c
double work( double *a, double *b, int n )
{
    int i;
    double tmp, sum;
    sum = 0.0;
    #pragma omp simd private(tmp) reduction(+:sum)
    for (i = 0; i < n; i++) {
        tmp = a[i] + b[i];
        sum += tmp;
    }
    return sum;
}
```
Example SIMD.3.f90 (omp_4.0)

```fortran
subroutine work( a, b, n, sum )
imPLICIT none
integer :: i, n
double precision :: a(n), b(n), sum, tmp
sum = 0.0d0
!$omp simd private(tmp) reduction(+:sum)
do i = 1,n
  tmp = a(i) + b(i)
sum = sum + tmp
end do
end subroutine work
```

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

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

Example SIMD.4.c (omp_4.0)

```c
void work( float *b, int n, int m )
{
  int i;
  #pragma omp simd safelen(16)
  for (i = m; i < n; i++)
    b[i] = b[i-m] - 1.0f;
} 
```
Example SIMD.4.f90 (omp_4.0)

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

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

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

Example SIMD.5.c (omp_4.0)

```c
void work( double **a, double **b, double **c, int n )
{
    int i, j;
    double tmp;

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

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

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

---

**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)*

```c
#include <stdio.h>

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

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

float goo(float *p){
  *p = *p + 18.5f;
  return *p;
}

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

---

Example SIMD.6.f90 (omp_4.0)

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

function myaddint(a, b, n) result(r)
  implicit none
  integer :: a(*), b(*), n, r
  integer :: i
  integer, external :: foo
  !$omp simd
  do i=1, n
    a(i) = foo(b(i)) ! foo is not called under a condition
  end do
  r = a(n)
end function myaddint

function goo(p) result(r)
  implicit none
  !$omp declare simd(goo) inbranch
  real :: p, r

Fortran

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

```fortran
function myaddfloat(x, y, n) result(r)
    implicit none
    real :: x(*), y(*), r
    integer :: n
    integer :: i
    real, external :: goo
    !$omp simd
    do i=1, n
        if (x(i) > y(i)) then
            x(i) = goo(y(i))
        ! goo is called under the condition (or within a branch)
        else
            x(i) = y(i)
        endif
    end do
    r = x(n)
end function myaddfloat
```

C / C++

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

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

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

Example \texttt{SIMD.7.c (omp\_4.0)}
int main(void)
{
    int i;

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

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

    printf("Done a[%d] = %d\n", N-1, a[N-1]);

    return 0;
}

Example SIMD.7.f90 (omp_4.0)

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

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

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

    write(*,*) "Done a(" , N-1 , ") = ", a(N-1)

end program

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

if (n <= 1) then
  r = n
else
  r = fib(n-1) + fib(n-2)
endif
end function fib
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.

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

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

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

int main(void)
{
    float pri, arr[1000];
    int i;
    for (i = 0; i < 1000; ++i) {
```
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       }

Example SIMD.8.f90 (omp_4.0)

module work
S-3       integer :: P(1000)
S-4       real :: A(1000)
S-5       contains
S-7       function do_work(arr) result(pri)
S-8           implicit none
S-9           real, dimension(*) :: arr
S-10       real :: pri
S-12       integer :: i, j
S-13       !$omp simd private(j) lastprivate(pri)
S-15       do i = 1, 999
S-16           j = P(i)
S-18           pri = 0.5
S-19           if (mod(j-1, 2) == 0) then
S-20               pri = A(j+1) + arr(i)
S-21           endif
S-22           A(j) = pri * 1.5
S-23           pri = pri + A(j)
S-24       end do
S-26       end function do_work
S-27       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

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.
Example linear_modifier.1.cpp (omp_5.2)

```
#include <stdio.h>

#define NN 1023

int a[NN];

#pragma omp declare simd linear(p: ref) simdlen(8)
void add_one2(int& p)
{
    p += 1;
}

int main(void)
{
    int i;
    int* p = a;

    for (i = 0; i < NN; i++) {
        a[i] = i;
    }

    #pragma omp simd linear(p) simdlen(8)
    for (i = 0; i < NN; i++) {
        int& k = *p;
        add_one2(k);
        add_one2(a[i]);
        p++;
    }

    for (i = 0; i < NN; i++) {
        if (a[i] != i+2) {
            printf("failed\n");
            return 1;
        }
    }

    printf("passed\n");
    return 0;
}
```
Example linear_modifier.1.f90 (omp_5.2)

module m
  integer, parameter :: NN = 1023
  integer :: a(NN)
contains
  subroutine add_one2(p)
    !$omp declare simd(add_one2) linear(p: ref) simdlen(8)
    implicit none
    integer :: p
    p = p + 1
    end subroutine
  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
  print *, "passed"
end program
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.

```
#include <stdio.h>

#define NN 1023
int a[NN];

#pragma omp declare simd linear(p: ref) linear(i: uval)
void add_one2(int& p, const int& i)
{
    p += i;
}

int main(void)
{
    int i;
    int* p = a;

    for (i = 0; i < NN; i++) {
        a[i] = i;
    }

    #pragma omp simd linear(p)
    for (i = 0; i < NN; i++) {
        int& k = *p;
        add_one2(k, i);
        p++;
    }

    for (i = 0; i < NN; i++) {
```
if (a[i] != i*2) {
    printf("failed\n");
    return 1;
}

printf("passed\n");
return 0;

Example linear_modifier.f90 (omp_5.2)

module m
    integer, parameter :: NN = 1023
    integer :: a(NN)
contains
    subroutine add_one2(p, i)
        !$omp declare simd(add_one2) linear(p: ref) linear(i: uval)
        implicit none
        integer :: p
        integer, intent(in) :: i
        p = p + i
    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)
    do i = 1, NN
        call add_one2(a(p), i)
        p = p + 1
    end do
    do i = 1, NN
        if (a(i) /= i*2) then
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 \texttt{val} modifier is used for the \texttt{linear} clause in the \texttt{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 \texttt{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.

\begin{verbatim}
#include <stdio.h>
#define N 128
#pragma omp declare simd simdlen(4) uniform(x, y) linear(i:val,step(1))
double func(double x[], double y[], int i)
{
  return (x[i] + y[i]);
}
int main(void)
{
  double x[N], y[N], z1[N], z2;
  int i, k;
  for (i = 0; i < N; i++) {
    x[i] = (double)i;
    y[i] = (double)i*2;
  }
  k = 0;
  #pragma omp simd linear(k)
  for (i = 0; i < N; i++) {
    z1[i] = func(x[i], y[i], i);
  }
}
\end{verbatim}

Example linear_modifier.3.c (omp_5.2)
S-24             z1[i] = func(x, y, k);
S-25     k++;
S-26 }           
S-27 
S-28 for (i = 0; i < N; i++) { 
S-29     z2 = (double)(i + i*2);
S-30 if (z1[i] != z2) {
S-31         printf("failed\n");
S-32         return 1;
S-33     }
S-34 }           
S-35 printf("passed\n");
S-36 return 0;
S-37 }           

Example linear_modifier.3.f90 (omp_5.2)

S-1 module func_mod
S-2 contains
S-3     real(8) function func(x, y, i)
S-4 !$omp declare simd(func) simdlen(4) uniform(x, y) linear(i:val,step(1))
S-5     implicit none
S-6     real(8), intent(in) :: x(*), y(*)
S-7     integer, intent(in) :: i
S-8 
S-9     func = x(i) + y(i)
S-10 
S-11 end function func
S-12 end module func_mod
S-13 
S-14 program main
S-15 use func_mod
S-16 implicit none
S-17 integer, parameter :: n = 128
S-18 real(8) :: x(n), y(n), z1(n), z2
S-19 integer :: i, k
S-20 
S-21 do i=1, n
S-22     x(i) = real(i, kind=8)
S-23     y(i) = real(i*2, kind=8)
S-24 enddo
S-25 
S-26 k = 1
S-27 !$omp simd linear(k)
S-28 do i=1, n
z1(i) = func(x, y, k)
k = k + 1

S-32
S-33
S-34
S-35
S-36
S-37
S-38
S-39
S-40
S-41

do i=1, n
  z2 = real(i+i*2, kind=8)
  if (z1(i) /= z2) then
    print *, 'failed'
    stop
  endif
endo

print *, 'passed'
end program main
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.l.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  }
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 \texttt{sizes(5,16)} clause of the \texttt{tile} construct specifies a 5x16 blocking, applied to the outer (column) and inner (row) loops. The worksharing-loop construct before the \texttt{tile} construct is applied after the transform.

\texttt{Example tile.1.f90 (omp_5.1)}

```fortran
subroutine func1(A)
    integer :: A(128,100)
    integer :: i, j
    !$omp parallel do
    !$omp tile sizes(5,16)
    do i = 1, 100
        do j = 1, 128
            A(j,i) = j*1000 + i
        end do; end do
    end do; end do
end subroutine
```

```fortran
subroutine func2(A)
    integer :: A(128,100)
    integer :: i1, j1, i2, j2
    !$omp parallel do
    do i1 = 1, 100, 5
        do j1 = 1, 128, 16
            do i2 = i1, i1+(5-1)
                do j2 = j1, j1+(16-1)
                    A(j2,i2) = j2*1000 + i2
                end do; end do
            end do; end do
        end do; end do
    end do; end do
end subroutine
```
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`.

---

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

```c
void func1(int A[100][128])
{
    #pragma omp tile sizes(4, 4)
    #pragma omp tile sizes(5,16)
    for (int i = 0; i < 100; ++i)
        for (int j = 0; j < 128; ++j)
            A[i][j] = i*1000 + j;
}
```

```c
void func2(int A[100][128])
{
    #pragma omp tile sizes(4,4)
    for (int i1 = 0; i1 < 100; i1+=5)
        for (int j1 = 0; j1 < 128; j1+=16)
            for (int i2 = i1; i2 < i1+5; ++i2)
                for (int j2 = j1; j2 < j1+16; ++j2)
                    A[i2][j2] = i2*1000 + j2;
}
```

```c
void func3(int A[100][128])
{
    for (int i11 = 0; i11 < 100; i11+= 5*4)
        for (int j11 = 0; j11 < 128; j11+=16*4)
            for (int i12 = i11; i12 < i11+( 5*4); i12+= 5)
                for (int j12 = j11; j12 < j11+(16*4); j12+=16)
                    for (int i2 = i12; i2 < i12+ 5; ++i2)
                        for (int j2 = j12; j2 < j12+16; ++j2)
                            A[i2][j2] = i2*1000 + j2;
}
```

---

CHAPTER 8. LOOP TRANSFORMATIONS 287
Example tile.2.f90 (omp_5.1)

subroutine func1(A)
  integer :: A(128,100)
  integer :: i, j
  !$omp tile sizes(4, 4)
  !$omp tile sizes(5,16)
  do i = 1, 100
    do j = 1, 128
      A(j,i) = j*1000 + i
    end do; end do
end subroutine

subroutine func2(A)
  integer :: A(128,100)
  integer :: i1, j1, i2, j2
  !$omp tile sizes(4,4)
  do i1 = 1, 100,5
    do j1 = 1, 128,16
      do i2 = i1, i1+(5-1)
        do j2 = j1, j1+(16-1)
          A(j2,i2) = j2*1000 + i2
        end do; end do
      end do; end do
    end do; end do
end subroutine

subroutine func3(A)
  integer :: A(128,100)
  integer :: i11, j11, i12, j12, i2, j2
  do i11 = 1, 100, 5*4
    do j11 = 1, 128, 16*4
      do i12 = i11, i11+(5*4-1), 5
        do j12 = j11, j11+(16*4-1), 16
          do i2 = i12, i12+ 5-1
            do j2 = j12, j12+16-1
              A(j2,i2) = j2*1000 + i2
            end do; end do
          end do; end do
        end do; end do
      end do; end do
    end do; end do
  enddo; enddo;
end subroutine
8.2 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 a 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.

```
void unroll(double A[], int n)
{
    #pragma omp unroll
    for (int i = 0; i < n; ++i)
        A[i] = 0;
}

void unroll_full(double A[])
{
    #pragma omp unroll full
    for (int i = 0; i < 4; ++i)
        A[i] = 0;
}

void unroll_full_equivalent(double A[])
{
    A[0] = 0;
    A[1] = 0;
    A[2] = 0;
    A[3] = 0;
}
```
Example unroll.1.f90 (omp_5.1)

```fortran
subroutine unroll(A, n)
  implicit none
  integer :: i, n
  double precision :: A(n)
  !$omp unroll
  do i = 1, n
    A(i) = 0.0d0
  end do
end subroutine

subroutine unroll_full(A)
  implicit none
  integer :: i
  double precision :: A(*)
  !$omp unroll full
  do i = 1, 4
    A(i) = 0.0d0
  end do
end subroutine

subroutine unroll_full_equivalent(A)
  implicit none
  double precision :: A(*)
  A(1) = 0.0d0
  A(2) = 0.0d0
  A(3) = 0.0d0
  A(4) = 0.0d0
end subroutine
```

The next example shows cases when it is incorrect to use full unrolling.
Example unroll.2.c (omp_5.1)

```c
void illegal_2a(double A[])
{
    #pragma omp for
    #pragma omp unroll full // ERROR: No loop left after full unrolling.
    for (int i = 0; i < 12; ++i)
        A[i] = 0;
}

void illegal_2b(double A[])
{
    // Loop might be fully unrolled (or a partially unrolled loop replacement). Hence, no canonical for-loop, resulting in non-compliant code. Implementations may suggest adding a "partial" clause.

    #pragma omp for
    #pragma omp unroll // ERROR: may result in non-compliant code
    for (int i = 0; i < 12; ++i)
        A[i] = 0;
}

void illegal_2c(int n, double A[])
{
    #pragma omp unroll full // ERROR: Constant iteration count required.
    for (int i = 0; i < n; ++i)
        A[i] = 0;
}
```

Example unroll.2.f90 (omp_5.1)

```fortran
subroutine illegal_2a(A)
    implicit none
    double precision :: A(*)
    integer :: i

    !$omp do
    !$omp unroll full !! ERROR: No loop left after full unrolling
    do i = 1,12
        A(i) = 0.0d0
    end do
end subroutine
```
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.

Example unroll.3.c (omp_5.1)

```c
void unroll3_partial(double A[])
{
  #pragma omp unroll partial(4)
  for (int i = 0; i < 128; ++i)
    A[i] = 0;
}
```
void unroll3_partial_equivalent(double A[]) {
    for (int i_iv = 0; i_iv < 32; ++i_iv) {
        A[i_iv * 4 + 0] = 0;
        A[i_iv * 4 + 1] = 0;
        A[i_iv * 4 + 2] = 0;
        A[i_iv * 4 + 3] = 0;
    }
}

void unroll3_partial_nofactor(double A[]) {
    #pragma omp unroll partial
    for (int i = 0; i < 128; ++i)
        A[i] = 0;
}

Example unroll.3.f90 (omp_5.1)

subroutine unroll3_partial(A)
    implicit none
    double precision :: A(*)
    integer :: i
!
    !$omp unroll partial(4)
    do i = 1, 128
        A(i) = 0
    end do
end subroutine

subroutine unroll3_partial_equivalent(A)
    implicit none
    double precision :: A(*)
    integer :: i_iv
    do i_iv = 0, 31
        A(i_iv * 4 + 1) = 0
        A(i_iv * 4 + 2) = 0
        A(i_iv * 4 + 3) = 0
        A(i_iv * 4 + 4) = 0
    end do
end subroutine

subroutine unroll3_partial_nofactor(A)
    implicit none
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.

Example unroll.4.c (omp_5.1)

```c
void unroll_partial_remainder(int n, int A[])
{
    #pragma omp parallel for
    #pragma omp unroll partial(4)
    for (int i = 0; i < n; ++i)
        A[i] = i;
}

void unroll_partial_remainder_option1(int n, int A[])
{
    #pragma omp parallel for
    for (int i_iv = 0; i_iv < (n+3)/4; ++i_iv) {
        A[i_iv * 4 + 0] = i_iv * 4 + 0;
        if (i_iv * 4 + 1 < n) A[i_iv * 4 + 1] = i_iv * 4 + 1;
        if (i_iv * 4 + 2 < n) A[i_iv * 4 + 2] = i_iv * 4 + 2;
        if (i_iv * 4 + 3 < n) A[i_iv * 4 + 3] = i_iv * 4 + 3;
    }
}
```
void unroll_partial_remainder_option2(int n, int A[]) {
    #pragma omp parallel for
    for (int i_iv = 0; i_iv < (n+3)/4; ++i_iv) {
        if (i_iv < n/4) {
            A[i_iv * 4 + 0] = i_iv * 4 + 0;
            A[i_iv * 4 + 1] = i_iv * 4 + 1;
            A[i_iv * 4 + 2] = i_iv * 4 + 2;
            A[i_iv * 4 + 3] = i_iv * 4 + 3;
        } else { // remainder loop
            for (int i_rem = i_iv*4; i_rem < n; ++i_rem)
                A[i_rem] = i_rem;
        }
    }
}

void unroll_partial_remainder_option3(int n, int A[]) {
    // main loop
    #pragma omp parallel for
    for (int i_iv = 0; i_iv < n/4; ++i_iv) {
        A[i_iv * 4 + 0] = i_iv * 4 + 0;
        A[i_iv * 4 + 1] = i_iv * 4 + 1;
        A[i_iv * 4 + 2] = i_iv * 4 + 2;
        A[i_iv * 4 + 3] = i_iv * 4 + 3;
    }
    // remainder loop
    #pragma omp parallel for
    for (int i_rem = (n/4)*4; i_rem < n; ++i_rem)
        A[i_rem] = i_rem;
}

#include <stdio.h>
define NT 12

int main(){
    int error=0, A[NT],C[NT];
    for(int i = 0; i<NT; i++) { A[i]=0; C[i]=i; }
    for(int i = 0; i<NT; i++) A[i]=0.0;
    unroll_partial_remainder(NT,A);
    for(int i = 0; i<NT; i++) if(A[i] != C[i]) error=1;
for(int i = 0; i<NT; i++) A[i]=0.0;
unroll_partial_remainder_option1(NT,A);
for(int i = 0; i<NT; i++) if(A[i] != C[i]) error=1;
for(int i = 0; i<NT; i++) A[i]=0.0;
unroll_partial_remainder_option2(NT,A);
for(int i = 0; i<NT; i++) if(A[i] != C[i]) error=1;
for(int i = 0; i<NT; i++) A[i]=0.0;
unroll_partial_remainder_option3(NT,A);
for(int i = 0; i<NT; i++) if(A[i] != C[i]) error=1;
if(!error) printf("OUT: Passed\n");
if( error) printf("OUT: Failed\n");
}

Example unroll.4.f90 (omp_5.1)

subroutine unroll_partial_remainder(n, A)
  implicit none
  integer :: n, i
  integer :: A(*)
  !$omp parallel do
  !$omp unroll partial(4)
  do i = 1, n
    A(i) = i
  end do
end subroutine

subroutine unroll_partial_remainder_option1(n, A)
  implicit none
  integer :: n, i_iv
  integer :: A(*)
  !$omp parallel do
  do i_iv = 0,(n+3)/4 -1
    A(i_iv * 4 + 1) = i_iv * 4 + 1
    if (i_iv * 4 + 2 <= n) A(i_iv * 4 + 2) = i_iv * 4 + 2
    if (i_iv * 4 + 3 <= n) A(i_iv * 4 + 3) = i_iv * 4 + 3
    if (i_iv * 4 + 4 <= n) A(i_iv * 4 + 4) = i_iv * 4 + 4
  end do
end subroutine

subroutine unroll_partial_remainder_option2(n, A)
  implicit none
  integer :: n, i_iv, i_rem
  integer :: A(*)

  !$omp parallel do
  do i_iv = 0, (n+3)/4 -1
    if (i_iv < n/4) then
      A(i_iv * 4 + 1) = i_iv * 4 + 1
      A(i_iv * 4 + 2) = i_iv * 4 + 2
      A(i_iv * 4 + 3) = i_iv * 4 + 3
      A(i_iv * 4 + 4) = i_iv * 4 + 4
    else
      !! remainder loop
      do i_rem = i_iv*4 +1, n
        A(i_rem) = i_rem
      end do
    end if
  end do
  !$omp parallel do
end subroutine

subroutine unroll_partial_remainder_option3(n, A)
  implicit none
  integer :: n, i_iv, i_rem
  integer :: A(*)

  !$omp parallel do
  do i_iv = 0, (n/4) -1
    A(i_iv * 4 + 1) = i_iv * 4 + 1
    A(i_iv * 4 + 2) = i_iv * 4 + 2
    A(i_iv * 4 + 3) = i_iv * 4 + 3
    A(i_iv * 4 + 4) = i_iv * 4 + 4
  end do
  !! remainder loop
  !$omp parallel do
  do i_rem = (n/4)*4 +1, n
    A(i_rem) = i_rem
  end do
end subroutine

program main
implicit none

integer, parameter :: NT=12

integer :: i

logical :: error=.false.

integer :: A(NT), C(NT)=[ (i, i=1,NT) ]

A(1:NT)=0
call unroll_partial_remainder(NT, A)
if( .not. all(A(1:NT) == C(1:NT)) ) error = .true.

A(1:NT)=0
call unroll_partial_remainder_option1(NT, A)
if( .not. all(A(1:NT) == C(1:NT)) ) error = .true.

A(1:NT)=0
call unroll_partial_remainder_option2(NT, A)
if( .not. all(A(1:NT) == C(1:NT)) ) error = .true.

