OpenMP Technical Report 6: Version 5.0 Preview 2

This Technical Report augments the OpenMP 4.5 Application Programming Interface Specification with language features for concurrent loops, task reductions, a runtime interface for first-party (OMPT) and for third-party tools (OMPD), major extensions to the device constructs, memory allocation features, new task dependencies, and several clarifications and corrections.

EDITORS

Bronis R. de Supinski

Michael Klemm

November 9, 2017

Expires November 8, 2019

We actively solicit comments. Please provide feedback on this document either to the Editors directly or in the OpenMP Forum at openmp.org

End of Public Comment Period: January 8, 2018

OpenMP Architecture Review Board – www.openmp.org – info@openmp.org
Ravi S. Rao, OpenMP, c/o Intel Corporation, 1300 MoPac Express Way, Austin, TX 78746, USA

This technical report describes possible future directions or extensions to the OpenMP Specification.

The goal of this technical report is to build more widespread existing practice for an expanded OpenMP. It gives advice on extensions or future directions to those vendors who wish to provide them possibly for trial implementation, allows OpenMP to gather early feedback, support timing and scheduling differences between official OpenMP releases, and offers a preview to users of the future directions of OpenMP with the provisions stated in the next paragraph.

This technical report is non-normative. Some of the components in this technical report may be considered for standardization in a future version of OpenMP, but they are not currently part of any OpenMP Specification. Some of the components in this technical report may never be standardized, others may be standardized in a substantially changed form, or it may be standardized as is in its entirety.

OpenMP Application Programming Interface

Version 5.0 rev 2, November 2017

Copyright © 1997-2017 OpenMP Architecture Review Board.
Permission to copy without fee all or part of this material is granted, provided the OpenMP Architecture Review Board copyright notice and the title of this document appear. Notice is given that copying is by permission of OpenMP Architecture Review Board.

This page intentionally left blank in published version.

This is Revision 2-TR6 (Second Official Draft) (9 November 2017) and includes the following internal tickets applied to the 4.5 LaTeX sources: 50, 354, 389, 399, 425, 452, 459, 461-463, 465-467, 484, 486, 489, 491-504, 508, 510, 511, 514, 518, 520, 521, 523, 524, 530-533, 536, 537, 545, 548, 551, 555-562, 564, 568-577, 581-584, 586, 589-592, 594, 597, 601, 603, 606, 608, 610, 611-615, 617, 618, 620, 621, 625-631, 633-644, 648, 650

This is a draft; contents will change in official release

Contents

1. Introduction **1**

1.1. Scope . . . 1
1.2. Glossary . . . 2
1.2.1. Threading Concepts . . . 2
1.2.2. OpenMP Language Terminology . . . 2
1.2.3. Loop Terminology . . . 8
1.2.4. Synchronization Terminology . . . 9
1.2.5. Tasking Terminology . . . 10
1.2.6. Data Terminology . . . 12
1.2.7. Implementation Terminology . . . 15
1.2.8. Tool Terminology . . . 16
1.3. Execution Model . . . 18
1.4. Memory Model . . . 20
1.4.1. Structure of the OpenMP Memory Model . . . 20
1.4.2. Device Data Environments . . . 22
1.4.3. Memory Management . . . 22
1.4.4. The Flush Operation . . . 22
1.4.5. Flush Synchronization and Happens Before . . . 24
1.4.6. OpenMP Memory Consistency . . . 25
1.5. Tool Interface . . . 26
1.5.1. OMPT . . . 26
1.5.2. OMPD . . . 27
1.6. OpenMP Compliance . . . 28
1.7. Normative References . . . 28
1.8. Organization of this Document . . . 31

2. Directives 35
2.1. Directive Format 36
2.1.1. Fixed Source Form Directives 39
2.1.2. Free Source Form Directives 40
2.1.3. Stand-Alone Directives 43
2.2. Conditional Compilation 44
2.2.1. Fixed Source Form Conditional Compilation Sentinels 45
2.2.2. Free Source Form Conditional Compilation Sentinel 45
2.3. **requires** Directive 46
2.4. Internal Control Variables 49
2.4.1. ICV Descriptions 49
2.4.2. ICV Initialization 51
2.4.3. Modifying and Retrieving ICV Values 53
2.4.4. How ICVs are Scoped 56
2.4.4.1. How the Per-Data Environment ICVs Work 57
2.4.5. ICV Override Relationships 58
2.5. Array Sections 60
2.6. Iterators 62
2.7. Memory Allocators 64
2.8. **parallel** Construct 66
2.8.1. Determining the Number of Threads for a **parallel** Region 71
2.8.2. Controlling OpenMP Thread Affinity 73
2.9. Canonical Loop Form 75
2.10. Worksharing Constructs 77
2.10.1. Loop Construct 78
2.10.1.1. Determining the Schedule of a Worksharing Loop 86
2.10.2. **sections** Construct 86
2.10.3. **single** Construct 90
2.10.4. **workshare** Construct 93
2.11. SIMD Constructs 96
2.11.1. **simd** Construct 96
2.11.2. **declare simd** Directive 100
2.11.3. Loop SIMD Construct 105