A(1:NT)=0
call unroll_partial_remainder_option3(NT, A)
if( .not. all(A(1:NT) == C(1:NT)) ) error = .true.

if(.not. error) print*, "OUT: Passed."
if( error) print*, "OUT: Failed"
end program

-------------------- Fortran ----------------------
8.3 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.

\[
\begin{array}{ccc}
0 & 1 & 2 \\
3 & 4 & 5 \\
6 & 7 & 8 \\
9 & 10 & 11
\end{array}
\]

(A) 2-dimensional tiling with partial tiles

\[
\begin{array}{c}
0 & 16 & 32 & 48 & 64 & 80 & 96 & 100 \\
5 & 10 & 15 & 20 & 25 & 30 & 35 & 40 \\
40 & 45 & 50 & 55 & 60 & 65 & 70 & 75 \\
80 & 85 & 90 & 95 & 100
\end{array}
\]

(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.

---

Example partial_tile.1.c (omp_5.1)

```c
int min(int a, int b) { return (a < b)? a : b; }

void func1(double A[100][100])
{
    #pragma omp tile sizes(4,16)
    for (int i = 0; i < 100; ++i)
        for (int j = 0; j < 100; ++j)
}

void func2(double A[100][100])
{
    for (int i1 = 0; i1 < 100; i1+=4)
        for (int j1 = 0; j1 < 100; j1+=16)
            for (int i2 = i1; i2 < i1+4; ++i2)
                for (int j2 = j1; j2 < min(j1+16,100); ++j2)
}

void func3(double A[100][100])
{
    // complete tiles
    for (int i1 = 0; i1 < 100; i1+=4)
        for (int j1 = 0; j1 < 96; j1+=16)
            for (int i2 = i1; i2 < i1+4; ++i2)
                for (int j2 = j1; j2 < j1+16; ++j2)
    // partial tiles / remainder
    for (int i1 = 0; i1 < 100; i1+=4)
        for (int i2 = i1; i2 < i1+4; ++i2)
            for (int j = 96; j < 100; j+=1)
}

void func4(double A[100][100])
{
    for (int i1 = 0; i1 < 100; i1+=4) {
        // complete tiles
        for (int j1 = 0; j1 < 96; j1+=16)
```
for (int i2 = i1; i2 < i1+4; ++i2)
    for (int j2 = j1; j2 < j1+16; ++j2)

// partial tiles
for (int i2 = i1; i2 < i1+4; ++i2)
    for (int j = 96; j < 100; j+=1)
}
}

void func5(double A[100][100])
{
    for (int i1 = 0; i1 < 100; i1+=4)
        for (int j1 = 0; j1 < 100; j1+=16)
            for (int i2 = i1; i2 < i1+4; ++i2)
                for (int j2 = j1; j2 < j1+16; ++j2)
                    if (j2 < 100)
}

void func6(double A[100][100])
{
    // complete tiles
    for (int i1 = 0; i1 < 100; i1+=4)
        for (int j1 = 0; j1 < 96; j1+=16)
            for (int i2 = i1; i2 < i1+4; ++i2)
                for (int j2 = j1; j2 < j1+16; ++j2)
    // partial tiles / remainder (not tiled)
    for (int i = 0; i < 100; ++i)
        for (int j = 96; j < 100; ++j)
}

Example partial_tile.1.f90 (omp_5.1)

subroutine func1(A)
    implicit none
    double precision :: A(100,100)
    integer :: i,j

    !$omp tile sizes(4,16)
    do i = 1, 100
        do j = 1, 100
...
A(j,i) = A(j,i) + 1
end do; end do

end subroutine

subroutine func2(A)
  implicit none
  double precision :: A(100,100)
  integer :: i1,i2,j1,j2

  do i1 = 1, 100, 4
    do j1 = 1, 100, 16
      do i2 = i1, i1 + 3
        do j2 = j1, min(j1+15,100)
          A(j2,i2) = A(j2,i2) + 1
        end do; end do; end do; end do
  end do; end do; end do; end do

end subroutine

subroutine func3(A)
  implicit none
  double precision :: A(100,100)
  integer :: i1,i2,j1,j2, j

  !! complete tiles
  do i1 = 1, 100, 4
    do j1 = 1, 96, 16
      do i2 = i1, i1 + 3
        do j2 = j1, j1 +15
          A(j2,i2) = A(j2,i2) + 1
        end do; end do; end do; end do
  end do; end do; end do; end do

  !! partial tiles / remainder
  do i1 = 1, 100, 4
    do i2 = i1, i1 +3
      do j = 97, 100
        A(j,i2) = A(j,i2) + 1
      end do; end do; end do
  end do; end do; end do; end do

end subroutine

subroutine func4(A)
  implicit none
  double precision :: A(100,100)
integer :: i1, i2, j1, j2, j

do i1 = 1, 100, 4
!! complete tiles
do j1 = 1, 96, 16
do i2 = i1, i1 + 3
do j2 = j1, j1 +15
A(j2, i2) = A(j2, i2) + 1
end do; end do; end do
!! partial tiles
do i2 = i1, i1 +3
do j = 97, 100
A(j, i2) = A(j, i2) + 1
end do; end do

end subroutine

subroutine func5(A)
implicit none
double precision :: A(100, 100)
integer :: i1, i2, j1, j2

do i1 = 1, 100, 4
do j1 = 1, 100, 16
do i2 = i1, i1 + 3
do j2 = j1, j1 +15
if (j2 < 101) A(j2, i2) = A(j2, i2) + 1
end do; end do; end do

end subroutine

subroutine func6(A)
implicit none
double precision :: A(100, 100)
integer :: i1, i2, j1, j2, i, j
!! complete tiles
do i1 = 1, 100, 4
do j1 = 1, 96, 16
do i2 = i1, i1 + 3
do j2 = j1, j1 +15
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
#include <stdio.h>

int min(int a, int b){ return (a < b) ? a : b; }

void func7(double A[100][100])
{
    #pragma omp parallel for schedule(static)
    #pragma omp tile sizes(4,16)
    for (int i = 0; i < 100; ++i)
        for (int j = 0; j < 100; ++j)
}

void func8(double A[100][100])
{
    #pragma omp parallel for schedule(static)
    for (int i1 = 0; i1 < 100; i1+=4)
        for (int j1 = 0; j1 < 100; j1+=16)
            for (int i2 = i1; i2 < i1+4; ++i2)
                for (int j2 = j1; j2 < min(j1+16,100); ++j2)
}
```

C / C++

void func9(double A[100][100])
{
    #pragma omp parallel
    {
        #pragma omp for schedule(static) nowait
        for (int i1 = 0; i1 < 100; i1+=4)
            for (int j1 = 0; j1 < 96; j1+=16)
                for (int i2 = i1; i2 < i1+4; ++i2)
                    for (int j2 = j1; j2 < j1+16; ++j2)
        #pragma omp for schedule(static)
        for (int i1 = 0; i1 < 100; i1+=4)
            for (int i2 = i1; i2 < i1+4; ++i2)
                for (int j = 96; j < 100; j+=1)
    }
}

Example partial_tile.2.f90 (omp_5.1)

subroutine func7(A)
    implicit none
    double precision    :: A(100,100)
    integer             :: i,j

    !$omp parallel do schedule(static)
    !$omp tile sizes(4,16)
    do i=1,100
        do j = 1, 100
            A(j,i) = A(j,i) + 1
        end do; end do
end subroutine

subroutine func8(A)
    implicit none
    double precision    :: A(100,100)
    integer             :: i1,i2,j1,j2

    do i1 = 1, 100, 4
        do j1 = 1, 100, 16

CHAPTER 8. LOOP TRANSFORMATIONS  305
S-23     do i2 = i1, i1 + 3
S-24     do j2 = j1, min(j1+15,100)
S-25          A(j2,i2) = A(j2,i2) + 1
S-26     end do; end do; end do; end do
S-27
S-28     end subroutine
S-29
S-30     subroutine func9(A)
S-31         implicit none
S-32         double precision :: A(100,100)
S-33         integer :: i1,i2,j1,j2,j
S-34
S-35          !$omp parallel
S-36
S-37          !$omp do schedule(static)
S-38          do i1 = 1, 100, 4
S-39          do j1 = 1, 96, 16
S-40          do i2 = i1, i1 + 3
S-41          do j2 = j1, j1 +15
S-42              A(j2,i2) = A(j2,i2) + 1
S-43          end do; end do; end do; end do
S-44          !$omp end do nowait
S-45
S-46          !$omp do schedule(static)
S-47          do i1 = 1, 100, 4
S-48          do i2 = i1, i1 +3
S-49          do j = 97, 100
S-50              A(j,i2) = A(j,i2) + 1
S-51          end do; end do; end do;
S-52          !$omp end parallel
S-53
S-54     end subroutine
S-55
S-56
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 do-across 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 $yaxis$.

Example critical.1.c

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

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

Example critical.1.f

```fortran
SUBROUTINE CRITICAL_EXAMPLE(X, Y)
REAL X(*), Y(*)
INTEGER IX_NEXT, IY_NEXT
!$OMP PARALLEL SHARED(X, Y) PRIVATE(IX_NEXT, IY_NEXT)
{...
    !$OMP CRITICAL(XAXIS)
    CALL DEQUEUE(IX_NEXT, X)
    !$OMP END CRITICAL(XAXIS)
    CALL WORK(IX_NEXT, X)
    !$OMP CRITICAL(YAXIS)
    CALL DEQUEUE(IY_NEXT, Y)
    !$OMP END CRITICAL(YAXIS)
    CALL WORK(IY_NEXT, Y)
}
```
The following example extends the previous example by adding the `hint` clause to the `critical` constructs.

```
#include <omp.h>

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

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

        #pragma omp critical (yaxis) hint(omp_sync_hint_contended)
        iy_next = dequeue(y);
        work(iy_next, y);
    }
}
```
SUBROUTINE CRITICAL_EXAMPLE(X, Y)
    USE OMP_LIB ! or INCLUDE "omp_lib.h"
    REAL X(*), Y(*)
    INTEGER IX_NEXT, IY_NEXT

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

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

    !$OMP END PARALLEL
    END SUBROUTINE CRITICAL_EXAMPLE
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
void critical_work()
{
    int i = 1;
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            #pragma omp critical (name)
            {
                #pragma omp parallel
                {
                    #pragma omp single
                    {
                        i++;
                    }
                }
            }
        }
    }
```

Example worksharing_critical.1.c

```c
S-20 }
```
Example worksharing_critical.1.f

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

```
Example barrier_regions.1.c

void work(int n) {}

void sub3(int n)
{
    work(n);
    #pragma omp barrier
    work(n);
}

void sub2(int k)
{
    #pragma omp parallel shared(k)
    sub3(k);
}

void sub1(int n)
{
    int i;
    #pragma omp parallel private(i) shared(n)
    {
        #pragma omp for
        for (i=0; i<n; i++)
            sub2(i);
    }
}

int main()
{
    sub1(2);
    sub2(2);
}
```
Example barrier_regions.1.f

SUBROUTINE WORK(N)
    INTEGER N
END SUBROUTINE WORK

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

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

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

PROGRAM EXAMPLE
    CALL SUB1(2)
    CALL SUB2(2)
    CALL SUB3(2)
END PROGRAM EXAMPLE
9.4 atomic Construct

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

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

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

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

Example atomic.1.f (omp_3.1)

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

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

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

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

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

    DO I=1,10000
        INDEX(I) = MOD(I, 1000) + 1
        Y(I) = 0.0
    ENDDO

CHAPTER 9. SYNCHRONIZATION
317
The following example illustrates the read and write clauses for the atomic directive. These clauses ensure that the given variable is read or written, respectively, as a whole. Otherwise, some other thread might read or write part of the variable while the current thread was reading or writing another part of the variable. Note that most hardware provides atomic reads and writes for some set of properly aligned variables of specific sizes, but not necessarily for all the variable types supported by the OpenMP API.

```
Example atomic.2.c (omp_3.1)

int atomic_read(const int *p)
{
    int value;
    /* Guarantee that the entire value of *p is read atomically. No part of
    * *p can change during the read operation.
    */
    #pragma omp atomic read
    value = *p;
    return value;
}

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

```
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.
Use fetch_and_add to implement a lock

```c
struct locktype {
    int ticketnumber;
    int turn;
};
```

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

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

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

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

---

**Example atomic.3.f (omp_3.1)**

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

! 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
```

```fortran
module m
interface
    function fetch_and_add(p)
        integer :: fetch_and_add
        integer, intent(inout) :: p
    end function
end interface
```

---

C / C++
Fortran
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
9.5 Restrictions on the \texttt{atomic} Construct

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

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

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

```
Example atomic_restrict.1.f (omp_3.1)

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

Example atomic_restrict.2.c (omp_3.1)

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

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

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

        #pragma omp atomic update
        *r += 1.0;
    }

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

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

Example atomic_restrict.2.f (omp_3.1)

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

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

SUBROUTINE ATOMIC_WRONG2()
    COMMON /BLK/ I
    INTEGER I

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

*Example atomic_restrict.3.f (omp_3.1)*

```fortran
!$OMP PARALLEL
!$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
9.6 flush Construct without a List

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

```
Example flush_nolist.1.c

int x, *p = &x;

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

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

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

    return sum;
}
```

```
/* because they were not accessible in f2 */
/* j was flushed because it was accessible */
sum += i + j + *p + n;
return sum;
}
}

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

Example flush_nolist.1.f

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

    Q = 1

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

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

    !$OMP BARRIER
    Q = 2

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

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

I = 1
SUM = 0
P = 1

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

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

!$OMP END PARALLEL

G = SUM
END FUNCTION G

PROGRAM FLUSH_NOLIST
COMMON /DATA/ X, P
INTEGER, TARGET :: X
INTEGER, POINTER :: P
INTEGER RESULT, G
P => X
RESULT = G(7)
PRINT *, RESULT
END PROGRAM FLUSH_NOLIST

Fortran
9.7 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 pairwise synchronization from the exit of one critical region to the entry to the next critical region. These pairwise 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.
Example acquire_release.1.c (omp_5.0)

```c
#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;
}
```

Example acquire_release.1.f90 (omp_5.0)

```fortran
program rel_acq_ex1
    use omp_lib
    integer :: x, y, thrd, tmp
    x = 0
    y = 0
    !$omp parallel num_threads(2) private(thrd, tmp)
       thrd = omp_get_thread_num()
       if (thrd == 0) then
           x = 10
           !$omp critical
           y = 1
           !$omp end critical
       else
```
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”. 
Example acquire_release.2.f90 (omp_5.0)

```fortran
program rel_acq_ex2
    use omp_lib
    integer :: x, y, thrd, tmp
    x = 0
    y = 0
    !$omp parallel num_threads(2) private(thrd, tmp)
    thrd = omp_get_thread_num()
    if (thrd == 0) then
        x = 10
        !$omp atomic write release ! or seq_cst
        y = 1
        !$omp end atomic
    else
        tmp = 0
        do while (tmp == 0)
            !$omp atomic read acquire ! or seq_cst
            tmp = y
            !$omp end atomic
        end do
        print *, "x = ", x !! always "x = 10"
    end if
    !$omp end parallel
end program
```

CHAPTER 9. SYNCHRONIZATION 331
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.

```
#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 flush // or with acq_rel or release clause
            #pragma omp atomic write // or with relaxed clause
            y = 1;
        } else {
            int tmp = 0;
            while (tmp == 0) {
                #pragma omp atomic read // or with relaxed clause
                tmp = y;
            }
            #pragma omp flush // or with acq_rel or acquire clause
            printf("x = %d\n", x); // always "x = 10"
        }
    }
    return 0;
}
```

---

Example acquire_release.3.c (omp_5.0)
Example acquire_release.3.f90 (omp_5.0)

```
program rel_acq_ex3
  use omp_lib
  integer :: x, y, thrd, tmp
  x = 0
  y = 0
  !$omp parallel num_threads(2) private(thrd, tmp)
     thrd = omp_get_thread_num()
     if (thrd == 0) then
       x = 10
       !$omp flush ! or with acq_rel or release clause
       !$omp atomic write
       y = 1
       !$omp end atomic
     else
       tmp = 0
       do while (tmp == 0)
         !$omp atomic read
         tmp = y
         !$omp end atomic
       end do
       !$omp flush ! or with acq_rel or acquire clause
       print *, "x = ", x !! always "x = 10"
     end if
  !$omp end parallel
end program
```

Example 4 will fail to order the write to \textit{x} on thread 0 before the read from \textit{x} on thread 1. Importantly, the implicit release flush on exit from the \texttt{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 \texttt{critical} construct, this might correspond to a \textit{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 \textit{Implicit Flushes} section of the OpenMP Specifications document). Either an explicit \texttt{flush} directive that provides a release flush (i.e., a flush without a list that does not have the \texttt{acquire} clause) must be specified between the \texttt{critical} construct and the atomic write, or an atomic operation that modifies \textit{y} and provides release semantics must be specified.
Example acquire_release_broke.4.c (omp_5.0)

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

int main()
{
    int x = 0, y;
    #pragma omp parallel num_threads(2)
    {
        int thrd = omp_get_thread_num();
        if (thrd == 0) {
            #pragma omp critical
            { x = 10; }
            // an explicit flush directive that provides
            // release semantics is needed here
            #pragma omp atomic write
            y = 1;
        } else {
            int tmp = 0;
            while (tmp == 0) {
                #pragma omp atomic read acquire // or seq_cst
                tmp = y;
            }
            #pragma omp critical
            { printf("x = %d\n", x); } // !! NOT ALWAYS 10
        }
    }
    return 0;
}
```

Example acquire_release_broke.4.f90 (omp_5.0)

```
program rel_acq_ex4
  use omp_lib
  integer :: x, y, thrd
  integer :: tmp
  x = 0

!! !!! THIS CODE WILL FAIL TO PRODUCE CONSISTENT RESULTS !!!!!!
!! !!! DO NOT PROGRAM SYNCHRONIZATION THIS WAY !!!!!!

!$omp parallel num_threads(2) private(thrd) private(tmp)
  thrd = omp_get_thread_num()
  if (thrd == 0) then
    !$omp critical
    x = 10
    !$omp end critical
    ! an explicit flush directive that provides
    ! release semantics is needed here to
    ! complete the synchronization.
    !$omp atomic write
    y = 1
    !$omp end atomic
  else
    tmp = 0
    do while(tmp == 0)
      !$omp atomic read acquire ! or seq_cst
      !$omp end atomic
    end do
    !$omp critical
    print *, "x = ", x !! !! NOT ALWAYS 10
    !$omp end critical
  end if
!$omp end parallel
end program
```
9.8 ordered Clause and ordered Construct

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

```c
#include <stdio.h>

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

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

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

```fortran
SUBROUTINE WORK(K)
    INTEGER k

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

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

---

OpenMP Examples Version 5.2 - April 2022
It is possible to have multiple ordered constructs within a loop region with the ordered clause specified. The first example is non-conforming because all iterations execute two ordered regions. An iteration of a loop must not execute more than one ordered region:

```
void work(int i) {} 
void ordered_wrong(int n) {
    int i;
    #pragma omp for ordered
    for (i=0; i<n; i++) {
        /* incorrect because an iteration may not execute more than one ordered region */
        #pragma omp ordered
        work(i);
        #pragma omp ordered
        work(i+1);
    }
}
```
Fortran

Example ordered.2.f

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

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

Fortran

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

C / C++

Example ordered.3.c

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

```fortran
SUBROUTINE ORDERED_GOOD(N)
  INTEGER N
  !$OMP DO ORDERED
  DO I = 1,N
    IF (I <= 10) THEN
      !$OMP ORDERED
      CALL WORK(I)
      !$OMP END ORDERED
    ENDIF
  ENDIF
  IF (I > 10) THEN
    !$OMP ORDERED
    CALL WORK(I+1)
    !$OMP END ORDERED
  ENDIF
  ENDDO
END SUBROUTINE ORDERED_GOOD
```

CHAPTER 9. SYNCHRONIZATION
9.9 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. The dependence is created from a dependence type and a storage location, within a 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++ (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 uninitialized a depend object (obj).

The first depobj construct initializes the obj depend object with an inout dependence type 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.

### Example depobj.1.c (omp_5.2)

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

#define N 100
#define TRUE 1
#define FALSE 0

void driver(int update, float a[], float b[], int n, omp_depend_t *obj);

void update_copy(int update, float a[], float b[], int n);

void checkpoint(float a[], int n);

void init(float a[], int n);

int main()
```
```c
float a[N], b[N];
omp_depend_t obj;

init(a, N);

#pragma omp depobj(obj) depend(inout: a)
driver(TRUE, a, b, N, &obj); // updating a occurs

#pragma omp depobj(obj) update(in)
driver(FALSE, a, b, N, &obj); // no updating of a

#pragma omp depobj(obj) destroy(obj) // obj is set to uninitialized
// state, resources are freed
return 0;

void driver(int update, float a[], float b[], int n, omp_depend_t *obj)
{
    #pragma omp parallel num_threads(2)
    #pragma omp single
    {
        #pragma omp task depend(depobj: *obj) // Task 1, uses depend object
        update_copy(update, a, b, n); // may update a, always copy a to b

        #pragma omp task depend(in: a[:n]) // Task 2, only read a
        checkpoint(a, n);
    }
}

void update_copy(int update, float a[], float b[], int n)
{
    if(update) for(int i=0; i<n; i++) a[i]+=1.0f;
    for(int i=0; i<n; i++) b[i]=a[i];
}

void checkpoint(float a[], int n)
{
    for(int i=0; i<n; i++) printf(" %f \n", a[i]);
    printf("\n");
}
```
void init(float a[], int n)  
{  
    for(int i=0; i<n; i++) a[i]=i;  
}  

---

C / C++

Fortran

---

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

```
program main
    use omp_lib
    implicit none

    integer, parameter :: N=100
    real :: a(N), b(N)
    integer(omp_depend_kind) :: obj

    call init(a, N)

    !$omp depobj(obj) depend(inout: a)
    call driver(.true., a, b, N, obj)  ! updating occurs

    !$omp depobj(obj) update(in)
    call driver(.false., a, b, N, obj)  ! no updating

    !$omp depobj(obj) destroy(obj)  ! obj is set to uninitialized state, resources are freed

end program
```

```
subroutine driver(update, a, b, n, obj)
    use omp_lib
    implicit none

    logical :: update
    real :: a(n), b(n)
    integer :: n
    integer(omp_depend_kind) :: obj

    !$omp parallel num_threads(2)
    !$omp single
    !$omp task depend(depobj: obj)  ! Task 1, uses depend object
```
call update_copy(update, a, b, n)  
!! update a or not, always copy a to b
 !$omp end task

 !$omp task depend(in: a)  
!! Task 2, only read a  
call checkpoint(a, n)
 !$omp end task

 !$omp end single

 !$omp end parallel

end subroutine

subsection update_copy(update, a, b, n)
implicit none
logical :: update
real :: a(n), b(n)
integer :: n

if (update) a = a + 1.0
b = a
end subroutine

subsection checkpoint(a, n)
implicit none
integer :: n
real :: a(n)
integer :: i

write(*,'( *(f5.0) )') (a(i), i=1,n)
end subroutine

subsection init(a, n)
implicit none
integer :: n
real :: a(n)
integer :: i

a=[ (i, i=1,n) ]
end subroutine
9.10 Doacross Loop Nest

An ordered clause can be used on a loop construct with an integer parameter argument to define the number of associated loops within a doacross loop nest where cross-iteration dependences exist. A 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 loop directive, and a single ordered directive without the doacross clause specified for the statement executing the bar function.

---

Example doacross.1.c (omp_5.2)

```c
float foo(int i);
float bar(float a, float b);
float baz(float b);
void work( int N, float *A, float *B, float *C )
{
    int i;
    #pragma omp for ordered(1)
    for (i=1; i<N; i++)
    {
        A[i] = foo(i);
        #pragma omp ordered doacross(sink: i-1)
        B[i] = bar(A[i], B[i-1]);
        #pragma omp ordered doacross(source: omp_cur_iteration)
        C[i] = baz(B[i]);
    }
}
```
Example doacross.1.f90 (omp_5.2)

subroutine work( N, A, B, C )
   integer :: N, i
   real, dimension(N) :: A, B, C
   real, external :: foo, bar, baz
   !$omp do ordered(1)
   do i=2, N
      A(i) = foo(i)
      !$omp ordered doacross(sink: i-1)
      B(i) = bar(A(i), B(i-1))
      !$omp ordered doacross(source: omp_cur_iteration)
      C(i) = baz(B(i))
   end do
end subroutine

The following code is similar to the previous example but with doacross loop nest extended to two nested loops, i and j, as specified by the ordered(2) clause on the loop directive. In the C/C++ code, the i and j loops are the first and second associated loops, respectively, whereas in the Fortran code, the j and i loops are the first and second associated loops, respectively. The 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).

Example doacross.2.c (omp_5.2)

void work( int N, int M, float **A, float **B, float **C )
{
   int i, j;
   #pragma omp for ordered(2)
   for (i=1; i<N; i++)
   {

CHAPTER 9. SYNCHRONIZATION 345
for (j=1; j<M; j++)
{
    A[i][j] = foo(i, j);
}
#pragma omp ordered doacross(sink: i-1,j) doacross(sink: i,j-1)
    B[i][j] = bar(A[i][j], B[i-1][j], B[i][j-1]);
#pragma omp ordered doacross(source:)
    C[i][j] = baz(B[i][j]);
}
}
}

\begin{Verbatim}
Example doacross2.f90 \texttt{(omp\_5.2)}
\end{Verbatim}

The following example shows the incorrect use of the \texttt{ordered} directive with a \texttt{doacross} clause. There are two issues with the code. The first issue is a missing \texttt{ordered doacross(source:)} directive, which could cause a deadlock. The second issue is the \texttt{doacross(sink:i+1,j)} and \texttt{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.
Example doacross.3.c (omp_5.2)

```c
#define N 100

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

#pragma omp parallel for ordered(2) private(i,j,k)
    for (i=1; i<N-1; i++)
    {
        for (j=1; j<N-1; j++)
        {
            #pragma omp ordered doacross(sink: i-1,j) doacross(sink: i+1,j) \
            #pragma omp & 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;
            }
        }
    }
}
```

Example doacross.3.f90 (omp_5.2)

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

 !$omp parallel do ordered(2) private(i,j,k,tmp1,tmp2,tmp3)
 do i=2, N-1
    do j=2, N-1
        !$omp ordered 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)
        enddo
    enddo
 enddo
```

CHAPTER 9. SYNCHRONIZATION
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 subroutine

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

Example doacross.4.c (omp_5.2)

double foo(int i, int j);

void work( int N, int M, double **A, double **B, double **C )
{
  int i, j;
  double alpha = 1.2;

  #pragma omp for collapse(2) ordered(2)
  for (i = 1; i < N-1; i++)
  {
    for (j = 1; j < M-1; j++)
    {
      A[i][j] = foo(i, j);
      #pragma omp ordered doacross(source:)
      B[i][j] = alpha * A[i][j];
      #pragma omp ordered doacross(sink: i-1,j) doacross(sink: i,j-1)
      C[i][j] = 0.2 * (A[i-1][j] + A[i+1][j] + 
    }
  }
}
Example doacross.4.f90 (omp_5.2)

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

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

CHAPTER 9. SYNCHRONIZATION
9.11 Lock Routines

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

9.11.1 omp_init_lock Routine

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

```
Example init_lock.1.cpp

```
9.11.2 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.

---

**Example init_lock_with_hint.1.cpp** (omp_5.0)

```cpp
#include <omp.h>
omp_lock_t *new_locks()
{
    int i;
    omp_lock_t *lock = new omp_lock_t[1000];

    #pragma omp parallel for private(i)
    for (i=0; i<1000; i++)
    {
        omp_init_lock_with_hint(&lock[i],
            static_cast<omp_lock_hint_t>(omp_sync_hint_contended |
            omp_sync_hint_speculative));
    }

    return lock;
}
```

---

**Example init_lock_with_hint.1.f** (omp_5.0)

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

    INTEGER I

    !$OMP PARALLEL DO PRIVATE(I)
    DO I=1,1000
        CALL OMP_INIT_LOCK_WITH_HINT(NEW_LOCKS(I),
            OMP_SYNC_HINT_CONTENDED + OMP_SYNC_HINT_SPECULATIVE)
    END DO

    !$OMP END PARALLEL DO

END FUNCTION NEW_LOCKS
```

---

CHAPTER 9. SYNCHRONIZATION 351
9.11.3 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 task regions; so a lock released by the `omp_unset_lock` routine in a task region must be owned by the same task region.

This change in ownership requires extra care when using locks. The following program is conforming in OpenMP 2.5 because the thread that releases the lock `lck` in the parallel region is the same thread that acquired the lock in the sequential part of the program (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.

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

int main()
{
    int x;
    omp_lock_t lck;

    omp_init_lock (&lck);
    omp_set_lock (&lck);
    x = 0;

    #pragma omp parallel shared (x)
    {
        #pragma omp masked
        {
            x = x + 1;
            omp_unset_lock (&lck);
        }
        /* Some more stuff. */
    }
    omp_destroy_lock (&lck);
    return 0;
}
```
Example lock_owner.1.f (omp_5.1)

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

9.11.4 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
#include <stdio.h>
#include <omp.h>

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

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

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

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

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

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

        omp_unset_lock(&lck);
    }

    omp_destroy_lock(&lck);

    return 0;
}
```
Example simple_lock.1.f

```fortran
S-1 SUBROUTINE SKIP(ID)
S-2 END SUBROUTINE SKIP
S-3
S-4 SUBROUTINE WORK(ID)
S-5 END SUBROUTINE WORK
S-6
S-7 PROGRAM SIMPLELOCK
S-8
S-9 INCLUDE "omp_lib.h" ! or USE OMP_LIB
S-10
S-11 INTEGER(OMP_LOCK_KIND) LCK
S-12 INTEGER ID
S-13
S-14 CALL OMP_INIT_LOCK(LCK)
S-15
S-16 !$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
S-17 ID = OMP_GET_THREAD_NUM()
S-18 CALL OMP_SET_LOCK(LCK)
S-19 PRINT *, 'My thread id is ', ID
S-20 CALL OMP_UNSET_LOCK(LCK)
S-21
S-22 DO WHILE (.NOT. OMP_TEST_LOCK(LCK))
S-23 CALL SKIP(ID) ! We do not yet have the lock
S-24 ! so we must do something else
S-25 END DO
S-26
S-27 CALL WORK(ID) ! We now have the lock
S-28 ! and can do the work
S-29
S-30 CALL OMP_UNSET_LOCK( LCK )
S-31
S-32 !$OMP END PARALLEL
S-33
S-34 CALL OMP_DESTROY_LOCK( LCK )
S-35
S-36 END PROGRAM SIMPLELOCK
```
9.11.5 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.

```
#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)
{
  p->a += a;
}

void incr_b(pair *p, int b)
{
  omp_set_nest_lock(&p->lck);
  p->b += b;
  omp_unset_nest_lock(&p->lck);
}

void incr_pair(pair *p, int a, int b)
{
  omp_set_nest_lock(&p->lck);
  incr_a(p, a);
  incr_b(p, b);
  omp_unset_nest_lock(&p->lck);
}
```
```c
void nestlock(pair *p) {
    #pragma omp parallel sections
    {
        #pragma omp section
        incr_pair(p, work1(), work2());
        #pragma omp section
        incr_b(p, work3());
    }
}

Example nestable_lock.1.f

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

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

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

CHAPTER 9. SYNCHRONIZATION
SUBROUTINE INCR_PAIR(P, A, B)
USE OMP_LIB ! or INCLUDE "omp_lib.h"
USE DATA
TYPE(LOCKED_PAIR) :: P
INTEGER A
INTEGER B

CALL OMP_SET_NEST_LOCK(P%LCK)
CALL INCR_A(P, A)
CALL INCR_B(P, B)
CALL OMP_UNSET_NEST_LOCK(P%LCK)
END SUBROUTINE INCR_PAIR

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

 !$OMP PARALLEL SECTIONS
 !$OMP SECTION
 CALL INCR_PAIR(P, WORK1(), WORK2())
 !$OMP SECTION
 CALL INCR_B(P, WORK3())
 !$OMP END PARALLEL SECTIONS
 END SUBROUTINE NESTLOCK
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 (+1 on entry and -1 on exited) in nested (structured) map regions and/or accumulative (unstructured) mappings, determines the operation. Details of the `map` clause and reference count operation are specified in the `map Clause` subsection of the OpenMP Specifications document.
10.1 threadprivate Directive

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

```
Example threadprivate.1.c
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  }
Example threadprivate.1.f
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
```

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

```
Example threadprivate.2.c
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

```c++
// 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   }
```
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*

```fortran
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*

```fortran
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*

```fortran
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       !$OMP PARALLEL COPYIN(/T/)
S-10      END SUBROUTINE INC_GOOD_SUB
S-11      END SUBROUTINE INC_GOOD
```
The following is an example of the use of `threadprivate` for local variables:

```
FUNCTION threadprivate.5.f

PROGRAM INC_GOOD2
   INTEGER, ALLOCATABLE, SAVE :: A(:)
   INTEGER, POINTER, SAVE :: PTR
   INTEGER, SAVE :: I
   INTEGER, TARGET :: TARG
   LOGICAL :: FIRSTIN = .TRUE.
   !$OMP THREADPRIVATE(A, I, PTR)
   ALLOCATE (A(3))
   A = (/1, 2, 3/)  
   PTR => TARG
   I = 5
   !$OMP PARALLEL COPYIN(I, PTR)
   !$OMP CRITICAL
   IF (FIRSTIN) THEN
      TARG = 4  ! Update target of ptr
      I = I + 10
      IF (ALLOCATED(A)) A = A + 10
      FIRSTIN = .FALSE.
   END IF
   IF (ALLOCATED(A)) THEN
      PRINT *, 'a = ', A
   ELSE
      PRINT *, 'A is not allocated'
   END IF
   PRINT *, 'ptr = ', PTR
   PRINT *, 'i = ', I
   PRINT *
   !$OMP END CRITICAL
   !$OMP END PARALLEL
END PROGRAM INC_GOOD2
```

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


Fortran (cont.)

\[ a = 11 \quad 12 \quad 13 \]
\[ \text{ptr} = 4 \]
\[ i = 15 \]

A is not allocated
\[ \text{ptr} = 4 \]
\[ i = 5 \]

or

A is not allocated
\[ \text{ptr} = 4 \]
\[ i = 15 \]
\[ a = 1 \quad 2 \quad 3 \]
\[ \text{ptr} = 4 \]
\[ i = 5 \]

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

\textit{Example threadprivate.6.f}

\begin{verbatim}
MODULE INC_MODULE_GOOD3
  REAL, POINTER :: WORK(:)
  SAVE WORK
  !$OMP THREADPRIVATE(WORK)
  END MODULE INC_MODULE_GOOD3

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

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

PROGRAM INC_GOOD3
  N = 10
  CALL SUB1(N)
\end{verbatim}
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()`:

```cpp
Example threadprivate.4.cpp
struct T { T(); T(int); ~T(); int t; };
int f();
static T t1;
#pragma omp threadprivate(t1)
static T t2(23);
#pragma omp threadprivate(t2)
static T t3 = f();
#pragma omp threadprivate(t3)
```

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

```cpp
Example threadprivate.5.cpp
class T {
public:
static int i;
#pragma omp threadprivate(i)
};
```
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
#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 */
}
```

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.f

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
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.

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

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

    #pragma omp parallel private(i) firstprivate(j)
    {
        i = 3;
        j = j + 2;
        assert (*ptr_i == 1 && *ptr_j == 2);
    }

    assert(i == 1 && j == 2);
    return 0;
}
```
In the following example, all uses of the variable `a` within the loop construct in the routine `f` refer to a private list item `a`, while it is unspecified whether references to `a` in the routine `g` are to a private list item or the original list item.
Fortran

Example private.2.f

```fortran
MODULE PRIV_EXAMPLE2

REAL A

CONTAINS

SUBROUTINE G(K)

REAL K

A = K ! Accessed in the region but outside of the
! construct; therefore unspecified whether
! original or private list item is modified.

END SUBROUTINE G

SUBROUTINE F(N)

INTEGER N

REAL A

INTEGER I

!$OMP PARALLEL DO PRIVATE(A)

DO I = 1,N

A = I

CALL G(A*2)

ENDDO

!$OMP END PARALLEL DO

END SUBROUTINE F

END MODULE PRIV_EXAMPLE2
```

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

```c
#include <assert.h>

void priv_example3()
{

int i, a;

#pragma omp parallel private(a)

{ a = 1;

#pragma omp parallel for private(a)
```
for (i=0; i<10; i++)
{
    a = 2;
}
assert(a == 1);
}

Example private.3.f

SUBROUTINE PRIV_EXAMPLE3()
    INTEGER I, A
    !$OMP PARALLEL PRIVATE(A)
        A = 1
    !$OMP PARALLEL DO PRIVATE(A)
        DO I = 1, 10
            A = 2
        END DO
    !$OMP END PARALLEL DO
    PRINT *, A ! Outer A still has value 1
    !$OMP END PARALLEL
END SUBROUTINE PRIV_EXAMPLE3
In general loop iteration variables will be private, when used in the do-loop of a do and parallel do construct or in sequential loops in a parallel construct (see Section 2.7.1 and Section 2.14.1 of the OpenMP 4.0 specification). In the following example of a sequential loop in a parallel construct the loop iteration variable \( I \) will be private.

*Example fort_loopvar.1.f90*

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

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

*Example fort_loopvar.2.f90*

```fortran
SUBROUTINE PLOOP_2(A,B,N,I1,I2)
REAL A(*), B(*)
INTEGER I1, I2, N
MYOFFSET = OMP_GET_THREAD_NUM() * N
DO I = 1, N
A(MYOFFSET+I) = FLOAT(I)
ENDDO
!$OMP END PARALLEL
END SUBROUTINE PLOOP_1
```
Note however that the use of shared loop iteration variables can easily lead to race conditions.
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

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

The following example is also conforming:

Example fort_sp_common.2.f

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

*Example fort_sp_common.3.f*

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

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

*Example fort_sp_common.4.f*

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

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

*Example fort_sp_common.5.f*

```fortran
S-1  SUBROUTINE COMMON_WRONG2()
S-2      COMMON /C/ X, Y
S-3    ! Incorrect: common block C cannot be declared both
S-4    ! shared and private
S-5     !$OMP PARALLEL PRIVATE (/C/), SHARED(/C/)
S-6           ! do work here
S-7     !$OMP END PARALLEL
S-8   END SUBROUTINE COMMON_WRONG2
S-9
```

---

376 OpenMP Examples Version 5.2 - April 2022
10.6 Fortran Restrictions on Storage Association with the \texttt{private} Clause

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

\textit{Example fort_sa_private.1.f}

\begin{verbatim}
SUBROUTINE SUB()
    COMMON /BLOCK/ X
    PRINT *,X ! X is undefined
END SUBROUTINE SUB

PROGRAM PRIV_RESTRICT
    COMMON /BLOCK/ X
    X = 1.0
    !$OMP PARALLEL PRIVATE (X)
    X = 2.0
    CALL SUB()
    !$OMP END PARALLEL
END PROGRAM PRIV_RESTRICT
\end{verbatim}

\textit{Example fort_sa_private.2.f}

\begin{verbatim}
PROGRAM PRIV_RESTRICT2
    COMMON /BLOCK2/ X
    X = 1.0

    !$OMP PARALLEL PRIVATE (X)
    X = 2.0
    CALL SUB()
    !$OMP END PARALLEL

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

    END PROGRAM PRIV_RESTRICT2
\end{verbatim}
Example fort_sa_private.3.f

```fortran
PROGRAM PRIV_RESTRICT3
   EQUIVALENCE (X,Y)
   X = 1.0
   !$OMP PARALLEL PRIVATE(X)
   PRINT *,Y ! Y is undefined
   Y = 10
   PRINT *,X ! X is undefined
   !$OMP END PARALLEL
END PROGRAM PRIV_RESTRICT3
```

Example fort_sa_private.4.f

```fortran
PROGRAM PRIV_RESTRICT4
   INTEGER I, J
   INTEGER A(100), B(100)
   EQUIVALENCE (A(51), B(1))
   !$OMP PARALLEL DO DEFAULT(PRIVATE) PRIVATE(I,J) LASTPRIVATE(A)
   DO I=1,100
      DO J=1,100
         B(J) = J - 1
      ENDDO
   S-10
   DO J=1,100
      A(J) = J ! B becomes undefined at this point
   ENDDO
   DO J=1,50
      B(J) = B(J) + 1 ! B is undefined
      A(J) = J ! A becomes undefined at this point
   ENDDO
   !$OMP END PARALLEL DO ! The LASTPRIVATE write for A has undefined results
   PRINT *, B ! B is undefined since the LASTPRIVATE write of A was not defined
END PROGRAM PRIV_RESTRICT4
```

Example fort_sa_private.5.f (omp_5.1)

```fortran
SUBROUTINE SUB1(X)
   DIMENSION X(10)
```

OpenMP Examples Version 5.2 - April 2022
This use of X does not conform to the specification. It would be legal Fortran 90, but the OpenMP private directive allows the compiler to break the sequence association that A had with the rest of the common block.

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

END SUBROUTINE SUB1

PROGRAM PRIV_RESTRICT5
  COMMON /BLOCK5/ A

  DIMENSION B(10)
  EQUIVALENCE (A,B(1))

  ! the common block has to be at least 10 words
  A = 0

  !$OMP PARALLEL PRIVATE(/BLOCK5/)

  ! Without the private clause,
  ! we would be passing a member of a sequence
  ! that is at least ten elements long.
  ! With the private clause, A may no longer be sequence-associated.

  CALL SUB1(A)

  !$OMP MASKED
  PRINT *, A
  !$OMP END MASKED

  !$OMP END PARALLEL
## 10.7 C/C++ Arrays in a firstprivate Clause

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

In this example:

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

Note that `B` and `E` involve variable length array types.