2.12. **concurrent** Construct 107
2.13. Tasking Constructs 110
2.13.1. **task** Construct 110
2.13.2. **taskloop** Construct 114
2.13.3. **taskloop simd** Construct 121
2.13.4. **taskyield** Construct 123
2.13.5. Initial Task 124
2.13.6. Task Scheduling 125
2.14. Memory Management Directives 127
2.14.1. **allocate** Directive 127
2.14.2. **allocate** Clause 130
2.15. Device Constructs 131
2.15.1. Device Initialization 131
2.15.2. **target data** Construct 132
2.15.3. **target enter data** Construct 134
2.15.4. **target exit data** Construct 137
2.15.5. **target** Construct 141
2.15.6. **target update** Construct 146
2.15.7. **declare target** Directive 150
2.15.8. **declare mapper** Directive 155
2.15.9. **teams** Construct 157
2.15.10.**distribute** Construct 160
2.15.11.**distribute simd** Construct 164
2.15.12.Distribute Parallel Loop Construct 165
2.15.13.Distribute Parallel Loop SIMD Construct 167
2.16. Combined Constructs 169
2.16.1. Parallel Loop Construct 169
2.16.2. **parallel sections** Construct 171
2.16.3. **parallel workshare** Construct 172
2.16.4. Parallel Loop SIMD Construct 173
2.16.5. **target parallel** Construct 175
2.16.6. Target Parallel Loop Construct 176
2.16.7. Target Parallel Loop SIMD Construct 178

2.16.8. **target simd** Construct 179
2.16.9. **target teams** Construct 180
2.16.10. **teams distribute** Construct 182
2.16.11. **teams distribute simd** Construct 183
2.16.12. **target teams distribute** Construct 184
2.16.13. **target teams distribute simd** Construct 185
2.16.14. Teams Distribute Parallel Loop Construct 187
2.16.15. Target Teams Distribute Parallel Loop Construct 188
2.16.16. Teams Distribute Parallel Loop SIMD Construct 189
2.16.17. Target Teams Distribute Parallel Loop SIMD Construct 191
2.17. **if** Clause 192
2.18. Master and Synchronization Constructs and Clauses 193
2.18.1. **master** Construct 194
2.18.2. **critical** Construct 195
2.18.3. **barrier** Construct 198
2.18.4. Implicit Barriers 200
2.18.5. **taskwait** Construct 202
2.18.6. **taskgroup** Construct 204
2.18.7. **atomic** Construct 206
2.18.8. **flush** Construct 215
2.18.9. **ordered** Construct 221
2.18.10. **depend** Clause 225
2.18.11. Synchronization Hints 229
2.19. Cancellation Constructs 232
2.19.1. **cancel** Construct 232
2.19.2. **cancellation point** Construct 237
2.20. Data Environment 239
2.20.1. Data-sharing Attribute Rules 239
2.20.1.1. Data-sharing Attribute Rules for Variables Referenced in a Construct 240
2.20.1.2. Data-sharing Attribute Rules for Variables Referenced in a Region but not in a Construct 243
2.20.2. **threadprivate** Directive 244

2.20.3. Data-Sharing Attribute Clauses 249
2.20.3.1. **default** Clause 250
2.20.3.2. **shared** Clause 251
2.20.3.3. **private** Clause 252
2.20.3.4. **firstprivate** Clause 256
2.20.3.5. **lastprivate** Clause 259
2.20.3.6. **linear** Clause 262
2.20.4. Reduction Clauses 265
2.20.4.1. Properties Common To All Reduction Clauses 266
2.20.4.2. Reduction Scoping Clauses 271
2.20.4.3. Reduction Participating Clauses 271
2.20.4.4. **reduction** Clause 272
2.20.4.5. **task_reduction** Clause 273
2.20.4.6. **in_reduction** Clause 274
2.20.5. Data Copying Clauses 275
2.20.5.1. **copyin** Clause 275
2.20.5.2. **copyprivate** Clause 277
2.20.6. Data-mapping Attribute Rules and Clauses 279
2.20.6.1. **map** Clause 280
2.20.6.2. **defaultmap** Clause 288
2.21. **declare reduction** Directive 289
2.22. Nesting of Regions 295

3. Runtime Library Routines **297**
3.1. Runtime Library Definitions 298
3.2. Execution Environment Routines 300
3.2.1. **omp_set_num_threads** 300
3.2.2. **omp_get_num_threads** 301
3.2.3. **omp_get_max_threads** 302
3.2.4. **omp_get_thread_num** 304
3.2.5. **omp_get_num_procs** 305
3.2.6. **omp_in_parallel** 305
3.2.7. **omp_set_dynamic** 306
3.2.8. **omp_get_dynamic** 308