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

*Example carrays_fpriv.1.c*

```
#include <assert.h>

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

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

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

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

        /* Private B and C have values of original B and C. */
        assert(&B[1][1] == &A[1][1]);
        assert(&C[3] == &A[1][1]);
        assert(D[1][1] == 4);
    }
```
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 }
10.8 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

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

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

    a[i]=b[i];  /* i == n-1 here */
}
```

Fortran

Example lastprivate.1.f

```fortran
SUBROUTINE LASTPRIV(N, A, B)

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

!$OMP PARALLEL
!$OMP DO LASTPRIVATE(I)
DO I=1,N-1
    A(I) = B(I) + B(I+1)
ENDDO

!$OMP END PARALLEL
A(I) = B(I) ! I has the value of N here
END SUBROUTINE LASTPRIV
```
The next example illustrates the use of the \textit{conditional} modifier in a \textit{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.

\begin{verbatim}
C / C++

Example lastprivate.2.c (omp_5.0)
#include <math.h>

float condlastprivate(float *a, int n)
{
    float x = 0.0f;
    #pragma omp parallel for simd lastprivate(conditional: x)
    for (int k = 0; k < n; k++) {
        if (a[k] < 108.5 || a[k] > 208.5) {
            x = sinf(a[k]);
        }
    }
    return x;
}

Fortran

Example lastprivate.2.f90 (omp_5.0)
function condlastprivate(a, n) result(x)
    implicit none
    real a(*), x
    integer n, k
    x = 0.0
    !$omp parallel do simd lastprivate(conditional: x)
    do k = 1, n
        if (a(k) < 108.5 .or. a(k) > 208.5) then
            x = sin(a(k))
        endif
    end do
    end function condlastprivate
\end{verbatim}
10.9 Reduction

This section covers ways to perform reductions in parallel, task, taskloop, and SIMD regions.

10.9.1 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
#include <math.h>

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

---

384 OpenMP Examples Version 5.2 - April 2022
Example reduction.1.f90

```fortran
SUBROUTINE REDUCTION1(A, B, C, D, X, Y, N)
  REAL :: X(*), A, D
  INTEGER :: Y(*), N, B, C
  INTEGER :: I
  A = 0
  B = 0
  C = Y(1)
  D = X(1)
  !$OMP PARALLEL DO PRIVATE(I) SHARED(X, Y, N) REDUCTION(+:A) &
        !$OMP& REDUCTION(IEOR:B) REDUCTION(MIN:C) REDUCTION(MAX:D)
  DO I=1,N
     A = A + X(I)
     B = IEOR(B, Y(I))
     C = MIN(C, Y(I))
     IF (D < X(I)) D = X(I)
  END DO
END SUBROUTINE REDUCTION1
```

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

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

CHAPTER 10. DATA ENVIRONMENT 385
for (i=0; i<n; i++) {
    a_p += x[i];
    b_p ^= y[i];
    if (c_p > y[i]) c_p = y[i];
    d_p = fmaxf(d_p, x[i]);
}
#pragma omp critical
{
    a += a_p;
    b ^= b_p;
    if (c > c_p) c = c_p;
    d = fmaxf(d, d_p);
}
}

Example reduction.2.f90

SUBROUTINE REDUCTION2(A, B, C, D, X, Y, N)
REAL :: X(*), A, D
INTEGER :: Y(*), N, B, C
REAL :: A_P, D_P
INTEGER :: I, B_P, C_P
A = 0
B = 0
C = Y(1)
D = X(1)
!$OMP PARALLEL SHARED(X, Y, A, B, C, D, N) &
A_P = 0.0
B_P = 0
C_P = HUGE(C_P)
D_P = -HUGE(D_P)
!$OMP DO PRIVATE(I)
DO I=1,N
    A_P = A_P + X(I)
    B_P = IEOR(B_P, Y(I))
    C_P = MIN(C_P, Y(I))
    IF (D_P < X(I)) D_P = X(I)
END DO
!$OMP CRITICAL
A = A + A_P
B = IEOR(B, B_P)
C = MIN(C, C_P)
D = MAX(D, D_P)
The following program is non-conforming because the reduction is on the intrinsic procedure name MAX but that name has been redefined to be the variable named MAX.

Example reduction.3.f90

```fortran
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
```

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

```fortran
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
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 \texttt{for} loop. Therefore, the example may print an incorrect value for \(a\).

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

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 **red**uction 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 **red**uction clause in Fortran is not permitted. But this oversight has been fixed in the OpenMP 5.0 specification.
Example reduction.7.c (omp_4.5)

```c
#include <stdio.h>

#define N 100

void init(int n, float (*b)[N]);

int main(){
    int i,j;
    float a[N], b[N][N];
    init(N,b);
    for(i=0; i<N; i++) a[i]=0.0e0;

    #pragma omp parallel for reduction(+:a[0:N]) private(j)
    for(i=0; i<N; i++){
        for(j=0; j<N; j++)
            a[j] += b[i][j];
    }
    printf(" a[0] a[N-1]: %f %f\n", a[0], a[N-1]);
    return 0;
}
```

Example reduction.7.f90

```fortran
program array_red

integer,parameter :: n=100
integer :: j
real :: a(n), b(n,n)

call init(n,b)

a(:) = 0.0e0

!$omp parallel do reduction(+:a)
    do j = 1, n
        a(:) = a(:) + b(:,j)
    end do
```
10.9.2 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).

### Example task_reduction.1.c (omp_5.0)

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

```fortran
print*, " a(1) a(n): ", a(1), a(n)
end program
```
```c
int main(int argc, char *argv[]) {
    int i;

    // Create the root node.
    node_t* root = (node_t*) malloc(sizeof(node_t));
    root->val = 1;

    node_t* aux = root;

    // Create N-1 more nodes.
    for(i=2;i<=N;++i){
        aux->next = (node_t*) malloc(sizeof(node_t));
        aux = aux->next;
        aux->val = i;
    }
    aux->next = 0;

    #pragma omp parallel
    #pragma omp single
    {
        int result = linked_list_sum(root);
        printf( "Calculated: %d Analytic:%d\n", result, (N*(N+1)/2) );
    }
    return 0;
}
```

```
Example task_reduction.1.f90 (omp_5.0)
```
implicit none

type(node_t), pointer :: p

type(node_t), pointer :: aux

integer :: res

res = 0

!$omp taskgroup task_reduction(+: res)
  aux => p
  do while (associated(aux))
    !$omp task in_reduction(+: res)
    res = res + aux%val
  !$omp end task
  aux => aux%next
  end do
!$omp end taskgroup

end function linked_list_sum

program main
  use m
  implicit none
  type(node_t), pointer :: root, aux
  integer :: res, i
  integer, parameter :: N=10

  interface
    function linked_list_sum(p) result(res)
      use m
      implicit none
      type(node_t), pointer :: p
      integer :: res
    end function
  end interface

  ! Create the root node.
  allocate(root)
  root%val = 1
  aux => root

  ! Create N-1 more nodes.
  do i = 2,N
    allocate(aux%next)
    aux => aux%next
    aux%val = i
    end do

  aux%next => null()
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) in 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.

---

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

```c
#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)
```
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
10.9.3 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.
# Example target_reduction.1.c (omp_5.0)

```c
#include <stdio.h>
int f(int);
int g(int);
int main()
{
    int sum1=0, sum2=0;
    int i;
    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; }
```

# Example target_reduction.1.f90 (omp_5.0)

```fortran
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
```
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)

```c
#include <stdio.h>

int f(int);

int g(int);

int main()
{
    int sum1=0, sum2=0;
    int i;
    const int n = 100;

    #pragma omp target data map(sum1,sum2)
    {
        #pragma omp target teams distribute reduction(+:sum1)
        for (int i = 0; i < n; i++) {
```
Example target_reduction.2.f90 (omp_5.0)

```fortran
program target_reduction_ex2
  use omp_lib
  implicit none
  integer :: sum1, sum2, i
  integer, parameter :: n = 100
  sum1 = 0
  sum2 = 0
  !$omp target data map(sum1, sum2)
  !$omp target teams distribute reduction(+:sum1)
    do i=1,n
       sum1 = sum1 + f(i)
    end do
  !$omp target teams distribute map(sum1) reduction(+:sum2)
    do i=1,n
       sum2 = sum2 + g(i)*sum1
    end do
  !$omp end target data
  print *, "sum1 = " , sum1, ", sum2 = ", sum2
  !$OUTPUT: sum1 = 10100 , sum2 = 153015000
end program
```

S-14   sum1 += f(i);
S-15
S-16
S-17   #pragma omp target teams distribute map(sum1) reduction(+:sum2)
S-18   for (int i = 0; i < n; i++) {
S-19     sum2 += g(i) * sum1;
S-20   }
S-21
S-22   printf( "sum1 = %d, sum2 = %d\n", sum1, sum2);
S-23   //OUTPUT: sum1 = 9900, sum2 = 147015000
S-24   return 0;
S-25
S-26   int f(int res){ return res*2; }
S-27   int g(int res){ return res*3; }
```
10.9.4 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.

Example `target_task_reduction.1.c` (omp_5.2)

```c
#include <stdio.h>

#pragma omp declare target enter(device_compute)
void device_compute(int *);
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;
```

Fortran

```fortran
integer function f(res)
    integer :: res
    f = res*2
end function

integer function g(res)
    integer :: res
    g = res*3
end function
```

C / C++
Example target_task_reduction.1.f90 (omp_5.2)

program target_task_reduction_ex1
interface
   subroutine device_compute(res)
      !$omp declare target enter(device_compute)
      integer :: res
   end subroutine device_compute
   subroutine host_compute(res)
      integer :: res
   end subroutine host_compute
end interface
integer :: sum
sum = 0
 !$omp parallel masked
   !$omp taskgroup task_reduction(+:sum)
   !$omp taskgroup in_reduction(+:sum) nowait
   call device_compute(sum)
   !$omp end taskgroup
   print *, "sum = ", sum
   !!OUTPUT: sum = 2
end program

subroutine device_compute(sum)
   integer :: sum
   sum = 1
end subroutine
subroutine host_compute(sum)
   integer :: sum
   sum = 1
end subroutine
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**.

---

**Example target_task_reduction.2a.c (omp_5.2)**

```c
#include <stdio.h>

#pragma omp declare target enter(device_compute)
extern void device_compute(int *);

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);
  return 0;
}

void device_compute(int *sum){ *sum = 1; }

void host_compute(int *sum){ *sum = 1; }
```

---
Example target_task_reduction.2a.f90 (omp_5.2)

```fortran
program target_task_reduction_ex2
  interface
    subroutine device_compute(res)
      !$omp declare target enter(device_compute)
      integer :: res
    end subroutine device_compute
    subroutine host_compute(res)
      integer :: res
    end subroutine host_compute
  end interface
  integer :: sum
  sum = 0
  !$omp parallel sections reduction(task,+:sum)
    !$omp section
      !$omp target in_reduction(+:sum) nowait
      call device_compute(sum)
      !$omp end target
    !$omp section
      call host_compute(sum)
    !$omp end parallel sections
  print *, "sum = ", sum
end program
```

```fortran
subroutine device_compute(sum)
  integer :: sum
  sum = 1
end subroutine
```

```fortran
subroutine host_compute(sum)
  integer :: sum
  sum = 1
end subroutine
```

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.
Example target_task_reduction.2b.c (omp_5.2)

```c
#include <stdio.h>

#pragma omp declare target enter(device_compute)
extern void device_compute(int *);
extern void host_compute(int *);

int main()
{
  int sum = 0;

  #pragma omp parallel masked reduction(task, +:sum)
  {
    #pragma omp target in_reduction(+:sum) nowait
    device_compute(&sum);
    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; }
```

Example target_task_reduction.2b.f90 (omp_5.2)

```fortran
program target_task_reduction_ex2b
  interface
    subroutine device_compute(res)
      !$omp declare target enter(device_compute)
      integer :: res
    end subroutine device_compute
    subroutine host_compute(res)
      integer :: res
    end subroutine host_compute
  end interface

  integer :: sum
  sum = 0

  !$omp parallel masked reduction(task,+:sum)
  !$omp target in_reduction(+:sum) nowait
  call device_compute(sum)
  !$omp end target
```
10.9.5 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.

```
#include <stdio.h>

int array_sum(int n, int *v) {
    int i;
    int res = 0;
    #pragma omp taskloop reduction(+: res)
    for(i = 0; i < n; ++i)
```
int main(int argc, char *argv[]) {
    int n = 10;
    int v[10] = {1,2,3,4,5,6,7,8,9,10};
    #pragma omp parallel
    #pragma omp single
    {
        int res = array_sum(n, v);
        printf("The result is %d\n", res);
    }
    return 0;
}

Example taskloop_reduction.1.f90 (omp_5.0)

function array_sum(n, v) result(res)
    implicit none
    integer :: n, v(n), res
    integer :: i
    res = 0
    !$omp taskloop reduction(+: res)
    do i=1, n
        res = res + v(i)
    end do
    !$omp end taskloop
end function array_sum

program main
    implicit none
    integer :: n, v(10), res
    integer :: i
    integer, external :: array_sum
    n = 10
    do i=1, n
        v(i) = i
    end do
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.

---

Example `taskloop_reduction.2.c` (omp_5.0)

```c
#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 argc, char *argv[]) {
    ```
int n = 10;
int v[10] = {1,2,3,4,5,6,7,8,9,10};

#pragma omp parallel
#pragma omp single
{
    int res = array_sum(n, v);
    printf("The result is %d\n", res);
}

return 0;

---

Example taskloop_reduction.2.f90 (omp_5.0)

function array_sum(n, v) result(res)
  implicit none
  integer :: n, v(n), res
  integer :: i

  res = 0
  !$omp taskgroup task_reduction(+: res)
  if (n > 0) then
      !$omp task in_reduction(+: res)
      res = res + v(1)
      !$omp end task
  !$omp taskloop in_reduction(+: res) nogroup
  do i=2, n
      res = res + v(i)
  end do
  !$omp end taskloop
endif
  !$omp end taskgroup
end function array_sum

program main
  implicit none
  integer :: n, v(10), res
  integer :: i

  integer, external :: array_sum

  n = 10
  do i=1, n
v(i) = i
end do

!$omp parallel
!$omp single
res = array_sum(n, v)
print *, "The result is", res
!$omp end single
!$omp end parallel
end program main

In the OpenMP 5.0 Specification, \texttt{reduction} clauses for the \texttt{taskloop simd} construct were also added.

The examples below compare reductions for the \texttt{taskloop} and the \texttt{taskloop simd} constructs. These examples illustrate the use of \texttt{reduction} clauses within "stand-alone" \texttt{taskloop} constructs, and the use of \texttt{in_reduction} clauses for tasks of taskloops to participate with other reductions within the scope of a parallel region.

\textbf{taskloop reductions:}

In the \texttt{taskloop reductions} section of the example below, \texttt{taskloop 1} uses the \texttt{reduction} clause in a \texttt{taskloop} construct for a sum reduction, accumulated in \texttt{asum}. The behavior is as though a \texttt{taskgroup} construct encloses the taskloop region with a \texttt{task_reduction} clause, and each taskloop task has an \texttt{in_reduction} clause with the specifications of the \texttt{reduction} clause. At the end of the taskloop region \texttt{asum} contains the result of the reduction.

The next taskloop, \texttt{taskloop 2}, illustrates the use of the \texttt{in_reduction} clause to participate in a previously defined reduction scope of a \texttt{parallel} construct.

The task reductions of \texttt{task 2} and \texttt{taskloop 2} are combined across the \texttt{taskloop} construct and the single \texttt{task} construct, as specified in the \texttt{reduction(task, +:asum)} clause of the \texttt{parallel} construct. At the end of the parallel region \texttt{asum} contains the combined result of all reductions.

\textbf{taskloop simd reductions:}

Reductions for the \texttt{taskloop simd} construct are shown in the second half of the code. Since each component construct, \texttt{taskloop} and \texttt{simd}, can accept a reduction-type clause, the \texttt{taskloop simd} construct is a composite construct, and the specific application of the reduction clause is defined within the \texttt{taskloop simd} construct section of the OpenMP 5.0 Specification.

The code below illustrates use cases for these reductions.

In the \texttt{taskloop simd reduction} section of the example below, \texttt{taskloop simd 3} uses the \texttt{reduction} clause in a \texttt{taskloop simd} construct for a sum reduction within a loop. For this case a \texttt{reduction} clause is used, as one would use for a \texttt{simd} construct. The SIMD reductions of each task are combined, and the results of these tasks are further combined just as in the \texttt{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` and `taskloop simd` 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.

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

int main(){
  int i, a[N], asum=0;
  for(i=0;i<N;i++) a[i]=i;
  // taskloop reductions
  #pragma omp parallel masked
  #pragma omp taskloop reduction(+:asum) // taskloop 1
  for(i=0;i<N;i++) asum += a[i];
  #pragma omp parallel reduction(task, +:asum) // parallel reduction a
  {
    #pragma omp masked
    #pragma omp task in_reduction(+:asum) // task 2
    for(i=0;i<N;i++) asum += a[i];
    #pragma omp masked taskloop in_reduction(+:asum) // taskloop 2
    for(i=0;i<N;i++) asum += a[i];
  }
  // taskloop simd reductions
  #pragma omp parallel masked
  #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

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

!! taskloop 1
#$omp parallel masked
#$omp taskloop reduction(+:asum)                   !! taskloop 1
    do i=1,N;  asum = asum + a(i);  enddo
#$omp end taskloop
#$omp end parallel masked

!! parallel reduction a
#$omp parallel reduction(task, +:asum)            !! parallel reduction a
#$omp masked
#$omp task in_reduction(+:asum)                   !! task 2
    do i=1,N;  asum = asum + a(i);  enddo
#$omp end task
#$omp end masked

!! taskloop 2
#$omp masked taskloop in_reduction(+:asum)        !! taskloop 2
do i=1,N; asum = asum + a(i); enddo

!$omp end masked taskloop

!$omp end parallel

!! taskloop simd reductions

!$omp parallel masked
!$omp taskloop simd reduction(+:asum) !! taskloop simd 3
do i=1,N; asum = asum + a(i); enddo
!$omp end taskloop simd
!$omp end parallel masked

!$omp parallel reduction(task, +:asum) !! parallel reduction b

!$omp masked
!$omp task in_reduction(+:asum) !! task 4
do i=1,N; asum = asum + a(i); enddo
!$omp end task
!$omp end masked

!$omp masked taskloop simd in_reduction(+:asum) !! taskloop simd 4
do i=1,N; asum = asum + a(i); enddo
!$omp end masked taskloop simd

!$omp end parallel

print*,"asum=",asum !! output: asum=30300
end program

10.9.6 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.
Example scope_reduction.1.cpp (omp_5.1)

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

void do_work(int n, float a[], float &s)
{
    float loc_s = 0.0f; // local sum
    static int nthrs;
    #pragma omp for
    for (int i = 0; i < n; i++)
        loc_s += a[i];
    #pragma omp single
    {
        s = 0.0f; // total sum
        nthrs = 0;
    }
    #pragma omp scope reduction(+:s,nthrs)
    {
        s += loc_s;
        nthrs++;
    }
    #pragma omp masked
    printf("total sum = %f, nthrs = %d\n", s, nthrs);
}

float work(int n, float a[])
{
    float s;
    #pragma omp parallel
    {
        do_work(n, a, s);
    }
    return s;
}
```
10.9.7 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, followed by the combiner, here the functions `minproc` and `maxproc` 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
#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 : \  
    minproc(&omp_out, &omp_in)) \  
    initializer( omp_priv = { INT_MAX, INT_MAX } )

#pragma omp declare reduction(max : struct point : \  
    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;
```
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.

```
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(*)
  !$omp declare reduction(min : point : minproc(omp_out, omp_in)) &
  !$omp& initializer( omp_priv = point( HUGE(0), HUGE(0) ) )
  !$omp declare reduction(max : point : maxproc(omp_out, omp_in)) &
  !$omp& initializer( omp_priv = point(0, 0) )
  integer :: i
  type(point) :: minp = point(HUGE(0), HUGE(0)), maxp = point(0, 0)
  !$omp parallel do reduction(min:minp) reduction(max:maxp)
  do i = 1, n
    call minproc(minp, points(i))
    call maxproc(maxp, points(i))
  end do
  print *, "min = (", minp%x, minp%y, ")" 
```

---

C / C++

```
#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);
```
print *, "max = (", maxp%x, maxp%y, ")"
contains
subroutine minproc ( out, in )
  implicit none
  type(point), intent(inout) :: out
  type(point), intent(in) :: in
  out%x = min( out%x, in%x )
  out%y = min( out%y, in%y )
end subroutine minproc

subroutine maxproc ( out, in )
  implicit none
  type(point), intent(inout) :: out
  type(point), intent(in) :: in
  out%x = max( out%x, in%x )
  out%y = max( out%y, in%y )
end subroutine maxproc
end subroutine

The following example shows the same computation as \textit{udr.1} but it illustrates that you can craft complex expressions in the user-defined reduction declaration. In this case, instead of calling the \textit{minproc} and \textit{maxproc} functions we inline the code in a single expression.

\begin{verbatim}
#include <stdio.h>
#include <limits.h>

struct point {
  int x;
  int y;
};

#pragma omp declare reduction(min : struct point : \
  omp_out.x = omp_in.x > omp_out.x ? omp_out.x : omp_in.x, \
  omp_out.y = omp_in.y > omp_out.y ? omp_out.y : omp_in.y ) \
initializer( omp_priv = { INT_MAX, INT_MAX } )

#pragma omp declare reduction(max : struct point : \
  omp_out.x = omp_in.x < omp_out.x ? omp_out.x : omp_in.x, \
  omp_out.y = omp_in.y < omp_out.y ? omp_out.y : omp_in.y ) \
\end{verbatim}
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++ ) {
    if ( points[i].x < minp.x ) minp.x = points[i].x;
    if ( points[i].y < minp.y ) minp.y = points[i].y;
    if ( points[i].x > maxp.x ) maxp.x = points[i].x;
    if ( points[i].y > maxp.y ) maxp.y = points[i].y;
  }
  printf("min = (%d, %d)\n", minp.x, minp.y);
  printf("max = (%d, %d)\n", maxp.x, maxp.y);
}

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&   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 : &
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.
void mx_combine(struct mx_s *out, struct mx_s *in)  
{  
  if ( out->value < in->value ) {  
    out->value = in->value;  
    out->index = in->index;  
  }  
}  

void mx_init(struct mx_s *priv, struct mx_s *orig)  
{  
  priv->value = orig->value;  
  priv->index = orig->index;  
}  

int main(void)  
{  
  struct mx_s mx;  
  float val[N], d;  
  int i, count = N;  
  for (i = 0; i < count; i++) {  
    d = (N*0.8f - i);  
    val[i] = N * N - d * d;  
  }  
  mx.value = val[0];  
  mx.index = 0;  
  #pragma omp parallel for reduction(maxloc: mx)  
  for (i = 1; i < count; i++) {  
    if (mx.value < val[i])  
    {  
      mx.value = val[i];  
      mx.index = i;  
    }  
  }  
  printf("max value = %g, index = %d\n", mx.value, mx.index);  
  /* prints 10000, 80 */  
  return 0;  
}  

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.
Example udr.3.f90 (omp_4.0)

```
program max_loc
  implicit none

  type :: mx_s
    real value
    integer index
  end type

  !$omp declare reduction(maxloc: mx_s: &
    !$omp& mx_combine(omp_out, omp_in)) &
    !$omp& initializer(mx_init(ompPriv, omp_orig))

  integer, parameter :: N = 100
  type(mx_s) :: mx
  real :: val(N), d
  integer :: i, count

  count = N
  do i = 1, count
    d = N*0.8 - i + 1
    val(i) = N * N - d * d
  enddo

  mx%value = val(1)
  mx%index = 1
  !$omp parallel do reduction(maxloc: mx)
    do i = 2, count
      if (mx%value < val(i)) then
        mx%value = val(i)
        mx%index = i
      endif
    enddo

  print *, 'max value = ', mx%value, ' index = ', mx%index
  ! prints 10000, 81
contains

  subroutine mx_combine(out, in)
    implicit none
    type(mx_s), intent(inout) :: out
    type(mx_s), intent(in) :: in

    if ( out%value < in%value ) then
```

CHAPTER 10. DATA ENVIRONMENT

421
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.

Example udr.4.f90 (omp_4.0)

```fortran
module data_red
!
  Declare data type.
  type dt
    real :: r1
    real :: r2
  end type
!
  Declare the user-defined operator .add.
  interface operator(.add.)
    module procedure addc
  end interface
!
  Declare the user-defined reduction operator .add.
  !$omp declare reduction(.add.:dt:omp_out=omp_out.add.omp_in) &
```
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.
The following examples show 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.
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 $k$.

---

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

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

int main(void) {
    int a[N], b[N];
    int x = 0;

    // initialization
    for (int k = 0; k < N; k++)
        a[k] = k + 1;

    // a[k] is included in the computation of producing results in b[k]
    #pragma omp parallel for simd reduction(inscan,+: x)
    for (int k = 0; k < N; k++) {
        x += a[k];
        #pragma omp scan inclusive(x)
        b[k] = x;
    }

    printf("x = %d, b[0:3] = %d %d %d\n", x, b[0], b[1], b[2]);
    // 5050, 1 3 6

    return 0;
```
Example scan.1.f90 (omp_5.0)

```fortran
program inclusive_scan
  implicit none
  integer, parameter :: n = 100
  integer a(n), b(n)
  integer x, k

  ! initialization
  x = 0
  do k = 1, n
    a(k) = k
  end do

  ! a(k) is included in the computation of producing results in b(k)
  !$omp parallel do simd reduction(inscan,+: x)
  do k = 1, n
    x = x + a(k)
    !$omp scan inclusive(x)
    b(k) = x
  end do

  print *, 'x =', x, ', b(1:3) =', b(1:3)
  ! 5050, 1 3 6
end program
```

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`.

Example scan.2.c (omp_5.0)

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

int main(void)
{
  int a[N], b[N];
  int x = 0;
  // initialization
```
for (int k = 0; k < N; k++)
a[k] = k + 1;

// a[k] is not included in the computation of producing
// results in b[k]
#pragma omp parallel for simd reduction(inscan,+: x)
for (int k = 0; k < N; k++) {
    b[k] = x;
    #pragma omp scan exclusive(x)
    x += a[k];
}

printf("x = %d, b[0:3] = %d %d %d\n", x, b[0], b[1], b[2]);
// 5050, 0 1 3
return 0;

Example scan.2.f90 (omp_5.0)

program exclusive_scan
    implicit none
    integer, parameter :: n = 100
    integer a(n), b(n)
    integer x, k

    ! initialization
    x = 0
    do k = 1, n
    a(k) = k
    end do

    ! a(k) is not included in the computation of producing results in b(k)
    !$omp parallel do simd reduction(inscan,+: x)
    do k = 1, n
        b(k) = x
        !$omp scan exclusive(x)
        x = x + a(k)
    end do

    print *, 'x =', x, ', b(1:3) =', b(1:3)
    ! 5050, 0 1 3
end program
10.11 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.

```
#include <stdlib.h>

float* work;
int size;
float tol;

#pragma omp threadprivate(work,size,tol)

void build()
{
    int i;
    work = (float*)malloc( sizeof(float)*size );
    for( i = 0; i < size; ++i ) work[i] = tol;
}

void copyin_example( float t, int n )
{
    tol = t;
    size = n;
    #pragma omp parallel copyin(tol,size)
    {
        build();
    }
}
```

C / C++
Example copyin.1.f

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

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

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

---

**Example copyprivate.1.c**

```c
#include <stdio.h>
float x, y;
#pragma omp threadprivate(x, y)

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

---

**Example copyprivate.1.f**

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

---

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.
1

Example copyprivate.2.c (omp_5.1)

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

float read_next() {
    float * tmp;
    float return_val;

    #pragma omp single copyprivate(tmp)
    {
        tmp = (float *) malloc(sizeof(float));
    } /* copies the pointer only */

    #pragma omp masked
    {
        scanf("%f", tmp);
    }

    #pragma omp barrier
    return_val = *tmp;

    #pragma omp barrier
    #pragma omp single nowait
    {
        free(tmp);
    }

    return return_val;
}
```

2

Example copyprivate.2.f (omp_5.1)

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

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

!$OMP MASKED
READ (11) TMP
!$OMP END MASKED
```
Suppose that the number of lock variables required within a parallel region cannot easily be determined prior to entering it. The copyprivate clause can be used to provide access to shared lock variables that are allocated within that parallel region.

Example copyprivate.3.c

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

omp_lock_t *new_lock()
{
    omp_lock_t *lock_ptr;

    #pragma omp single copyprivate(lock_ptr)
    {
        lock_ptr = (omp_lock_t *) malloc(sizeof(omp_lock_t));
        omp_init_lock( lock_ptr );
    }

    return lock_ptr;
}
```
Example copyprivate.3.f

```fortran
FUNCTION NEW_LOCK()
    USE OMP_LIB ! or INCLUDE "omp_lib.h"
    INTEGER(OMP_LOCK_KIND), POINTER :: NEW_LOCK

    !$OMP SINGLE
    ALLOCATE(NEW_LOCK)
    CALL OMP_INIT_LOCK(NEW_LOCK)
    !$OMP END SINGLE COPYPRIVATE(NEW_LOCK)
END FUNCTION NEW_LOCK
```

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

Example copyprivate.4.f

```fortran
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
```

CHAPTER 10. DATA ENVIRONMENT  433
10.13 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)

```c++
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; // no matter if the argument is determined private 
        gen_task (y); // or shared in the enclosing context. 
        gen_task (z); // or shared in the enclosing context. 
        y++; // each thread has its own int object y refers to 
    
    } 
}
```

434  OpenMP Examples Version 5.2 - April 2022
The following is an invalid example of specifying an associate name on a data-sharing attribute clause. The constraint in the Data Sharing Attribute Rules section in the OpenMP 4.0 API Specifications states that an associate name preserves the association with the selector established at the **ASSOCIATE** statement. The associate name \( b \) is associated with the shared variable \( a \). With the predetermined data-sharing attribute rule, the associate name \( b \) is not allowed to be specified on the **private** clause.

**Example associate.1.f** (omp_4.0)

```fortran
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)

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

The following example illustrates the effect of specifying a selector name on a data-sharing attribute clause. The associate name \( u \) is associated with \( v \) and the variable \( v \) is specified on the **private** clause of the **parallel** construct. The construct association is established prior to the **parallel** region. The association between \( u \) and the original \( v \) is retained (see the Data Sharing Attribute Rules section in the OpenMP 4.0 API Specifications). Inside the **parallel** region, \( v \) has the value of -1 and \( u \) has the value of the original \( v \).
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.3.f90 (omp_4.0)**

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

---

**Example associate.4.f90 (omp_5.1)**

```fortran
program main
  integer :: scalr, aray(3)
  scalr = -1 ; aray = -1
  associate(a_scalr=>scalr, a_aray=>aray)
  !$omp target    !! TARGET 1
  aray = [1,2,3]
  !$omp end target
  print *, a_aray, aray    !! 1 2 3 1 2 3
```

---

OpenMP Examples Version 5.2 - April 2022
!$omp target !! TARGET 2
a_aray = [4,5,6]
(!$omp end target
print *, a_aray, aray !! 4 5 6 4 5 6

!!!$omp target !! TARGET 3
!! !! mapping, in this case implicit,
!! !! !of aray AND a_aray NOT ALLOWED
!! a_aray = [4,5,6]
!! aray = [4,5,6]
!!!$omp end target

!$omp target !! TARGET 4
scalr = 1 !! scalr is firstprivate
(!$omp end target
print *, a_scalr, scalr !! -1 -1

!$omp target !! TARGET 5
!! a_scalr implicitly mapped
a_scalr = 2
(!$omp end target
print *, a_scalr, scalr !! 2 2

!!$omp target !! TARGET 6
scalr = 3 !! scalr is firstprivate
print *, a_scalr, scalr !! 2 3
a_scalr = 4 !! a_scalr implicitly mapped
print *, a_scalr, scalr !! 4 3
(!$omp end target
print *, a_scalr, scalr !! 4 4

!!!$omp target map(a_scalr,scalr) !! TARGET 7
!! mapping, in this case explicit,
!! !of scalr AND a_scalr NOT ALLOWED
!! scalr = 5
!! a_scalr = 5
!!!$omp end target

end associate
end program
This page intentionally left blank
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 \texttt{xval} could be either 2 or 5, depending on the timing of the threads. The \texttt{atomic} directives are necessary for the accesses to \texttt{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 \texttt{xval}. Otherwise, thread 1 is guaranteed to see 2 in \texttt{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 \texttt{x}, they may concurrently access \texttt{x} without requiring \texttt{atomic} directives to avoid a data race.

---

**Example mem_model.1.c (omp_3.1)**

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

int main(){
  int x;
  x = 2;
  #pragma omp parallel num_threads(2) shared(x)
  {
    if (omp_get_thread_num() == 0) {
      #pragma omp atomic write
      x = 5;
    } else {
      int xval;
      #pragma omp atomic read
      xval = x;
      /* Print 1: xval can be 2 or 5 */
      printf("1: Thread# %d: x = %d\n", omp_get_thread_num(), xval);
    }
    /* Print 2 */
  }
  #pragma omp barrier
  if (omp_get_thread_num() == 0) {
    /* Print 2 */
  }
```
Example mem_model.1.f90 (omp_3.1)

```fortran
PROGRAM MEMMODEL
  INCLUDE "omp_lib.h" ! or USE OMP_LIB
  INTEGER X, XVAL

  X = 2
  !$OMP PARALLEL NUM_THREADS(2) SHARED(X)
  IF (OMP_GET_THREAD_NUM() .EQ. 0) THEN
    !$OMP ATOMIC WRITE
    X = 5
  ELSE
    !$OMP ATOMIC READ
    XVAL = X
  ENDIF
  ! PRINT 1: XVAL can be 2 or 5
  PRINT *, "1: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", XVAL
ENDIF

  !$OMP BARRIER

  IF (OMP_GET_THREAD_NUM() .EQ. 0) THEN
    ! PRINT 2
    PRINT *, "2: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
  ELSE
    ! PRINT 3
    PRINT *, "3: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
  ENDIF

  !$OMP END PARALLEL
END PROGRAM MEMMODEL
```
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
#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);
        }
    }
}
```
Example mem_model.2.f (omp_3.1)

PROGRAM EXAMPLE
    INCLUDE "omp_lib.h" ! or USE OMP_LIB
    INTEGER DATA
    INTEGER FLAG, FLAG_VAL

    FLAG = 0

    !$OMP PARALLEL NUM_THREADS(2)
    IF(OMP_GET_THREAD_NUM() .EQ. 0) THEN
        ! Write to the data buffer that will be read by thread 1
        DATA = 42
    ENDIF
    ! Flush DATA to thread 1 and strictly order the write to DATA
    ! relative to the write to the FLAG
    !$OMP FLUSH(FLAG, DATA)
    ! Set FLAG to release thread 1
    !$OMP ATOMIC WRITE
    FLAG = 1
    ENDIF
    !$OMP END PARALLEL
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
#include <omp.h>
#include <stdio.h>

int data0 = 0, data1 = 0;

int main()
{
    int flag=0;
    
    #pragma omp parallel num_threads(3)
    {
        if(omp_get_thread_num()==0)
        {
            data0 = 17;
            #pragma omp flush
            /* Set flag to release thread 1 */
            #pragma omp atomic update
            flag++;
            /* Flush of flag is implied by the atomic directive */
        }
        else if(omp_get_thread_num()==1)
        {
            int flag_val = 0;
            /* Loop until we see that flag reaches 1*/
            while(flag_val < 0)
            {
                #pragma omp atomic read
                flag_val = flag;
            }
            #pragma omp flush(data0)
            /* data0 is 17 here */
            printf("Thread 1 awoken (data0 = %d)\n", data0);
            data1 = 42;
            #pragma omp flush(data1)
            /* Set flag to release thread 2 */
            #pragma omp atomic update
        }
    }
    printf("Thread 1 awoken (data0 = %d)\n", data0);
    data1 = 42;
    #pragma omp flush(data1)
    /* Set flag to release thread 2 */
    #pragma omp atomic update
}
flag++;
/* Flush of flag is implied by the atomic directive */
}
else if(omp_get_thread_num()==2)
{
    int flag_val = 0;
    /* Loop until we see that flag reaches 2 */
    while(flag_val < 2)
    {
        #pragma omp atomic read
        flag_val = flag;
    
        #pragma omp flush(data0,data1)
    /* there is a data race here;
    data0 is 17 and data1 is undefined */
        printf("Thread 2 awoken (data0 = %d, data1 = %d)\n",
        data0, data1);
    
}
}
return 0;

Example mem_model.3.f (omp_3.1)

PROGRAM EXAMPLE
INCLUDE "omp_lib.h" ! or USE OMP_LIB
INTEGER FLAG, FLAG_VAL
INTEGER DATA0, DATA1
FLAG = 0
!$OMP PARALLEL NUM_THREADS(3)
IF(OMP_GET_THREAD_NUM() .EQ. 0) THEN
    DATA0 = 17
!$OMP FLUSH
! Set flag to release thread 1
!$OMP ATOMIC UPDATE
    FLAG = FLAG + 1
! Flush of FLAG is implied by the atomic directive
ELSE IF(OMP_GET_THREAD_NUM() .EQ. 1) THEN
    ! Loop until we see that FLAG reaches 1
    !$OMP FLUSH(FLAG, DATA)
    FLAG_VAL = 0
    DO WHILE(FLAG_VAL .LT. 1)
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.

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 */
        }
    }
}
```
Example mem_model.4a.f90 (omp_3.1)

1 subroutine flush_incorrect
2 use omp_lib
3 implicit none
4 integer a, b, tmp
5 integer myid
6 a = 0; b = 0
7 !$omp parallel private(myid,tmp) num_threads(2)
8 myid = omp_get_thread_num()
9 !$omp atomic write
10 b = 1
11 !$omp flush(b) ! flushes are not ordered
12 !$omp flush(a) ! compiler may move them around
13 !$omp atomic read
14 tmp = a
15 else ! thread 1
16 !$omp atomic write
17 a = 1
18 !$omp flush(a) ! flushes are not ordered
19 !$omp flush(b) ! compiler may move them around
20 !$omp atomic read
21 tmp = b
22 endif
23 if ( tmp == 0 ) then ! exclusive access not guaranteed
24 !! protected section
25 endif
26 !$omp end parallel
27 end subroutine

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.
Example mem_model.4b.c (omp_3.1)

```c
#include <omp.h>

void flush_correct()
{
    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(a,b) // flushes are ordered
            #pragma omp atomic read
            tmp = a;
        }
        else { // thread 1
            #pragma omp atomic write
            a = 1;
            #pragma omp flush(a,b) // flushes are ordered
            #pragma omp atomic read
            tmp = b;
        }

        if ( tmp == 0 ) { // access by single thread
            /* protected section */
        }
    }
}
```

Example mem_model.4b.f90 (omp_3.1)

```fortran
subroutine flush_correct
use omp_lib
implicit none
integer a, b, tmp
integer myid

a = 0; b = 0
!$omp parallel private(myid,tmp) num_threads(2)
myid = omp_get_thread_num()
```
if ( myid == 0 ) then ! thread 0
    !$omp atomic write
    b = 1
    !$omp flush(a,b) ! flushes are ordered
    !$omp atomic read
    tmp = a
  else ! thread 1
    !$omp atomic write
    a = 1
    !$omp flush(a,b) ! flushes are ordered
    !$omp atomic read
    tmp = b
  endif
if ( tmp == 0 ) then ! access by single thread
  ! protected section
endif
 !$omp end parallel
end subroutine
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_alloc()` 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.

---

Example allocators.1.c (omp_5.0)

```c
#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;

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. 300.
dcall(x,y)
call omp_destroy_allocator(xy_alloc)
end program

---

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_allocr 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.
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.
Example allocators.3.f90 (omp_5.2)

```fortran
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
```

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.
Example allocators.4.c (omp_5.1)

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

void my_init(double *, double *, int, double *, double *, int, 
              double *, double *, int);

void lib_saxpy(double *, double *, double, int);

void my_gather(double *, double *, int);

#pragma omp begin declare target

void my_gpu_vxv(double *, double *, int);

#pragma omp end declare target

#define Nhb 1024*1024 // high bandwith
#define Nbg 1024*1024*64 // big memory, default
#define Nll 1024*1024 // low latency memory

void test_allocate() {

    double v1[Nhb], v2[Nhb];
    double v3[Nbg], v4[Nbg];
    double v5[Nll], v6[Nll];

    /*** CASE 1: USING ALLOCATE DIRECTIVE ***/
    #pragma omp allocate(v1,v2) allocator(omp_high_bw_mem_alloc)
    #pragma omp allocate(v3,v4) allocator(omp_default_mem_alloc)
    my_init(v1,v2,Nhb, v3,v4,Nbg, v5,v6,Nll);
    lib_saxpy(v1,v2,5.0,Nhb);
    #pragma omp target map(to: v3[0:Nbg], v4[0:Nbg]) map(from:v3[0:Nbg])
    my_gpu_vxv(v3,v4,Nbg);

    /*** CASE 2: USING ALLOCATE CLAUSE ***/
    #pragma omp task private(v5,v6) \\ 
    allocate(allocator(omp_low_lat_mem_alloc): v5,v6)
    {
    my_gather(v5,v6,Nll);
    }
}
```
subroutine test_allocate
  use omp_lib
  interface
    subroutine my_gpu_vxv(va,vb,n)
      !$omp declare target
      integer :: n
      double precision :: va(n), vb(n)
    end subroutine
  end interface
  integer,parameter :: Nhb=1024*1024, & !! high bandwidth
  Nbg=1024*1024*64,& !! big memory, default
  Nll=1024*1024 !! low latency memory
  double precision :: v1(Nhb), v2(Nhb)
  double precision :: v3(Nbg), v4(Nbg)
  double precision :: v5(Nll), v6(Nll)

!*** CASE 1: USING ALLOCATE DIRECTIVE ***!
  !$omp allocate(v1,v2) allocator(omp_high_bw_mem_alloc)
  !$omp allocate(v3,v4) allocator(omp_default_mem_alloc)
  call my_init(v1,v2,Nhb, v3,v4,Nbg, v5,v6,Nll)
  call lib_saxpy(v1,v2,5.0,Nhb)
  !$omp target map(to: v3, v4) map(from:v3)
  call my_gpu_vxv(v3,v4,Nbg)
  !$omp end target

!*** CASE 2: USING ALLOCATE CLAUSE ***!
  !$omp task private(v5,v6) &
  !$omp& allocate(allocator(omp_low_lat_mem_alloc): v5,v6)
  call my_gather(v5,v6,Nll)
  !$omp end task
end subroutine test_allocate
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

```fortran
SUBROUTINE SHARED_RACE
  INCLUDE "omp_lib.h" ! or USE OMP_LIB
  REAL A(20)
  INTEGER MYTHREAD
  !$OMP PARALLEL SHARED(A) PRIVATE(MYTHREAD)
  MYTHREAD = OMP_GET_THREAD_NUM()
  IF (MYTHREAD .EQ. 0) THEN
    CALL SUB(A(1:10)) ! compiler may introduce writes to A(6:10)
  ELSE
    A(6:10) = 12
  ENDIF
  !$OMP END PARALLEL
END SUBROUTINE SHARED_RACE

SUBROUTINE SUB(X)
  REAL X(*)
  X(1:5) = 4
END SUBROUTINE SUB
```
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_get_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 Conditional Compilation