3.2.9. omp_get_cancellation 308
3.2.10. omp_set_nested 309
3.2.11. omp_get_nested 311
3.2.12. omp_set_schedule 311
3.2.13. omp_get_schedule 313
3.2.14. omp_get_thread_limit 314
3.2.15. omp_set_max_active_levels 315
3.2.16. omp_get_max_active_levels 317
3.2.17. omp_get_level 318
3.2.18. omp_get_ancestor_thread_num 319
3.2.19. omp_get_team_size 320
3.2.20. omp_get_active_level 321
3.2.21. omp_in_final 322
3.2.22. omp_get_proc_bind 323
3.2.23. omp_get_num_places 325
3.2.24. omp_get_place_num_procs 326
3.2.25. omp_get_place_proc_ids 327
3.2.26. omp_get_place_num 328
3.2.27. omp_get_partition_num_places 329
3.2.28. omp_get_partition_place_nums 330
3.2.29. omp_set_affinity_format 331
3.2.30. omp_get_affinity_format 332
3.2.31. omp_display_affinity 333
3.2.32. omp_capture_affinity 334
3.2.33. omp_set_default_device 336
3.2.34. omp_get_default_device 337
3.2.35. omp_get_num_devices 337
3.2.36. omp_get_device_num 338
3.2.37. omp_get_num_teams 339
3.2.38. omp_get_team_num 340
3.2.39. omp_is_initial_device 341
3.2.40. omp_get_initial_device 342
3.2.41. omp_get_max_task_priority 343

3.3. Lock Routines 344
3.3.1. **omp_init_lock** and **omp_init_nest_lock** 346
3.3.2. **omp_init_lock_with_hint** and
omp_init_nest_lock_with_hint 347
3.3.3. **omp_destroy_lock** and **omp_destroy_nest_lock** 349
3.3.4. **omp_set_lock** and **omp_set_nest_lock** 350
3.3.5. **omp_unset_lock** and **omp_unset_nest_lock** 352
3.3.6. **omp_test_lock** and **omp_test_nest_lock** 354
3.4. Timing Routines 356
3.4.1. **omp_get_wtime** 356
3.4.2. **omp_get_wtick** 357
3.5. Device Memory Routines 358
3.5.1. **omp_target_alloc** 358
3.5.2. **omp_target_free** 360
3.5.3. **omp_target_is_present** 361
3.5.4. **omp_target_memcpy** 362
3.5.5. **omp_target_memcpy_rect** 363
3.5.6. **omp_target_associate_ptr** 365
3.5.7. **omp_target_disassociate_ptr** 367
3.6. Memory Management Routines 368
3.6.1. Memory Management Types 368
3.6.2. **omp_set_default_allocator** 369
3.6.3. **omp_get_default_allocator** 370
3.6.4. **omp_alloc** 371
3.6.5. **omp_free** 372
3.7. Tool Control Routines 372

4. Tool Support **377**
4.1. OMPT 378
4.1.1. Activating an OMPT Tool 378
4.1.1.1. Determining Whether an OMPT Tool Should be Initialized . . . 378
4.1.1.2. Initializing an OMPT Tool 379
4.1.1.3. Monitoring Activity on the Host with OMPT 381
4.1.1.4. Tracing Activity on Target Devices with OMPT 383

4.1.2. Finalizing an OMPT Tool 388
4.1.3. OMPT Data Types 388
4.1.3.1. Tool Initialization and Finalization 388
4.1.3.2. Callbacks 389
4.1.3.3. Tracing 391
4.1.3.4. Miscellaneous Type Definitions 394
4.1.4. OMPT Tool Callback Signatures and Trace Records 403
4.1.4.1. Initialization and Finalization Callback Signature 404
4.1.4.2. Event Callback Signatures and Trace Records 406
4.1.5. OMPT Runtime Entry Points for Tools 440
4.1.5.1. Entry Points in the OMPT Callback Interface 441
4.1.5.2. Entry Points in the OMPT Device Tracing Interface 461
4.1.5.3. Lookup Entry Point 475
4.2. OMPD 476
4.2.1. Activating an OMPD Tool 477
4.2.1.1. Enabling the Runtime for OMPD 478
4.2.1.2. Finding the OMPD plugin 478
4.2.2. OMPD Data Types 479
4.2.2.1. Basic Types 479
4.2.2.2. System Device Identifiers 481
4.2.2.3. Thread Identifiers 481
4.2.2.4. OMPD Handle Types 482
4.2.2.5. Tool Context Types 482
4.2.2.6. Return Code Types 483
4.2.2.7. OpenMP Scheduling Types 484
4.2.2.8. OpenMP Proc Binding Types 485
4.2.2.9. Primitive Types 485
4.2.2.10. Runtime State Types 486
4.2.3. OMPD Tool Callback Interface 487
4.2.3.1. Memory Management of OMPD Plugin 488
4.2.3.2. Context Management and Navigation 490
4.2.3.3. Sizes of Primitive Types 492
4.2.3.4. ompd_callback_symbol_addr_fn_t 494