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

*Example cond_comp.1.c*

```c
#include <stdio.h>

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

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

*Example cond_comp.1.f*

```fortran
PROGRAM EXAMPLE
C234567890
!$ PRINT *, "Compiled by an OpenMP-compliant implementation."
END PROGRAM EXAMPLE
```
12.2 Internal Control Variables (ICVs)

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

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

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

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

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

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

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

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

The print statement in an inner `parallel` region is also executed by only one of the threads in the team. Since we have a total of two inner `parallel` regions, the print statement will be executed twice – once per inner `parallel` region.
Example icv.1.c

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

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

CHAPTER 12. PROGRAM CONTROL

463
Example icv.1.f

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
12.3 Placement of \texttt{flush, barrier, taskwait} and \texttt{taskyield} Directives

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

\begin{verbatim}
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    if (a != 0)
S-20       #pragma omp taskwait
S-21     /* incorrect as taskwait cannot be immediate substatement
S-22     of if statement */
S-23
S-24 }
\end{verbatim}
The following example is non-conforming, because the \texttt{flush}, \texttt{barrier}, \texttt{taskwait}, and \texttt{taskyield} directives are stand-alone directives and cannot be the action statement of an \texttt{if} statement or a labeled branch target.

\begin{verbatim}
Example standalone.1.f90 (omp_3.1)

SUBROUTINE STANDALONE_WRONG()

    INTEGER A
    A = 1

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

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

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

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

    GOTO 100

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

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

    ! the TASKWAIT directive must not be a labeled branch target
    ! statement
    300 !$OMP TASKWAIT
    GOTO 400
\end{verbatim}
! the TASKYIELD directive must not be a labeled branch target

END SUBROUTINE

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

Example standalone.2.c (omp_3.1)

```c
void standalone_ok()
{
    int a = 1;
    #pragma omp parallel
    {
        if (a != 0) {
            #pragma omp flush(a)
        }
        if (a != 0) {
            #pragma omp barrier
        }
        if (a != 0) {
            #pragma omp taskwait
        }
        if (a != 0) {
            #pragma omp taskyield
        }
    }
}
```
The following example is conforming because the `flush`, `barrier`, `taskwait`, and `taskyield` directives are enclosed in an `if` construct or follow the labeled branch target.

Example standalone.2.f90 (omp_3.1)

```
SUBROUTINE STANDALONE_OK()
   INTEGER A
   A = 1
   IF (A .NE. 0) THEN
     !$OMP FLUSH(A)
   ENDIF
   IF (A .NE. 0) THEN
     !$OMP BARRIER
   ENDIF
   IF (A .NE. 0) THEN
     !$OMP TASKWAIT
   ENDIF
   IF (A .NE. 0) THEN
     !$OMP TASKYIELD
   ENDIF
   GOTO 100
 100 CONTINUE
   !$OMP FLUSH(A)
   GOTO 200
 200 CONTINUE
   !$OMP BARRIER
   GOTO 300
 300 CONTINUE
   !$OMP TASKWAIT
   GOTO 400
 400 CONTINUE
   !$OMP TASKYIELD
END SUBROUTINE
```
12.4 Cancellation Constructs

The following example shows how the `cancel` directive can be used to terminate an OpenMP region. Although the `cancel` construct terminates the OpenMP worksharing region, programmers must still track the exception through the pointer `ex` and issue a cancellation for the `parallel` region if an exception has been raised. The 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 <cstdlib>

#define N 10000

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

void example() {
    std::exception *ex = NULL;
    #pragma omp parallel shared(ex)
    {
        #pragma omp for
        for (int i = 0; i < N; i++) {
            // no 'if' that prevents compiler optimizations
            try {
                causes_an_exception();
            } catch (std::exception *e) {
                // still must remember exception for later handling
                #pragma omp atomic write
                ex = e;
                // cancel worksharing construct
                #pragma omp cancel for
            }
            // if an exception has been raised, cancel parallel region
            if (ex) {
                #pragma omp cancel parallel
            }
            phase_1();
        }
    }
    #pragma omp barrier
```
The following example illustrates the use of the `cancel` construct in error handling. If there is an error condition from the `allocate` statement, the cancellation is activated. The encountering thread sets the shared variable `err` and other threads of the binding thread set proceed to the end of the worksharing construct after the cancellation has been activated.

```
Example cancellation.1.f90 (omp_4.0)

subroutine example(n, dim)
  integer, intent(in) :: n, dim(n)
  integer :: i, s, err
  real, allocatable :: B(:)
  err = 0
  !$omp parallel shared(err)
  ! ...
  !$omp do private(s, B)
  do i=1, n
    !$omp cancellation point do
    allocate(B(dim(i)), stat=s)
    if (s .gt. 0) then
      !$omp atomic write
      err = s
      !$omp cancel do
      endif
    ! ...
    !$omp deallocate private array B
    if (allocated(B)) then
      deallocate(B)
    endif
  enddo
  !$omp end parallel
end subroutine
```
The following example shows how to cancel a parallel search on a binary tree as soon as the search value has been detected. The code creates a task to descend into the child nodes of the current tree node. If the search value has been found, the code remembers the tree node with the found value through an atomic write to the result variable and then cancels execution of all search tasks. The function `search_tree_parallel` groups all search tasks into a single task group to control the effect of the cancel taskgroup directive. The `level` argument is used to create undefined tasks after the first ten levels of the tree.

```c
#include <stddef.h>

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

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

Example cancellation.2.c (omp_5.1)
binary_tree_t *search_tree_parallel(binary_tree_t *tree, int value) {
    binary_tree_t *found = NULL;
    #pragma omp parallel shared(found, tree, value)
    {
        #pragma omp masked
        {
            #pragma omp taskgroup
            {
                found = search_tree(tree, value, 0);
            }
        }
    }
    return found;
}

The following is the equivalent parallel search example in Fortran.

module parallel_search
    type binary_tree
        integer :: value
        type(binary_tree), pointer :: right
        type(binary_tree), pointer :: left
    end type
contains
    recursive subroutine search_tree(tree, value, level, found)
        type(binary_tree), intent(in), pointer :: tree
        integer, intent(in) :: value, level
        type(binary_tree), pointer :: found
        type(binary_tree), pointer :: found_left => NULL(), &
        found_right => NULL()
    if (associated(tree)) then
        if (tree%value .eq. value) then
            found => tree
        else
            !$omp task shared(found) if(level<10)
            call search_tree(tree%left, value, level+1, found_left)
            if (associated(found_left)) then
                !$omp critical
            end if
        end if
    end if
end subroutine search_tree
end module parallel_search
found => found_left

!$omp end critical

!$omp cancel taskgroup
endif

!$omp end task

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

!$omp critical
found => found_right
!$omp end critical
endif

!$omp cancel taskgroup
endif
!$omp end task

!$omp taskwait
endif
end subroutine

end subroutine search_tree_parallel(tree, value, found)

found => NULL()

!$omp parallel shared(found, tree, value)
!$omp masked
!$omp taskgroup
call search_tree(tree, value, 0, found)
!$omp end taskgroup
!$omp end masked
!$omp end parallel
end subroutine

end module parallel_search
12.5 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.

```
#include <iostream>
using namespace std;

#pragma omp requires unified_shared_memory

typedef struct mypoints
{
 double res;
 double data[500];
} mypoints_t;

void do_something_with_p(mypoints_t *p, int q);

int main()
{
 mypoints_t p;
 int q=0;

#pragma omp target // no map clauses needed
{
 q++;  
 do_something_with_p(&p,q);
}

 cout<< p.res << " " << q << endl;  // output 1 0
 return 0;

 void do_something_with_p(mypoints_t *p, int q)
{

```
module data

!$omp requires unified_shared_memory

  type, public :: mypoints
    double precision :: res
    double precision :: data(500)
  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

  type(mypoints) :: p
  integer :: q

  p%res = q;
  do i=1,size(p%data)
    p%data(i)=q*i
  enddo

end subroutine
12.6 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 call site. 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. The scoring algorithm can be found in the OpenMP 5.0 Specification.

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.)

---

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

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

void p_vxv(int *v1, int *v2, int *v3, int n);
void t_vxv(int *v1, int *v2, int *v3, int n);

#pragma omp declare variant( p_vxv ) match( construct={parallel} )
#pragma omp declare variant( t_vxv ) match( construct={target} )
void vxv(int *v1, int *v2, int *v3, int n) // base function
{
    for (int i= 0; i< n; i++) v3[i] = v1[i] * v2[i];
}
void p_vxv(int *v1, int *v2, int *v3, int n) // function variant
```
{ #pragma omp for
  for (int i= 0; i< n; i++) v3[i] = v1[i] * v2[i]*3;
}

#pragma omp begin declare target

void t_vxv(int *v1,int *v2,int *v3,int n)  // function variant
{
  #pragma omp distribute simd
  for (int i= 0; i< n; i++) v3[i] = v1[i] * v2[i]*2;
}
#pragma omp end declare target

int main()
{
  int v1[N], v2[N], v3[N];
  for(int i=0; i<N; i++){ v1[i]=(i+1); v2[i]=-(i+1); v3[i]=0; }  //init

  #pragma omp parallel
  {
    vxv(v1,v2,v3,N);
  }
  printf(" %d %d
",v3[0],v3[N-1]);  //from p_vxv -- output: -3 -30000

  #pragma omp target teams map(to: v1[:N],v2[:N]) map(from: v3[:N])
  {
    vxv(v1,v2,v3,N);
  }
  printf(" %d %d
",v3[0],v3[N-1]);  //from t_vxv -- output: -2 -20000

  vxv(v1,v2,v3,N);
  printf(" %d %d
",v3[0],v3[N-1]);  //from vxv -- output: -1 -10000

  return 0;
}

!! base function

integer,intent(in) :: v1(:),v2(:)
integer,intent(out) :: v3(:)
integer :: i, n

!$omp declare variant( p_vxv ) match( construct={parallel} )
!$omp declare variant( t_vxv ) match( construct={target} )

n = size(v1)
do i = 1, n; v3(i) = v1(i) * v2(i); enddo
end subroutine

end subroutine p_vxv(v1, v2, v3) !!! function variant
integer, intent(in) :: v1(:), v2(:)
integer, intent(out) :: v3(:)
integer :: i, n
n = size(v1)

!$omp do
do i = 1, n; v3(i) = v1(i) * v2(i) * 3; enddo
end subroutine

end subroutine t_vxv(v1, v2, v3) !!! function variant
integer, intent(in) :: v1(:), v2(:)
integer, intent(out) :: v3(:)
integer :: i, n

!$omp declare target
n = size(v1)

!$omp distribute simd
do i = 1, n; v3(i) = v1(i) * v2(i) * 2; enddo
end subroutine

end module subs

program main
use omp_lib
use subs
integer, parameter :: N = 100
integer :: v1(N), v2(N), v3(N)
do i = 1, N; v1(i) = i; v2(i) = -i; v3(i) = 0; enddo !!! init
!$omp parallel
call vxv(v1, v2, v3)
!$omp end parallel
print *, v3(1), v3(N) !!! from p_vxv -- output: -3 -30000
In this example, traits from the device set are used to select a function variant. In the declare variant directive, an isa 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 (for Xeon or Xeon Phi architectures). 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)

```c
#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];
}
```

---

**Fortran**

```fortran
!$omp target teams map(to: v1,v2) map(from: v3)
call vxv(v1,v2,v3)
!$omp end target teams
print *, v3(1),v3(N) !! from t_vxv -- output: -2 -20000

call vxv(v1,v2,v3)
print *, v3(1),v3(N) !! from vxv -- output: -1 -10000

end program
```
// 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],\mn y[N-1]: %5.0f %5.0f\n", y[0], y[N-1]);
  // output: y[0],y[N-1]: 3 3000
  return 0;
}

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\times+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 -------------------------------
12.7 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 "selectable" directive is called a **directive variant**. Traits are grouped by *construct*, *implementation* and *device sets* to be used by a selector of the same name.

In the first example the architecture trait *arch* 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 *device* as used 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*.

```
#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;
}
```

---
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.
Example metadirective.2.f90 (omp_5.2)

```fortran
program main
  use omp_lib
  implicit none
  integer, parameter :: N=1000
  external :: work_on_chunk
  integer :: i, idev

  do idev=0,omp_get_num_devices()-1
    !$omp target device(idev)
    !$omp begin metadirective
    !$omp& when( implementation={vendor(nvidia)}, &
              device={arch("kepler")}): &
    !$omp& teams num_teams(512) thread_limit(32) ) &
    !$omp& when( implementation={vendor(amd)}, &
              device={arch("fiji")}): &
    !$omp& teams num_teams(512) thread_limit(64) ) &
    !$omp& otherwise( teams )
    !$omp& when( implementation={vendor(amd)}, &
              device={arch("fiji")}): &
    !$omp& teams num_teams(512) thread_limit(64) ) &
    !$omp& otherwise( teams )
    !$omp& when( implementation={vendor(amd)}, &
              device={arch("fiji")}): &
    !$omp& teams num_teams(512) thread_limit(64) ) &
    !$omp& otherwise( teams )
    !$omp& when( implementation={vendor(nvidia)}, &
              device={arch("kepler")}): &
    !$omp& teams num_teams(512) thread_limit(32) ) &
    !$omp& otherwise( teams )
    !$omp& when( implementation={vendor(amd)}, &
              device={arch("fiji")}): &
    !$omp& teams num_teams(512) thread_limit(64) ) &
    !$omp& otherwise( teams )
    !$omp& when( implementation={vendor(nvidia)}, &
              device={arch("kepler")}): &
    !$omp& teams num_teams(512) thread_limit(32) ) &
    !$omp& otherwise( teams )
    !$omp& when( implementation={vendor(amd)}, &
              device={arch("fiji")}): &
    !$omp& teams num_teams(512) thread_limit(64) ) &
    !$omp& otherwise( teams )

    !$omp& otherwise( teams )
    !$omp& when( implementation={vendor(nvidia)}, &
              device={arch("kepler")}): &
    !$omp& teams num_teams(512) thread_limit(32) ) &
    !$omp& when( implementation={vendor(amd)}, &
              device={arch("fiji")}): &
    !$omp& teams num_teams(512) thread_limit(64) ) &
    !$omp& otherwise( teams )

    do i=1,N
      call work_on_chunk(idev,i)
    end do
  end do
end program
```

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)

```c
#include <stdio.h>
#include <math.h>
define N 1000

#pragma omp begin declare target
void exp_pi_diff(double *d, double my_pi){
    #pragma omp metadirective \
    when( construct={target}: distribute parallel for ) \
    otherwise( parallel for simd )
    for(int i = 0; i<N; i++) d[i] = exp( (M_PI-my_pi)*i );
}
#pragma omp end declare target

int main()
{
    //Calculates sequence of exponentials: (M_PI-my_pi) * index
    //M_PI is from math.h, and my_pi is user provided.
    double d[N];
    double my_pi=3.14159265358979e0;
```
Example metadirective.3.f90 (omp_5.2)

module params
  integer, parameter :: N=1000
  DOUBLE PRECISION, PARAMETER::M_PI=4.0d0*DATAN(1.0d0)
end module

subroutine exp_pi_diff(d, my_pi)
  use params
  implicit none
  integer :: i
  double precision :: d(N), my_pi
  !$omp declare target
  !$omp metadirective &
  !$omp& when( construct={target}: distribute parallel do ) &
  !$omp& otherwise( parallel do simd )
  do i = 1, size(d)
    d(i) = exp( (M_PI-my_pi)*i )
  end do
end subroutine

program main
  ! Calculates sequence of exponentials: (M_PI-my_pi) * index
  ! M_PI is from usual way, and my_pi is user provided.
  ! Fortran Standard does not provide PI
  use params
  implicit none
  double precision :: d(N)
  double precision :: my_pi=3.14159265358979d0
The user selector set can be used in a metadirective to select directives at execution time when the
condition(boolean-exp) 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.)

Example metadirective.4.c (omp_5.2)

```c
#define N 100
#include <stdbool.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() == 1 && omp_get_thread_num() == 0)
        {printf("PASSED 2 of 3\n");}
        #pragma omp metadirective \
        when( construct={parallel}, \
        user={condition(unbalanced)}: for schedule(guided) \
        private(b)) \
        when( construct={parallel} : for schedule(static))
        for (int i=0; i<n; i++) {a[i]=i; if(i==n-1) b=1;}
    } // if guided b=0, because b is private
    if(b==0) printf("PASSED 3 of 3\n");
}
void foo(int *a, int n, bool use_gpu);
void bar(int *a, int n, bool run_parallel, bool unbalanced);
int main(){
    int p[N];
    // App normally sets these, dependent on input parameters
    bool use_gpu=true, run_parallel=true, unbalanced=true;
    // Testing: set Env Var MK_FAIL to anything to fail tests
    if(getenv("MK_FAIL")!=NULL) {
        use_gpu=false; run_parallel=false; unbalanced=false;
    }
    foo(p, N, use_gpu);
    bar(p, N, run_parallel, unbalanced);
}
Example metadirective.4.f90 (omp_5.2)

subroutine foo(a, n, use_gpu)
  integer :: n, a(n)
  logical :: use_gpu
  integer :: b=0 ! use b to detect if run on gpu
  !$omp metadirective &
  !$omp& when(user={condition(use_gpu)}: &
  !$omp& target teams distribute parallel for &
  !$omp& private(b) map(from:a(1:n)) ) &
  !$omp& otherwise( &
  !$omp& parallel do)
  do i = 1,n; a(i)=i; if(i==n) b=1; end do
  if(b==0) print *, "PASSED 1 of 3" ! bc b is firstprivate for gpu run
end subroutine

subroutine bar (a, n, run_parallel, unbalanced)
  use omp_lib, only : omp_get_thread_num
  integer :: n, a(n)
  logical :: run_parallel, unbalanced
  integer :: b=0
  !$omp begin metadirective when(user={condition(run_parallel)}: parallel)
  if(omp_in_parallel() == 1 .and. omp_get_thread_num() == 0) &
    print *,"PASSED 2 of 3"
  !$omp metadirective &
  !$omp& when(construct={parallel}, user={condition(unbalanced)}: &
  !$omp& for schedule(guided) private(b) ) &
  !$omp& when(construct={parallel}: for schedule(static))
  do i = 1,n; a(i)=i; if(i==n) b=1; end do
  !$omp end metadirective
  if(b==0) print *, "PASSED 3 of 3" !!if guided, b=0 since b is private
end subroutine

program meta
  use omp_lib
  integer, parameter :: N=100
  integer :: p(N)
  integer :: env_stat

CHAPTER 12. PROGRAM CONTROL  489
!! App normally sets these, dependent on input parameters
logical :: use_gpu=.true., run_parallel=.true., unbalanced=.true.

!! Testing: set Env Var MK_FAIL to anything to fail tests
call get_environment_variable('MK_FAIL',status=env_stat)
if(env_stat /= 1) then
  use_gpu=.false.; run_parallel=.false.; unbalanced=.false.
endif

call foo(p, N, use_gpu)
call bar(p, N, run_parallel,unbalanced)

end program

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.