4.2.3.5. Accessing Memory in the OpenMP Program or Runtime 495
4.2.3.6. Data Format Conversion 497
4.2.3.7. ompd_callback_print_string_fn_t 499
4.2.3.8. The Callback Interface 499
4.2.4. OMPD Tool Interface Routines 501
4.2.4.1. Per OMPD Library Initialization and Finalization 501
4.2.4.2. Per OpenMP Process Initialization and Finalization 505
4.2.4.3. Thread and Signal Safety 507
4.2.4.4. Available Processors and Threads 508
4.2.4.5. Thread Handles 510
4.2.4.6. Parallel Region Handles 515
4.2.4.7. Task Handles 521
4.2.4.8. Display Control variables 543
4.2.5. Runtime Entry Points for OMPD 545
4.2.5.1. Event notification 545
4.2.6. Entry Points for OMPD defined in the OpenMP program 549
4.2.6.1. Enabling Support for OMPD at Runtime 550
4.3. Tool Foundation 551
4.3.1. Data Types 551
4.3.1.1. Thread States 551
4.3.1.2. Frames 556
4.3.1.3. Wait Identifiers 558
4.3.2. Global Symbols 558
4.3.2.1. **ompt_start_tool** 558
4.3.2.2. **ompd_dll_locations** 560
4.3.2.3. **ompd_dll_locations_valid** 561

5. Environment Variables **563**
5.1. **OMP_SCHEDULE** 564
5.2. **OMP_NUM_THREADS** 565
5.3. **OMP_DYNAMIC** 566
5.4. **OMP_PROC_BIND** 566
5.5. **OMP_PLACES** 567
5.6. **OMP_NESTED** 569

5.7. **OMP_STACKSIZE** 570
5.8. **OMP_WAIT_POLICY** 571
5.9. **OMP_MAX_ACTIVE_LEVELS** 572
5.10. **OMP_THREAD_LIMIT** 572
5.11. **OMP_CANCELLATION** 572
5.12. **OMP_DISPLAY_ENV** 573
5.13. **OMP_DISPLAY_AFFINITY** 574
5.14. **OMP_AFFINITY_FORMAT** 575
5.15. **OMP_DEFAULT_DEVICE** 577
5.16. **OMP_MAX_TASK_PRIORITY** 577
5.17. **OMP_TARGET_OFFLOAD** 578
5.18. **OMP_TOOL** 579
5.19. **OMP_TOOL_LIBRARIES** 579
5.20. **OMPD_ENABLED** 580
5.21. **OMP_ALLOCATOR** 580

A. Stubs for Runtime Library Routines **581**
A.1. C/C++ Stub Routines 582
A.2. Fortran Stub Routines 594

B. Interface Declarations **605**
B.1. Example of the **omp.h** Header File 606
B.2. Example of an Interface Declaration **include** File 611
B.3. Example of a Fortran Interface Declaration **module** 617
B.4. Example of a Generic Interface for a Library Routine 625

C. OpenMP Implementation-Defined Behaviors **627**

D. Task Frame Management for the Tool Interface **635**

E. Interaction Diagram of OMPD Components **637**

F. Features History **639**
F.1. Deprecated Features 639
F.2. Version 4.5 to 5.0 Differences 639
F.3. Version 4.0 to 4.5 Differences 642

F.4. Version 3.1 to 4.0 Differences 644
F.5. Version 3.0 to 3.1 Differences 645
F.6. Version 2.5 to 3.0 Differences 646

Index **649**

This page intentionally left blank

CHAPTER 1

Introduction

The collection of compiler directives, library routines, and environment variables described in this document collectively define the specification of the OpenMP Application Program Interface (OpenMP API) for parallelism in C, C++ and Fortran programs.

This specification provides a model for parallel programming that is portable across architectures from different vendors. Compilers from numerous vendors support the OpenMP API. More information about the OpenMP API can be found at the following web site

http://www.openmp.org

The directives, library routines, and environment variables defined in this document allow users to create and to 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, mapping 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.

1.1 Scope

The OpenMP API covers only user-directed parallelization, wherein the programmer explicitly specifies the actions to be taken by the compiler and runtime system in order to execute the program in parallel. OpenMP-compliant implementations are not required to check for data dependencies, data conflicts, race conditions, or deadlocks, any of which may occur in conforming programs. In addition, compliant implementations are not required to check for code sequences that cause a

program to be classified as non-conforming. Application developers are responsible for correctly using the OpenMP API to produce a conforming program. The OpenMP API does not cover compiler-generated automatic parallelization and directives to the compiler to assist such parallelization.

1.2 Glossary

1.2.1 Threading Concepts

thread An execution entity with a stack and associated static memory, called *threadprivate memory*.

OpenMP thread A *thread* that is managed by the OpenMP implementation.

idle thread An *OpenMP thread* that is not currently part of any **`parallel`** region.

thread-safe routine A routine that performs the intended function even when executed concurrently (by more than one *thread*).

processor Implementation defined hardware unit on which one or more *OpenMP threads* can execute.

device An implementation defined logical execution engine.

COMMENT: A *device* could have one or more *processors*.

host device The *device* on which the *OpenMP program* begins execution.

target device A device onto which code and data may be offloaded from the *host device*.

1.2.2 OpenMP Language Terminology

base language A programming language that serves as the foundation of the OpenMP specification.

COMMENT: See Section 1.7 on page 28 for a listing of current *base languages* for the OpenMP API.

base program A program written in a *base language*.

1 **program order** An ordering of operations performed by the same thread as determined by the
2 execution sequence of operations specified by the *base language*.

3 COMMENT: For C11 and C++11, *program order* corresponds to the
4 sequenced before relation between operations performed by the same
5 thread.

6 **structured block** For C/C++, an executable statement, possibly compound, with a single entry at the
7 top and a single exit at the bottom, or an OpenMP *construct*.

8 For Fortran, a block of executable statements with a single entry at the top and a
9 single exit at the bottom, or an OpenMP *construct*.

10 COMMENTS:

11 For all *base languages*:

12 • Access to the *structured block* must not be the result of a branch; and

13 • The point of exit cannot be a branch out of the *structured block*.

14 For C/C++:

15 • The point of entry must not be a call to **setjmp()**;

16 • **longjmp()** and **throw()** must not violate the entry/exit criteria;

17 • Calls to **exit()** are allowed in a *structured block*; and

18 • An expression statement, iteration statement, selection statement, or try
19 block is considered to be a *structured block* if the corresponding
20 compound statement obtained by enclosing it in **{** and **}** would be a
21 *structured block*.

22 For Fortran:

23 • **STOP** statements are allowed in a *structured block*.

24 **enclosing context** In C/C++, the innermost scope enclosing an OpenMP *directive*.

25 In Fortran, the innermost scoping unit enclosing an OpenMP *directive*.

26 **directive** In C/C++, a **#pragma**, and in Fortran, a comment, that specifies *OpenMP program*
27 behavior.

28 COMMENT: See Section 2.1 on page 36 for a description of OpenMP
29 *directive* syntax.

30 **white space** A non-empty sequence of space and/or horizontal tab characters.

31 **OpenMP program** A program that consists of a *base program*, annotated with OpenMP *directives* and
32 runtime library routines.

conforming program An *OpenMP program* that follows all rules and restrictions of the OpenMP specification.

declarative directive An OpenMP *directive* that may only be placed in a declarative context. A *declarative directive* results in one or more declarations only; it is not associated with the immediate execution of any user code.

executable directive An OpenMP *directive* that is not declarative. That is, it may be placed in an executable context.

stand-alone directive An OpenMP *executable directive* that has no associated executable user code.

construct An OpenMP *executable directive* (and for Fortran, the paired **`end`** *directive*, if any) and the associated statement, loop or *structured block*, if any, not including the code in any called routines. That is, the lexical extent of an *executable directive*.

combined construct A construct that is a shortcut for specifying one construct immediately nested inside another construct. A combined construct is semantically identical to that of explicitly specifying the first construct containing one instance of the second construct and no other statements.

composite construct A construct that is composed of two constructs but does not have identical semantics to specifying one of the constructs immediately nested inside the other. A composite construct either adds semantics not included in the constructs from which it is composed or the nesting of the one construct inside the other is not conforming.

region All code encountered during a specific instance of the execution of a given *construct* or of an OpenMP library routine. A *region* includes any code in called routines as well as any implicit code introduced by the OpenMP implementation. The generation of a *task* at the point where a *task generating construct* is encountered is a part of the *region* of the *encountering thread*, but an *explicit task region* associated with a *task generating construct* is not unless it is an *included task region*. The point where a **`target`** or **`teams`** directive is encountered is a part of the *region* of the *encountering thread*, but the *region* associated with the **`target`** or **`teams`** directive is not.

COMMENTS:

A *region* may also be thought of as the dynamic or runtime extent of a *construct* or of an OpenMP library routine.

During the execution of an *OpenMP program*, a *construct* may give rise to many *regions*.

active parallel region A **`parallel`** *region* that is executed by a *team* consisting of more than one *thread*.

inactive parallel region A **`parallel`** *region* that is executed by a *team* of only one *thread*.

sequential part All code encountered during the execution of an *initial task region* that is not part of a `parallel` *region* corresponding to a `parallel` *construct* or a `task` *region* corresponding to a `task` *construct*.

COMMENTS:

A *sequential part* is enclosed by an *implicit parallel region*.

Executable statements in called routines may be in both a *sequential part* and any number of explicit `parallel` *regions* at different points in the program execution.

master thread An *OpenMP thread* that has *thread* number 0. A *master thread* may be an *initial thread* or the *thread* that encounters a `parallel` *construct*, creates a *team*, generates a set of *implicit tasks*, and then executes one of those *tasks* as *thread* number 0.

parent thread The *thread* that encountered the `parallel` *construct* and generated a `parallel` *region* is the *parent thread* of each of the *threads* in the *team* of that `parallel` *region*. The *master thread* of a `parallel` *region* is the same *thread* as its *parent thread* with respect to any resources associated with an *OpenMP thread*.

child thread When a thread encounters a `parallel` construct, each of the threads in the generated `parallel` region's team are *child threads* of the encountering *thread*. The `target` or `teams` region's *initial thread* is not a *child thread* of the thread that encountered the `target` or `teams` construct.

ancestor thread For a given *thread*, its *parent thread* or one of its *parent thread's ancestor threads*.

descendent thread For a given *thread*, one of its *child threads* or one of its *child threads' descendent threads*.

team A set of one or more *threads* participating in the execution of a `parallel` *region*.