```cpp
#include <stdio.h>

// revised Fibonacci from tasking.4.c example

template <bool tasking>
int fib(int n) {
    int i, j;
    if (n<2) {
        return n;
    } else if ( tasking && n<8 ) { // serial/taskless cutoff for n<8
        return fib<false>(n);
    } else {
        #pragma omp metadirective 
        when(user={condition(tasking)}: task shared(i))
        {
            i=fib<tasking>(n-1);
        }
        #pragma omp metadirective 
        when(user={condition(tasking)}: task shared(j))
        {
            j=fib<tasking>(n-2);
        }
    }
}
```

C++ Example metadirective.5.cpp (omp_5.0)
```
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
```
12.8 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
S-1 void work(int i, int j) {}
S-2
S-3 void good_nesting(int n)
S-4 {
S-5 int i, j;
S-6 #pragma omp parallel default(shared)
S-7 {
S-8 #pragma omp for
S-9 for (i=0; i<n; i++) {
S-10 #pragma omp parallel shared(i, n)
S-11 {
S-12 #pragma omp for
S-13 for (j=0; j < n; j++)
S-14 work(i, j);
S-15 }
S-16 }
S-17 }
S-18 }
```

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

OpenMP Examples Version 5.2 - April 2022
The following variation of the preceding example is also conforming:

```
Example nested_loop.2.c

```
Example nested_loop.2.f

```
S-1 SUBROUTINE WORK(I, J)
S-2 INTEGER I, J
S-3 END SUBROUTINE WORK
S-4
S-5 SUBROUTINE WORK1(I, N)
S-6 INTEGER J
S-7 !$OMP PARALLEL DEFAULT(SHARED)
S-8 !$OMP DO
S-9 DO J = 1, N
S-10 CALL WORK(I, J)
S-11 END DO
S-12 !$OMP END PARALLEL
S-13 END SUBROUTINE WORK1
S-14
S-15 SUBROUTINE GOOD_NESTING2(N)
S-16 INTEGER N
S-17 !$OMP PARALLEL DEFAULT(SHARED)
S-18 !$OMP DO
S-19 DO I = 1, N
S-20 CALL WORK1(I, N)
S-21 END DO
S-22 !$OMP END PARALLEL
S-23 END SUBROUTINE GOOD_NESTING2
```
12.9 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

S-1
S-2    void work(int i, int j) {}
S-3
S-4    void wrong1(int n)
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++

Example nesting_restrict.1.f

S-1
S-2    SUBROUTINE WORK(I, J)
S-3    INTEGER I, J
S-4
S-5    END SUBROUTINE WORK
S-6
S-7    SUBROUTINE WRONG1(N)
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
```
The following orphaned version of the preceding example is also non-conforming:

```c
void work(int i, int j) {}
void work1(int i, int n)
{
    int j;
    /* incorrect nesting of loop regions */
    #pragma omp for
    for (j=0; j<n; j++)
        work(i, j);
}

void wrong2(int n)
{
    #pragma omp parallel default(shared)
    {
        int i;
        #pragma omp for
        for (i=0; i<n; i++)
            work1(i, n);
    }
}
```
Example nesting_restrict.2.f

```fortran
SUBROUTINE WORK1(I,N)
   INTEGER I, N
   INTEGER J
   !$OMP DO ! incorrect nesting of loop regions
   DO J = 1, N
      CALL WORK(I,J)
   END DO
END SUBROUTINE WORK1
```

Example nesting_restrict.3.c

```c
void work(int i, int j) {}
void wrong3(int n)
{
   #pragma omp parallel default(shared)
   {
      int i;
      #pragma omp for
      for (i=0; i<n; i++) {
         /* incorrect nesting of regions */
         #pragma omp single
         work(i, 0);
      }
   }
}
```
Example nesting_restrict.3.f

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

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

Example nesting_restrict.4.c

void work(int i, int j) {}
void wrong4(int n)
{
  #pragma omp parallel default(shared)
  {
    int i;
    #pragma omp for
    for (i=0; i<n; i++) {
      work(i, 0);
    }
    /* incorrect nesting of barrier region in a loop region */
    #pragma omp barrier
    work(i, 1);
  }
}
Example nesting_restrict.4.f

```fortran
SUBROUTINE WRONG4(N)
  INTEGER N
  INTEGER I
  !$OMP PARALLEL DEFAULT(SHARED)
  !$OMP DO DO I = 1, N
  CALL WORK(I, 1)
  ! incorrect nesting of barrier region in a loop region
  !$OMP BARRIER
  CALL WORK(I, 2)
  END DO
  !$OMP END PARALLEL
END SUBROUTINE WRONG4
```

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

Example nesting_restrict.5.c

```c
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);
    }
  }
}
```
Example nesting_restrict.5.f

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

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

Example nesting_restrict.6.c

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 }
Example nesting_restrict.6.f

SUBROUTINE WRONG6(N)
  INTEGER N
  !$OMP PARALLEL DEFAULT(SHARED)
  !$OMP SINGLE
  CALL WORK(N,1)
  !$OMP BARRIER
  CALL WORK(N,2)
  !$OMP END SINGLE
  !$OMP END PARALLEL
END SUBROUTINE WRONG6
12.10 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 VERBOS.

---

C / C++

Example target_offload_control.1.c (omp_5.0)

```c
#include <omp.h>
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>
#include <string.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--;
```
\[ n = (\text{int})(\text{end} - \text{env}); \]

// 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;
```

```c
int main()
{
    int i;
    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, "
        "it will be ignored.\n");
    }

    // Set target device number to an unavailable device to test offload policy.
    device_num = omp_get_num_devices() + 1;

    // Policy:
    printf("OMP_TARGET_OFFLOAD Policy: ");
    if (policy==MANDATORY)
        printf("MANDATORY-Terminate if dev. not avail\n");
    else if(policy==DISABLED)
        printf("DISABLED -(if supported) Only on Host\n");
    else if(policy==DEFAULT)
        printf("DEFAULT -On host if device not avail\n");
    else if(policy==UNKNOWN)
        printf("OMP_TARGET_OFFLOAD has unknown value\n");
    else if(policy==NOTSET)
        printf("OMP_TARGET_OFFLOAD not set\n");

    on_init_dev = 1;
    // device# out of range--not supported
    #pragma omp target device(device_num) map(tofrom: on_init_dev)
```
on_init_dev=omp_is_initial_device();

if (policy == MANDATORY && _OPENMP >= 201811)
    printf("ERROR: OpenMP implementation ignored MANDATORY policy.\n");

printf("Target region executed on init dev %s\n", on_init_dev ? "TRUE":"FALSE");

return 0;
}

module offload_policy
    implicit none
    integer, parameter :: LEN_POLICY=10
contains
    character(LEN_POLICY) function get_offload_policy()
        character(64) :: env
        integer :: length, i
        env=repeat(' ',length)
        !policy is blank if not found *
        call get_environment_variable("OMP_TARGET_OFFLOAD",env,length)
        do i = 1,len(env) !Makes a-z upper case
            if(iachar(env(i:i))>96) env(i:i)=achar(iachar(env(i:i))-32)
        end do
        get_offload_policy = trim(adjustl(env)) !remove peripheral spaces
        if(length==0) get_offload_policy="NOTSET"
    return
end function
end module

program policy_test
    use omp_lib
    use offload_policy
    integer :: i, device_num
    logical :: on_init_dev

Example target_offload_control.1.f90 (omp_5.0)
character(LEN_POLICY) :: policy

policy = get_offload_policy() !!Get OMP_TARGET_OFFLOAD value

if (OPENMP_VERSION < 201811) then
  print*,"Warning: OMP_TARGET_OFFLOAD NOT supported by VER.", &
  OPENMP_VERSION
  print*," If OMP_TARGET_OFFLOAD is set, it will be ignored."
endif

! Set target device number to an unavailable device
! to test offload policy.
device_num = omp_get_num_devices() + 1

!! Report OMP_TARGET_OFFLOAD value
select CASE (policy)
  case("MANDATORY")
    print*,"Policy: MANDATORY-Terminate if dev. not avail."
  case("DISABLED")
    print*,"Policy: DISABLED-(if supported) Only on Host."
  case("DEFAULT")
    print*,"Policy: DEFAULT On host if device not avail."
  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
12.11 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
#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];
    }
}
```

Example reproducible.1.c (omp_5.0)
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
{
    printf("sum = %d on thread %d\n", sum, omp_get_thread_num());
}

return 0;

Example reproducible.1.f90 (omp_5.0)

program main
    use omp_lib
    implicit none
    integer, parameter :: n = 1000
    integer :: v(n), u(n)
    integer :: i
    integer, save :: sum
    !$omp threadprivate(sum)
    !! no data dependences, so can execute concurrently
    !$omp parallel do order(concurrent)
    do i = 1, n
        u(i) = i
        v(i) = i
        v(i) = v(i) + u(i) * u(i)
    end do
    !! with data dependences, so cannot execute iterations
    !! concurrently with the order(concurrent) clause
    !$omp parallel do ordered
    do i = 2, n
        v(i) = v(i) + u(i) * u(i)
        !$omp ordered
        v(i) = v(i) + v(i-1)
        !$omp end ordered
    end do
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.
// reproducible schedules are used for the following two constructs
#pragma omp for order(reproducible: concurrent) nowait
for (int i = 0; i < n; i++) {
    u[i] = i;
    v[i] = i;
}
#pragma omp for order(reproducible: concurrent)
for (int i = 0; i < n; i++) {
    v[i] += u[i] * u[i];
}
#pragma omp parallel
{
    // static schedules preserve data dependences between the loops
    #pragma omp for schedule(static) order(concurrent) nowait
    for (int i = 0; i < n; i++) {
        u[i] = i;
        v[i] = i;
    }
    #pragma omp for schedule(static) order(concurrent)
    for (int i = 0; i < n; i++) {
        v[i] += u[i] * u[i];
    }
}
#pragma omp parallel
{
    // the default reproducibility by the static schedule is not
    // preserved due to the unconstrained order clause.
    // use of nowait here could result in data race.
    #pragma omp for schedule(static) order(unconstrained: concurrent)
    for (int i = 0; i < n; i++) {
        u[i] = i;
        v[i] = i;
    }
    #pragma omp for schedule(static) order(unconstrained: concurrent)
    for (int i = 0; i < n; i++) {
        v[i] += u[i] * u[i];
    }
}
return 0;
Fortran

Example reproducible.2.f90 (omp_5.1)

program main
  implicit none
  integer, parameter :: n = 1000
  integer :: v(n), u(n)
  integer :: i

  !$omp parallel
    !! reproducible schedules are used the following two constructs
    !$omp do order(reproducible: concurrent) nowait
    do i = 1, n
      u(i) = i
      v(i) = i
    end do
    !$omp do order(reproducible: concurrent)
    do i = 1, n
      v(i) = v(i) + u(i) * u(i)
    end do
  !$omp end parallel

  !$omp parallel
    !! static schedules preserve data dependences between the loops
    !$omp do schedule(static) order(concurrent) nowait
    do i = 1, n
      u(i) = i
      v(i) = i
    end do
    !$omp do schedule(static) order(concurrent)
    do i = 1, n
      v(i) = v(i) + u(i) * u(i)
    end do
  !$omp end parallel

  !$omp parallel
    !! the default reproducibility by the static schedule is not
    !! preserved due to the unconstrained order clause.
    !! use of nowait here could result in data race.
    !$omp do schedule(static) order(unconstrained: concurrent)
    do i = 1, n
      u(i) = i
      v(i) = i
    end do
    !$omp do schedule(static) order(unconstrained: concurrent)
    do i = 1, n
      v(i) = v(i) + u(i) * u(i)
end do

!$omp end parallel

end program
12.12 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
#include <omp.h>
#include <stdio.h>
#include <stdlib.h>
#include <cublas_v2.h>
#include <cuda_runtime_api.h>

#define N 16384

void myVectorSet(int n, double s, double *x)
{
    for(int i=0; i<n; ++i) x[i] = s*(i+1);
}

void myDaxpy(int n, double s, double *x, double *y)
{
    for(int i=0; i<n; ++i) y[i] = s*x[i]+y[i];
}

void myDscal(int n, double s, double *x)
{
    for(int i=0; i<n; ++i) x[i] = s*x[i];
}
```

Example interop.1.c (omp_5.1)
int main()
{
    const double scalar=2.0;
    double *x, *y, *d_x, *d_y;
    int    dev;

    omp_interop_t obj=omp_interop_none;
    intptr_t type;

    // Async Memcpy requires pinned memory
    cudaMemcpy( (void**)&x, N*sizeof(double) );
    cudaMemcpy( (void**)&y, N*sizeof(double) );
    cudaMemcpy( (void**)&d_x, N*sizeof(double) );
    cudaMemcpy( (void**)&d_y, N*sizeof(double) );

    dev = omp_get_default_device();
    omp_target_associate_ptr(&x[0], d_x, sizeof(double)*N, 0, dev);
    omp_target_associate_ptr(&y[0], d_y, sizeof(double)*N, 0, dev);

    #pragma omp target nowait depend(out: x[0:N]) map(from: x[0:N]) device(dev)
    myVectorSet(N, 1.0, x);

    #pragma omp task depend(out: y[0:N])
    myVectorSet(N, -1.0, y);

    // get obj for syncing
    #pragma omp interop init(targetsync: obj) device(dev) \
        depend(in: x[0:N]) depend(inout: y[0:N])
    int id = (int )omp_get_interop_int(obj, omp_ipr_fr_id, NULL);
    char* rt_name = (char*)omp_get_interop_str(obj, omp_ipr_fr_name, NULL);

    if(obj != omp_interop_none && id == omp_ifr_cuda) {
        printf(" OpenMP working with %s runtime to execute cublas daxpy.\n", rt_name);
        cublasHandle_t handle;
        int rc;
        cublasCreate(&handle);

        cudaMemcpy( (cudaStream_t)omp_get_interop_ptr(obj, omp_ipr_targetsync, &rc);
        if(rc != omp_irc_success) {
            fprintf(stderr,"ERROR: Failed to get %s stream, rt error= %d.\n", rt_name, rc);
            if(rc == omp_irc_no_value)
fprintf(stderr,
    "Parameters valid, no meaningful value available.");
exit(1);
}

cublasSetStream( handle,s );
cudaMemcpyAsync( d_x, x, N*sizeof(double),
    cudaMemcpyHostToDevice, s );
cudaMemcpyAsync( d_y, y, N*sizeof(double),
    cudaMemcpyHostToDevice, s );
cublasDaxpy( handle, N, &scalar, &d_x[0], 1, &d_y[0], 1 ) ;
cudaMemcpyAsync( y, d_y, N*sizeof(double),
    cudaMemcpyDeviceToHost, s );

} else {    // Execute as OpenMP offload.

    printf(" Notice: Offloading myDaxpy to perform daxpy calculation.\n");

    #pragma omp target depend(inout: y[0:N]) depend(in: x[0:N]) nowait \
        map(to: x[0:N]) map(tofrom: y[0:N]) device(dev)
        myDaxpy(N, scalar, x, y);

} // This also ensures foreign tasks complete.

#pragma omp interop destroy(obj) nowait depend(out: y[0:N])

#pragma omp target depend(inout: x[0:N])
myDscal(N, scalar, x);

#pragma omp taskwait
printf("(-1:-16384) %f:%f\n", y[0], y[N-1]);
printf("(-2:-32768) %f:%f\n", x[0], x[N-1]);
12.13 Utilities

This section contains examples of utility routines and features.

12.13.1 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
```

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

void work_to_be_timed()
{
    sleep(2);
}

int main()
{
    double start, end;

    start = omp_get_wtime();
    work_to_be_timed(); // any parallel or serial codes
    end = omp_get_wtime();

    printf("Work took %f seconds\n", end - start);
    printf("Precision of the timer is %f (sec)\n", omp_get_wtick());
    return 0;
}
```

```
C / C++
```
Example get_wtime.1.f90

```fortran
subroutine work_to_be_timed
  use, intrinsic :: iso_c_binding, only: c_int
  interface
    subroutine fsleep(sec) bind(C, name="sleep")
    import c_int
    integer(c_int), value :: sec
   end subroutine
  end interface
  call fsleep(2)
end subroutine

program do_work
  use omp_lib
  implicit none
  double precision :: start, end
  start = omp_get_wtime()
  call work_to_be_timed ! any parallel or serial codes
  end = omp_get_wtime()
  print *, "Work took", end - start, "seconds"
  print *, "Precision of the timer is", omp_get_wtick(), "(sec)"
end program
```

12.13.2 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`. 

```fortran
program do_work
  use omp_lib
  implicit none
  double precision :: start, end
  start = omp_get_wtime()
  call work_to_be_timed ! any parallel or serial codes
  end = omp_get_wtime()
  print *, "Work took", end - start, "seconds"
  print *, "Precision of the timer is", omp_get_wtick(), "(sec)"
end program
```
Example display_env.1.c (omp_5.1)

```c
#include <omp.h>

//implementers: customize debug routines for app debugging
int debug(){ return 1; }
int debug_omp_verbose(){ return 0; }

int main()
{
    if( debug() ) omp_display_env( debug_omp_verbose() );
    return 0;
}
```

Example display_env.1.f90 (omp_5.1)

```fortran
!implementers: customize debug routines for app debugging
function debug()
    logical :: debug
    debug = .true.
end function

function debug_omp_verbose()
    logical :: debug_omp_verbose
    debug_omp_verbose = .false.
end function

program display_omp_environment
    use omp_lib
    logical :: debug, debug_omp_verbose
    if( debug() ) call omp_display_env( debug_omp_verbose() )
    !! ...
end program
```
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_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
```

### 12.13.3 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.

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

int main(){
    #pragma omp metadirective
    when(implementation={vendor(gnu)}: nothing ) \
    otherwise(error at(compilation) severity(fatal) \
```
message("GNU compiler required.")

if( omp_get_num_procs() < 3 ){
    #pragma omp error at(runtime) 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(translation) severity(warning) \ 
    message("Notice: master is deprecated.")
    #pragma omp error at(runtime) severity(warning) \ 
    message("Notice: masked used next release.")
    printf("Hello from thread number 0.
");
}

Example error.1.f90 (omp_5.2)

program main
use omp_lib

!$omp metadirective &
!$omp& when( implementation={vendor(gnu)}: nothing ) &
!$omp& otherwise( error at ( translation ) severity ( fatal ) &
!$omp& message( "GNU compiler required." ) )

if( omp_get_num_procs() < 3 ) then
    !$omp error at(runtime) severity(fatal) &
    !$omp& message("3 or more procs required.")
endif

 !$omp parallel master

!! Give notice about master deprecation at compile time and run time.
 !$omp error at(translation) severity(warning) &
 !$omp& message("Notice: master is deprecated.")
 !$omp error at(runtime) severity(warning) &
 !$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
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
#include <stdio.h>
#include <omp.h>
#include <omp-tools.h>

ompt_start_tool_result_t *ompt_start_tool(
    unsigned int omp_version,
    const char *runtime_version
){
    if(omp_version != _OPENMP)
        printf("Warning: OpenMP runtime version (%i) "
            "does not match the compile time version (%i)"
            " for runtime identifying as %s\n",
            omp_version, _OPENMP, runtime_version);
    return NULL;
}

int main(void){
    printf("Running with %i threads\n", omp_get_max_threads());
    return 0;
}
```
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

<table>
<thead>
<tr>
<th>Version</th>
<th>Deprecated Feature</th>
<th>Replacement</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.2</td>
<td><code>default</code> clause on metadirectives</td>
<td><code>otherwise</code> clause</td>
</tr>
<tr>
<td>5.2</td>
<td>delimited <code>declare target</code> directive for C/C++</td>
<td><code>begin declare target</code> directive</td>
</tr>
<tr>
<td>5.2</td>
<td><code>to</code> clause on <code>declare target</code> directive</td>
<td><code>enter</code> clause</td>
</tr>
<tr>
<td>5.2</td>
<td>non-argument <code>destroy</code> clause on <code>depobj</code> construct</td>
<td><code>destroy(argument)</code></td>
</tr>
<tr>
<td>5.2</td>
<td><code>allocate</code> construct for Fortran ALLOCATE statements</td>
<td><code>allocators</code> construct</td>
</tr>
<tr>
<td>5.2</td>
<td><code>depend</code> clause on <code>ordered</code> construct</td>
<td><code>doacross</code> clause</td>
</tr>
<tr>
<td>5.2</td>
<td><code>linear</code> (<code>modifier(list): linear-step</code>) clause</td>
<td><code>linear(list: step(linear-step), modifier) clause</code></td>
</tr>
<tr>
<td>5.1</td>
<td><code>master</code> construct</td>
<td><code>masked</code> construct</td>
</tr>
<tr>
<td>5.1</td>
<td><code>master</code> affinity policy</td>
<td><code>primary</code> affinity policy</td>
</tr>
<tr>
<td>5.0</td>
<td><code>omp_lock_hint_*</code> constants</td>
<td><code>omp_sync_hint_*</code> constants</td>
</tr>
</tbody>
</table>

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 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.2: Updated Examples for Version 5.2

<table>
<thead>
<tr>
<th>Example Name</th>
<th>Earlier Version</th>
<th>Feature Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>error.1.c, f90</td>
<td>5.1</td>
<td><strong>default</strong> clause on metadirectives replaced with <strong>otherwise</strong> clause</td>
</tr>
<tr>
<td>metadirective.1.c, f90</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>metadirective.2.c, f90</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>metadirective.3.c, f90</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>metadirective.4.c, f90</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>target_ptr_map.4.c</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>target_ptr_map.5.c, f90</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>array_shaping.1.f90</td>
<td>5.0</td>
<td><strong>to</strong> clause on <strong>declare target</strong> directive replaced with <strong>enter</strong> clause</td>
</tr>
<tr>
<td>target_reverse_offload.7.c</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>target_task_reduction.1.c, f90</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>target_task_reduction.2a.c, f90</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>target_task_reduction.2b.c, f90</td>
<td>5.1</td>
<td></td>
</tr>
<tr>
<td>array_shaping.1.c</td>
<td>5.0</td>
<td><strong>delimited</strong> <strong>declare target</strong> directive replaced with <strong>begin declare target</strong> directive for C/C++</td>
</tr>
<tr>
<td>async_target.1.c</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>async_target.2.c</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>declare_target.1.c</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>declare_target.2c.cpp</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>declare_target.3.c</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>declare_target.4.c</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>declare_target.5.c</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>declare_target.6.c</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>declare_variant.1.c</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>device.1.c</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>metadirective.3.c</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>target_ptr_map.2.c</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>target_ptr_map.3a.c</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>target_ptr_map.3b.c</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>target_struct_map.1.c</td>
<td>5.0</td>
<td></td>
</tr>
</tbody>
</table>

*table continued on next page*
Table A.3 lists the updated examples for 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.3**: Updated Examples for Version 5.1

<table>
<thead>
<tr>
<th>Example Name</th>
<th>Earlier Version</th>
<th>Feature Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td>target_struct_map.2.cpp</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>target_struct_map.3.c</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>target_struct_map.4.c</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>doacross.1.c, f90</td>
<td>4.5</td>
<td>depend clause on ordered construct replaced with doacross clause</td>
</tr>
<tr>
<td>doacross.2.c, f90</td>
<td>4.5</td>
<td></td>
</tr>
<tr>
<td>doacross.3.c, f90</td>
<td>4.5</td>
<td></td>
</tr>
<tr>
<td>doacross.4.c, f90</td>
<td>4.5</td>
<td></td>
</tr>
<tr>
<td>linear_modifer.1.cpp, f90</td>
<td>4.5</td>
<td>modifier syntax change for linear clause on declare simd directive</td>
</tr>
<tr>
<td>linear_modifer.2.cpp, f90</td>
<td>4.5</td>
<td></td>
</tr>
<tr>
<td>linear_modifer.3.c, f90</td>
<td>4.5</td>
<td></td>
</tr>
<tr>
<td>allocators.1.f90</td>
<td>5.0</td>
<td>allocate construct replaced with allocators construct for Fortran allocate statements</td>
</tr>
<tr>
<td>depobj.1.c, f90</td>
<td>5.0</td>
<td>argument added to destroy clause on depobj construct</td>
</tr>
<tr>
<td>affinity.5.c, f</td>
<td>4.0</td>
<td>master affinity policy replaced with primary policy</td>
</tr>
<tr>
<td>async_target.3.c, f90</td>
<td>5.0</td>
<td>master construct replaced with masked construct</td>
</tr>
<tr>
<td>cancellation.2.c, f90</td>
<td>4.0</td>
<td></td>
</tr>
<tr>
<td>copyprivate.2.c, f</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>fort_sa_private.5.f</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>lock_owner.1.c, f</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>masked.1.c, f</td>
<td>3.0; master.1.c, f</td>
<td></td>
</tr>
<tr>
<td>parallel_masked_taskloop.1.c, f90</td>
<td>5.0; parallel_master_taskloop.1.c, f90</td>
<td></td>
</tr>
<tr>
<td>reduction.6.c, f</td>
<td>3.0</td>
<td></td>
</tr>
<tr>
<td>target_task_reduction.1.c, f90</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>target_task_reduction.2b.c, f90</td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td>taskloop_simd_reduction.1.c, f90</td>
<td>5.0</td>
<td></td>
</tr>
</tbody>
</table>

Table continued from previous page
Table A.4 lists the updated examples for 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.4: Updated Examples for Version 5.0**

<table>
<thead>
<tr>
<th>Example Name</th>
<th>Earlier Version</th>
<th>Feature Updated</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>task_detach.1.c, f90</code></td>
<td>5.0</td>
<td></td>
</tr>
<tr>
<td><code>critical.2.c, f</code></td>
<td>4.5</td>
<td><code>omp_lock_hint_</code>* constants replaced with <code>omp_sync_hint_</code>* constants</td>
</tr>
<tr>
<td><code>init_lock_with_hint.1.cpp, f</code></td>
<td>4.5</td>
<td></td>
</tr>
</tbody>
</table>
B Document Revision History

B.1 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.2 for details.

• Added the following examples for the 5.2 features:
  – Mapping class objects with virtual functions (Section 6.7 on page 184)
  – allocators construct for Fortran allocate statement (Section 11.2 on page 451)
  – Behavior of reallocation of variables through OpenMP allocator in Fortran (Section 11.2 on page 451)

• 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 9)
  – filter clause on masked construct (Section 3.14 on page 49)
  – omp_all_memory reserved locator for specifying task dependences (Section 5.3.9 on page 122)
  – Behavior of Fortran allocatable variables in target regions (Section 6.5 on page 176)
  – Device memory routines in Fortran (Section 6.17.5 on page 258)
  – Partial tiles from tile construct (Section 8.3 on page 299)
  – Fortran associate names and selectors in target region (Section 10.14 on page 436)
  – allocate directive for variable declarations and allocate clause on task constructs (Section 11.2 on page 451)
– Controlling concurrency and reproducibility with \texttt{order} clause (Section 12.11 on page 506)

• Added other examples:
  – Using lambda expressions with \texttt{target} constructs (Section 6.14 on page 230)
  – Target memory and device pointer routines (Section 6.17.5 on page 258)
  – Examples to illustrate the ordering properties of the \texttt{flush} operation (Section 11.1 on page 440)
  – User selector in the \texttt{metadirective} directive (Section 12.7 on page 482)

\section*{B.2 Changes from 5.0.1 to 5.1}

• General changes:
  – Replaced \texttt{master} construct example with equivalent \texttt{masked} construct example (Section 3.14 on page 49)
  – Primary thread is now used to describe thread number 0 in the current team
  – \texttt{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 \texttt{omp_lock_hint\_\*} constants have been renamed \texttt{omp_sync_hint\_\*} (Section 9.1 on page 309, Section 9.11 on page 350)

• Added the following new chapters:
  – Deprecated Features (on page 525)
  – Directive Syntax (Section 2 on page 3)
  – Loop Transformations (Section 8 on page 285)
  – OMPT Interface (Section 13 on page 521)

• Added the following examples for the 5.1 features:
  – OpenMP directives in C++ \texttt{attribute} specifiers (Section 2.2 on page 5)
  – Directive syntax adjustment to allow Fortran \texttt{BLOCK ... END BLOCK} as a structured block (Section 2.4 on page 9)
  – \texttt{omp_target_is_accessible} API routine (Section 6.3 on page 160)
  – Fortran allocatable array mapping in \texttt{target} regions (Section 6.5 on page 176)
  – \texttt{begin declare target} (with \texttt{end declare target}) directive (Section 6.13.2 on page 218)
  – \texttt{tile} construct (Section 8.1 on page 285)
• Included additional examples for the 5.0 features:
  – **collapse** clause for non-rectangular loop nest (Section 3.8 on page 32)
  – **detach** clause for tasks (Section 5.4 on page 126)
  – Pointer attachment for a structure member (Section 6.4 on page 169)
  – Host and device pointer association with the **omp_target_associate_ptr** routine (Section 6.17.4 on page 255)
  – Sample code on activating the tool interface (Section 13.1 on page 522)
• Added other examples:
  – The **omp_get_wtime** routine (Section 12.13.1 on page 515)

## B.3 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 155)
  – Transferring noncontiguous data with the **target update** directive in Fortran (Section 6.8 on page 185)
  – **conditional** modifier for the **lastprivate** clause (Section 10.8 on page 382)
  – **task** modifier for the **reduction** clause (Section 10.9.2 on page 391)
  – Reduction on combined target constructs (Section 10.9.3 on page 396)
  – Task reduction with **target** constructs (Section 10.9.4 on page 400)
  – **scan** directive for returning the **prefix sum** of a reduction (Section 10.10 on page 425)
• Included additional examples for the 4.x features:
– Dependence for undeferred tasks (Section 5.3.9 on page 122)
– ref, val, uval modifiers for linear clause (Section 7.4 on page 277)

• Clarified the description of pointer mapping and pointer attachment in Section 6.3 on page 160.
• Clarified the description of memory model examples in Section 11.1 on page 440.

B.4 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 18)
  – loop and teams loop constructs specify loop iterations that can execute concurrently (Section 3.15 on page 52)
  – 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 68)
  – taskwait with dependences (Section 5.3.6 on page 110)
  – mutexinoutset task dependences (Section 5.3.7 on page 117)
  – Multidependence Iterators (in depend clauses) (Section 5.3.8 on page 120)
  – Combined constructs: parallel master taskloop and parallel master taskloop simd (Section 5.8 on page 140)
  – Reverse Offload through ancestor modifier of device clause. (Section 6.1.6 on page 152)
  – Pointer Mapping - behavior of mapped pointers (Section 6.3 on page 160)
  – Structure Mapping - behavior of mapped structures (Section 6.4 on page 169)
  – Array Shaping with the shape-operator (Section 6.8 on page 185)
  – The declare mapper directive (Section 6.9 on page 189)
  – Acquire and Release Semantics Synchronization: Memory ordering clauses acquire, release, and acq_rel were added to flush and atomic constructs (Section 9.7 on page 328)
  – depobj construct provides dependence objects for subsequent use in depend clauses (Section 9.9 on page 340)
  – reduction clause for task construct (Section 10.9.2 on page 391)
  – reduction clause for taskloop construct (Section 10.9.5 on page 405)
- reduction clause for taskloop simd construct (Section 10.9.5 on page 405)
- Memory Allocators for making OpenMP memory requests with traits (Section 11.2 on page 451)
- requires directive specifies required features of implementation (Section 12.5 on page 474)
- declare variant directive - for function variants (Section 12.6 on page 476)
- metadirective directive - for directive variants (Section 12.7 on page 482)
- OMP_TARGET_OFFLOAD Environment Variable - controls offload behavior (Section 12.10 on page 502)

- Included the following additional examples for the 4.x features:
  - more taskloop examples (Section 5.7 on page 136)
  - user-defined reduction (UDR) (Section 10.9.7 on page 414)

B.5 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(to/from) for scalar variables in a number of examples to comply with the change of the default behavior for scalar variables from map(to/from) to firstprivate in the 4.5 specification
- Added the following new examples:
  - linear clause in loop constructs (Section 3.9 on page 38)
  - priority clause for task construct (Section 5.2 on page 103)
  - taskloop construct (Section 5.7 on page 136)
  - directive-name modifier in multiple if clauses on a combined construct (Section 6.1.5 on page 149)
  - unstructured data mapping (Section 6.11 on page 209)
  - link clause for declare target directive (Section 6.13.5 on page 227)
  - asynchronous target execution with nowait clause (Section 6.16 on page 242)
  - device memory routines and device pointers (Section 6.17.5 on page 258)
  - doacross loop nest (Section 9.10 on page 344)
  - locks with hints (Section 9.11 on page 350)
C/C++ array reduction (Section 10.9.1 on page 384)
C++ reference types in data sharing clauses (Section 10.13 on page 434)

B.6 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 263)
- Applied miscellaneous fixes in several source codes
- Added the revision history

B.7 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 131)

B.8 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 105)
  - `target` construct (Section 6.1 on page 144)
  - array sections in device constructs (Section 6.6 on page 180)
  - `target data` construct (Section 6.10 on page 196)
  - `target update` construct (Section 6.12 on page 212)
  - `declare target` directive (Section 6.13 on page 216)
  - `teams` constructs (Section 6.15 on page 233)
  - asynchronous execution of a `target` region using tasks (Section 6.16.1 on page 242)
  - device runtime routines (Section 6.17 on page 251)
  - Fortran ASSOCIATE construct (Section 10.14 on page 435)
  - cancellation constructs (Section 12.4 on page 469)
Index

A
acq_rel clause, 328
acquire clause, 328
affinity
  affinity clause, 66
  close policy, 63
  master policy, 527
  primary policy, 65, 527
  proc_bind clause, 60
  spread policy, 60, 79
  task affinity, 66
affinity clause, 66
affinity display
  OMP_AFFINITY_FORMAT, 68
  omp_capture_affinity routine, 75
  OMP_DISPLAY_AFFINITY, 68
  omp_display_affinity routine, 68
  omp_get_affinity_format routine, 75
  omp_set_affinity_format routine, 75
affinity query
  omp_get_num_places routine, 79
  omp_get_place_num routine, 79
  omp_get_place_num_procs routine, 79
alloc map-type, 209
allocate construct, 526
allocate directive, 455
  allocator clause, 455
allocator clause, 451
allocators construct, 526
allocators directive, 451
  allocator clause, 451
always modifier, 255
ancestor modifier, 152
array sections
  in map clause, 180
array shaping
  in motion-clause, 185
ASSOCIATE construct, Fortran, 435
at clause, 518
atomic construct, 316, 322, 328, 330, 440
  capture clause, 319
  memory ordering clauses, 330
  read clause, 318, 330
  relaxed atomic operations, 332
  update clause, 316
  write clause, 318, 330
attribute syntax, C++, 5
B
barrier construct, 465
begin declare target directive, 162, 163, 169, 185, 216, 218, 221, 224, 227, 242, 251, 476, 485, 526
binding
  barrier regions, 314
C
cancel construct, 469
cancellation
  cancel construct, 469
  cancellation point construct, 470
  for parallel region, 469
  for taskgroup region, 471
  for worksharing region, 469
cancellation point construct, 470
capture clause, 319
clauses
  acq_rel, 328
  acquire, 328
  affinity, 66
  allocator, 451
  at, 518
  capture, 319
pragma, C/C++, 4
directives
allocate, 455
allocate, 451
begin declare target, 162, 163, 169, 185, 216, 218, 221, 224, 227, 242, 251, 476, 485, 526
declare mapper, 189
declare reduction, 414
declare simd, 224, 264
declare target, 152, 185, 216, 221, 224, 227, 242, 243, 251, 476, 485
declare variant, 476
error, 518
metadirective, 482
requires, 152, 474
scan, 425
threadprivate, 361, 428
dist_schedule clause, 238
distribute construct, 235
dist_schedule clause, 238
do construct, 15, 27
doacross clause, 344, 526
doacross loop nest
doacross clause, 344
ordered construct, 344
E
enter clause, 152, 400, 526
environment display
OMP_DISPLAY_ENV, 516
omp_display_env routine, 516
environment variables
OMP_AFFINITY_FORMAT, 68
OMP_DISPLAY_AFFINITY, 68
OMP_DISPLAY_ENV, 516
OMP_NUM_THREADS, 22
OMP_PLACES, 71, 79
OMP_TARGET_OFFLOAD, 502
error directive, 518
at clause, 518
severity clause, 518
example label, 2
omp_verno, 2
exclusive clause, 425
F
filter clause, 49
final clause, 98
firstprivate clause, 41
C/C++ arrays in, 380
flush construct, 325, 328, 332, 442, 465
flushes
acquire, 328
flush construct, 442
flush with a list, 444
flush without a list, 325
implicit, 328, 440
release, 328
for construct, 15
free form syntax, Fortran, 9
full clause, 289
G
grainsize clause, 136
H
hint clause, 309
I
if clause, 100, 149, 204, 214
in_reduction clause, 391, 400
with target construct, 403
inbranch clause, 270
inclusive clause, 425
init clause, 512
initializer clause, 414
inscan modifier, 425
internal control variables, 462
interop construct, 512
depend clause, 512
destroy clause, 512
init clause, 512
is_device_ptr clause, 258
iterator modifier, 120
L
lambda expressions, 230
lastprivate clause, 382
    conditional modifier, 383
linear clause, 38, 264, 277, 526
link clause, 227
loop construct, 52
loop scheduling
    static, 30
loop variables, Fortran, 373

M
map clause, 145, 180
    alloc map-type, 201, 209
    always modifier, 255
array sections in, 147
delete map-type, 209
from map-type, 146
mapper modifier, 190
to map-type, 146
tofrom map-type, 199
mapper modifier, 190
mapping
    allocatable array, Fortran, 176
deep copy, 167, 190
pointer, 160
pointer attachment, 160
structure, 169
virtual functions, C++, 184
masked construct, 49, 140, 430, 527
    filter clause, 49
master construct, 527
match clause, 476
memory allocators
    allocator traits, 451
    allocators directive, 451
    memory space, 451
    omp_alloc routine, 451
memory ordering clauses
    acq_rel, 328
    acquire, 328
    release, 328
    seq_cst, 330
mergeable clause, 97
metadirective directive, 482
    otherwise clause, 482
    when clause, 482
modifiers, linear
    ref, 277
    uval, 277
    val, 277
motion-clause
    from clause, 212
to clause, 212

N
nestable lock, 356
nested loop constructs, 492
nogroup clause, 136
non-rectangular loop nest, 36
notinbranch clause, 270
nowait clause, 29, 246, 248
num_teams clause, 233
num_threads clause, 25

O
OMP_AFFINITY_FORMAT, 68
omp_alloc routine, 451
omp_capture_affinity routine, 75
OMP_DISPLAY_AFFINITY, 68
omp_display_affinity routine, 68
OMP_DISPLAY_ENV, 516
omp_display_env routine, 516
omp_fulfill_event routine, 126
omp_get_affinity_format routine, 75
omp_get_interop_int routine, 512
omp_get_interop_ptr routine, 512
omp_get_num_places routine, 79
omp_get_num_teams routine, 233
omp_get_num_threads routine, 56
omp_get_place_num routine, 79
omp_get_place_num_procs routine, 79
omp_get_team_num routine, 233
omp_get_wtick routine, 515
omp_get_wtime routine, 515
omp_in_final routine, 98
omp_init_allocator routine, 451
omp_init_lock routine, 350
omp_init_lock_with_hint routine, 351
omp_is_initial_device routine, 251
OMP_NUM_THREADS, 22
OMP_PLACES, 71, 79
omp_set_affinity_format routine, 75
omp_set_default_device routine, 254
omp_set_dynamic routine, 25, 54
omp_set_lock routine, 353
omp_set_num_threads routine, 54
omp_target_alloc routine, 255, 258
omp_targetAssociate_ptr routine, 255, 259
omp_target_disassociate_ptr routine, 255
omp_target_free routine, 255, 258
omp_target_is_accessible routine, 166
omp_target_is_present routine, 259
omp_target_memcpy routine, 258
OMP_TARGET_OFFLOAD, 502
omp_test_lock routine, 353
omp_unset_lock routine, 352
omp_verno, 2
OMPT interface
   activating, 522
   ompt_set_callback routine, 522
   ompt_start_tool routine, 522
ompt_set_callback routine, 522
ompt_start_tool routine, 522
OpenMP variable identifiers
   omp_in, 414
   omp_orig, 419
   omp_out, 414
   omp_priv, 414
order (concurrent) clause, 506
reproducible modifier, 508
unconstrained modifier, 508
ordered clause, 34, 336
ordered construct, 336
doacross loop nest, 344
otherwise clause, 482, 526
P
parallel construct, 15, 16, 140
parallel sections construct, 40
partial clause, 289
pointer attachment, 160
pragma syntax, C/C++, 4
primary policy, 65
priority clause, 103
private clause, 267, 369
   common blocks, Fortran, 375
   storage association, Fortran, 377
proc_bind clause, 60
R
random access iterator, C++, 53
read clause, 318, 330
reduction clause, 267, 384
   inscan modifier, 425
   on scope construct, 412
   on target construct, 396
   on taskloop construct, 405
   on taskloop simd construct, 409
   on teams construct, 235
task modifier, 394, 402
reductions
   declare reduction directive, 414
   in_reduction clause, 391
   reduction clause, 267, 384
   task_reduction clause, 391
   user-defined, 414
region nesting rules, 495
release clause, 328
requires directive, 474
   reverse_offload clause, 152
   unified_shared_memory clause, 474
reverse_offload clause, 152
routines
   omp_alloc, 451
   omp_capture_affinity, 75
omp_display_affinity, 68
omp_display_env, 516
omp_fulfill_event, 126
omp_get_affinity_format, 75
omp_get_interop_int, 512
omp_get_interop_ptr, 512
omp_get_num_places, 79
omp_get_num_teams, 233
omp_get_num_threads, 56
omp_get_place_num, 79
omp_get_place_num_procs, 79
omp_get_team_num, 233
omp_get_wtick, 515
omp_get_wtime, 515
omp_in_final, 98
omp_init_allocator, 451
omp_init_lock, 350
omp_init_lock_with_hint, 351
omp_is_initial_device, 251
omp_set_affinity_format, 75
omp_set_default_device, 254
omp_set_dynamic, 25, 54
omp_set_lock, 353
omp_set_num_threads, 54
omp_target_alloc, 255, 258
omp_target_associate_ptr, 255, 259
omp_target_disassociate_ptr, 255
omp_target_free, 255, 258
omp_target_is_accessible, 166
omp_target_is_present, 259
omp_target_memcpy, 258
omp_test_lock, 353
omp_unset_lock, 352
ompt_set_callback, 522
ompt_start_tool, 522

schedule clause, 238
scope construct, 412
section construct, 41
sections construct, 41
seq_cst clause, 330
severity clause, 518
shared clause
  common blocks, Fortran, 375
  shared variables
    race conditions, 458
simd construct, 140, 263
single construct, 43, 430
sizes clause, 285
spread policy, 60, 79
standalone directive placement, 465
static scheduling, 30

T
target construct, 144, 180, 233, 242, 396, 400, 403
defaultmap clause, 155
depend clause, 248
device clause, 152
if clause, 149
implicit mapping, 144, 199
is_device_ptr clause, 258
map clause, 145, 199
nowait clause, 246, 248
target data construct, 180, 196, 212
  if clause, 204
  map clause, 199
target enter data construct, 209
target exit data construct, 209
target reverse offload
  requires directive, 152
  reverse_offload clause, 152
target update construct, 212, 255
  from clause, 185, 212
  if clause, 214
  motion-clause, 212
to clause, 185, 212
task construct, 84, 105, 242
depend clause, 105, 245
detach clause, 126

S
safelen clause, 268
scan directive, 425
  exclusive clause, 425
  inclusive clause, 425
final clause, 98
if clause, 100
mergeable clause, 97
priority clause, 103
untied clause, 90
task dependences
  anti dependence, 106
  concurrent execution with, 108
  flow dependence, 105
  matrix multiplication, 109
  mutually exclusive execution, 117
  output dependence, 107
taskwait construct with, 110
  undeferred tasks, 122
  using iterators, 120
task modifier, 394, 402
task scheduling point, 90
task_reduction clause, 391
taskgroup construct, 131
taskloop construct, 136, 140, 405
grainsize clause, 136
nogroup clause, 136
taskloop simd construct, 409
taskwait construct, 84, 110, 465
  depend clause, 110
taskyield construct, 134, 465
teams construct, 18, 233
  num_teams clause, 233
threadprivate directive, 361, 428
tile construct, 285
  sizes clause, 285
to clause, 526

U
unified_shared_memory clause, 474
uniform clause, 264
unroll construct, 289
  full clause, 289
  partial clause, 289
untied clause, 90
update clause, 316, 340

W
when clause, 482