COMMENTS:

For an *active parallel region*, the team comprises the *master thread* and at least one additional *thread*.

For an *inactive parallel region*, the *team* comprises only the *master thread*.

league The set of *teams* created by a `teams` construct.

contention group An initial *thread* and its *descendent threads*.

implicit parallel region An *inactive parallel region* that is not generated from a `parallel` *construct*. *Implicit parallel regions* surround the whole *OpenMP program*, all `target` *regions*, and all `teams` *regions*.

initial thread A *thread* that executes an *implicit parallel region*.

initial team A *team* that comprises an *initial thread* executing an *implicit parallel region*.

nested construct A *construct* (lexically) enclosed by another *construct*.

closely nested construct A *construct* nested inside another *construct* with no other *construct* nested between them.

nested region A *region* (dynamically) enclosed by another *region*. That is, a *region* generated from the execution of another *region* or one of its *nested regions*.

COMMENT: Some nestings are *conforming* and some are not. See Section 2.22 on page 295 for the restrictions on nesting.

closely nested region A *region nested* inside another *region* with no **`parallel`** *region nested* between them.

strictly nested region A *region nested* inside another *region* with no other *region nested* between them.

all threads All OpenMP *threads* participating in the *OpenMP program*.

current team All *threads* in the *team* executing the innermost enclosing **`parallel`** *region*.

encountering thread For a given *region*, the *thread* that encounters the corresponding *construct*.

all tasks All *tasks* participating in the *OpenMP program*.

current team tasks All *tasks* encountered by the corresponding *team*. The *implicit tasks* constituting the **`parallel`** *region* and any *descendent tasks* encountered during the execution of these *implicit tasks* are included in this set of tasks.

generating task For a given *region*, the task for which execution by a *thread* generated the *region*.

binding thread set The set of *threads* that are affected by, or provide the context for, the execution of a *region*.

The *binding thread* set for a given *region* can be *all threads* on a *device*, *all threads* in a *contention group*, all *master threads* executing an enclosing **`teams`** *region*, the *current team*, or the *encountering thread*.

COMMENT: The *binding thread set* for a particular *region* is described in its corresponding subsection of this specification.

binding task set The set of *tasks* that are affected by, or provide the context for, the execution of a *region*.

The *binding task* set for a given *region* can be *all tasks*, the *current team tasks*, the *binding implicit task* or the *generating task*.

COMMENT: The *binding task* set for a particular *region* (if applicable) is described in its corresponding subsection of this specification.

1 **binding region** The enclosing *region* that determines the execution context and limits the scope of
2 the effects of the bound *region* is called the *binding region*.

3 *Binding region* is not defined for *regions* for which the *binding thread* set is *all*
4 *threads* or the *encountering thread*, nor is it defined for *regions* for which the *binding*
5 *task set* is *all tasks*.

6 COMMENTS:

7 The *binding region* for an **ordered** *region* is the innermost enclosing
8 *loop region*.

9 The *binding region* for a **taskwait** *region* is the innermost enclosing
10 *task region*.

11 The *binding region* for a **cancel** *region* is the innermost enclosing
12 *region* corresponding to the *construct-type-clause* of the **cancel**
13 construct.

14 The *binding region* for a **cancellation point** *region* is the
15 innermost enclosing *region* corresponding to the *construct-type-clause* of
16 the **cancellation point** construct.

17 For all other *regions* for which the *binding thread set* is the *current team*
18 or the *binding task set* is the *current team tasks*, the *binding region* is the
19 innermost enclosing **parallel** *region*.

20 For *regions* for which the *binding task set* is the *generating task*, the
21 *binding region* is the *region* of the *generating task*.

22 A **parallel** *region* need not be *active* nor explicit to be a *binding*
23 *region*.

24 A *task region* need not be explicit to be a *binding region*.

25 A *region* never binds to any *region* outside of the innermost enclosing
26 **parallel** *region*.

27 **orphaned construct** A *construct* that gives rise to a *region* for which the *binding thread set* is the *current*
28 *team*, but is not nested within another *construct* giving rise to the *binding region*.

29 **worksharing construct** A *construct* that defines units of work, each of which is executed exactly once by one
30 of the *threads* in the *team* executing the *construct*.

31 For C/C++, *worksharing constructs* are **for**, **sections**, and **single**.

32 For Fortran, *worksharing constructs* are **do**, **sections**, **single** and
33 **workshare**.

place Unordered set of *processors* on a device that is treated by the execution environment as a location unit when dealing with OpenMP thread affinity.

place list The ordered list that describes all OpenMP *places* available to the execution environment.

place partition An ordered list that corresponds to a contiguous interval in the OpenMP *place list*. It describes the *places* currently available to the execution environment for a given parallel *region*.

place number A number that uniquely identifies a *place* in the *place list*, with zero identifying the first *place* in the *place list*, and each consecutive whole number identifying the next *place* in the *place list*.

SIMD instruction A single machine instruction that can operate on multiple data elements.

SIMD lane A software or hardware mechanism capable of processing one data element from a *SIMD instruction*.

SIMD chunk A set of iterations executed concurrently, each by a *SIMD lane*, by a single *thread* by means of *SIMD instructions*.

memory allocator An OpenMP object that fulfills requests to allocate and deallocate storage for program variables.

1.2.3 Loop Terminology

loop directive An OpenMP *executable* directive for which the associated user code must be a loop nest that is a *structured block*.

associated loop(s) The loop(s) controlled by a *loop directive*.

COMMENT: If the *loop directive* contains a `collapse` or an `ordered(`*n*`)` clause then it may have more than one *associated loop*.

sequential loop A loop that is not associated with any OpenMP *loop directive*.

SIMD loop A loop that includes at least one *SIMD chunk*.

doacross loop nest A loop nest that has cross-iteration dependence. An iteration is dependent on one or more lexicographically earlier iterations.

COMMENT: The `ordered` clause parameter on a loop directive identifies the loop(s) associated with the *doacross loop nest*.

1.2.4 Synchronization Terminology

barrier A point in the execution of a program encountered by a *team* of *threads*, beyond which no *thread* in the team may execute until all *threads* in the *team* have reached the barrier and all *explicit tasks* generated by the *team* have executed to completion. If *cancellation* has been requested, threads may proceed to the end of the canceled *region* even if some threads in the team have not reached the *barrier*.

cancellation An action that cancels (that is, aborts) an OpenMP *region* and causes executing *implicit* or *explicit* tasks to proceed to the end of the canceled *region*.

cancellation point A point at which implicit and explicit tasks check if cancellation has been requested. If cancellation has been observed, they perform the *cancellation*.

COMMENT: For a list of cancellation points, see Section 2.19.1 on page 232

flush An operation, applied to a set of variables, that a *thread* performs to enforce consistency between its view and other *threads*' view of memory.

flush-set The set of variables to which a given *flush* operation is applied.

flush property Properties that determine the manner in which a *flush* operation enforces memory consistency. These properties are:

- *strong*: guarantees a completion ordering between memory operations on different threads upon which all threads agree;
- *write*: makes modifications by the executing thread visible to other threads;
- *read*: makes modifications by other threads visible to the executing thread;
- *synchronizable*: allows the *flush* operation to synchronize with another *flush* operation.

COMMENT: A given *flush* operation can have one or more *flush properties*, but must have at least the write or read flush property.

strong flush A *flush* operation that has the *strong flush property*.

weak flush A *flush* operation that does not have the *strong flush property*.

write flush A *flush* operation that has the *write flush property*.

read flush A *flush* operation that has the *read flush property*.

synchronizable flush A *flush* operation that has the *synchronizable flush property*.

non-synchronizable flush A *flush* operation that does not have the *synchronizable flush property*.

release flush A *write flush* that is a *synchronizable flush*.

acquire flush A *read flush* that is a *synchronizable flush*.

sync-set The set of variables associated with a *release flush* or *acquire flush* that a program may use to force a *release flush* to synchronize with an *acquire flush*.

1.2.5 Tasking Terminology

task A specific instance of executable code and its data environment that the OpenMP implementation can schedule for execution by threads.

task region A *region* consisting of all code encountered during the execution of a *task*.

> COMMENT: A **`parallel`** *region* consists of one or more implicit *task regions*.

implicit task A *task* generated by an *implicit parallel region* or generated when a **`parallel`** *construct* is encountered during execution.

binding implicit task The *implicit task* of the current thread team assigned to the encountering thread.

explicit task A *task* that is not an *implicit task*.

initial task An *implicit task* associated with an *implicit parallel region*.

current task For a given *thread*, the *task* corresponding to the *task region* in which it is executing.

child task A *task* is a *child task* of its generating *task region*. A *child task region* is not part of its generating *task region*.

sibling tasks *Tasks* that are *child tasks* of the same *task region*.

descendent task A *task* that is the *child task* of a *task region* or of one of its *descendent task regions*.

task completion *Task completion* occurs when the end of the *structured block* associated with the *construct* that generated the *task* is reached.

> COMMENT: Completion of the *initial task* that is generated when the program begins occurs at program exit.

task scheduling point A point during the execution of the current *task region* at which it can be suspended to be resumed later; or the point of *task completion*, after which the executing thread may switch to a different *task region*.

> COMMENT: For a list of *task scheduling points*, see Section 2.13.6 on page 125.

task switching The act of a *thread* switching from the execution of one *task* to another *task*.

1 **tied task** A *task* that, when its *task region* is suspended, can be resumed only by the same
2 *thread* that suspended it. That is, the *task* is tied to that *thread*.

3 **untied task** A *task* that, when its *task region* is suspended, can be resumed by any *thread* in the
4 team. That is, the *task* is not tied to any *thread*.

5 **undeferred task** A *task* for which execution is not deferred with respect to its generating *task region*.
6 That is, its generating *task region* is suspended until execution of the *undeferred task*
7 is completed.

8 **included task** A *task* for which execution is sequentially included in the generating *task region*.
9 That is, an *included task* is *undeferred* and executed immediately by the *encountering*
10 *thread*.

11 **merged task** A *task* for which the *data environment*, inclusive of ICVs, is the same as that of its
12 generating *task region*.

13 **mergeable task** A *task* that may be a *merged task* if it is an *undeferred task* or an *included task*.

14 **final task** A *task* that forces all of its *child tasks* to become *final* and *included tasks*.

15 **task dependence** An ordering relation between two *sibling tasks*: the *dependent task* and a previously
16 generated *predecessor task*. The *task dependence* is fulfilled when the *predecessor*
17 *task* has completed.

18 **dependent task** A *task* that because of a *task dependence* cannot be executed until its *predecessor*
19 *tasks* have completed.

20 **mutually exclusive tasks** *Tasks* that may be executed in any order, but not at the same time.

21 **predecessor task** A *task* that must complete before its *dependent tasks* can be executed.

22 **task synchronization construct** A **`taskwait`**, **`taskgroup`**, or a **`barrier`** *construct*.

23 **task generating construct** A *construct* that generates one or more *explicit tasks*.

24 **target task** A *mergeable* and *untied task* that is generated by a **`target`**, **`target enter`**
25 **`data`**, **`target exit data`**, or **`target update`** *construct*.

26 **taskgroup set** A set of tasks that are logically grouped by a **`taskgroup`** *region*.

1.2.6 Data Terminology

variable A named data storage block, for which the value can be defined and redefined during the execution of a program.

Note – An array or structure element is a variable that is part of another variable.

scalar variable For C/C++: A scalar variable, as defined by the base language.

For Fortran: A scalar variable with intrinsic type, as defined by the base language, excluding character type.

array section A designated subset of the elements of an array.

array item An array, an array section, or an array element.

base expression For C/C++: The expression in an array section or an array element that specifies the address of the initial element of the original array.

named array For C/C++: An expression that is an array but not an array element and appears as the array referred to by a given array item.

For Fortran: A variable that is an array and appears as the array referred to by a given array item.

named pointer For C/C++: An lvalue expression that is a pointer and appears as a pointer to the array implicitly referred to by a given array item.

For Fortran: A variable that has the **POINTER** attribute and appears as a pointer to the array to which a given array item implicitly refers.

Note – A given array item cannot have a *named pointer* if it has a *named array*.

attached pointer A pointer variable in a device data environment to which the effect of a **map** clause assigns the address of an array section. The pointer is an attached pointer for the remainder of its lifetime in the device data environment.

simply contiguous array section An array section that statically can be determined to have contiguous storage.

structure A structure is a variable that contains one or more variables.

For C/C++: Implemented using struct types.

For C++: Implemented using class types.

For Fortran: Implemented using derived types.

private variable With respect to a given set of *task regions* or *SIMD lanes* that bind to the same **`parallel`** *region*, a *variable* for which the name provides access to a different block of storage for each *task region* or *SIMD lane*.

A *variable* that is part of another variable (as an array or structure element) cannot be made private independently of other components.

shared variable With respect to a given set of *task regions* that bind to the same **`parallel`** *region*, a *variable* for which the name provides access to the same block of storage for each *task region*.

A *variable* that is part of another variable (as an array or structure element) cannot be *shared* independently of the other components, except for static data members of C++ classes.

threadprivate variable A *variable* that is replicated, one instance per *thread*, by the OpenMP implementation. Its name then provides access to a different block of storage for each *thread*.

A *variable* that is part of another variable (as an array or structure element) cannot be made *threadprivate* independently of the other components, except for static data members of C++ classes.

threadprivate memory The set of *threadprivate variables* associated with each *thread*.

data environment The *variables* associated with the execution of a given *region*.

device data environment The initial *data environment* associated with a device.

device address An *implementation defined* reference to an address in a *device data environment*.

device pointer A *variable* that contains a *device address*.

mapped variable An original *variable* in a *data environment* with a corresponding *variable* in a device *data environment*.

COMMENT: The original and corresponding *variables* may share storage.

map-type decay The process used to determine the final map type used when mapping a variable with a user defined mapper. The combination of the two map types determines the final map type based on the following table.

	alloc	to	from	tofrom	release	delete
alloc	alloc	alloc	alloc	alloc	release	delete
to	alloc	to	alloc	to	release	delete
from	alloc	alloc	from	from	release	delete
tofrom	alloc	to	from	tofrom	release	delete

mappable type A type that is valid for a *mapped variable*. If a type is composed from other types (such as the type of an array or structure element) and any of the other types are not mappable then the type is not mappable.

> COMMENT: Pointer types are *mappable* but the memory block to which the pointer refers is not *mapped*.

For C: The type must be a complete type.

For C++: The type must be a complete type.

In addition, for class types:

- All member functions accessed in any **`target`** region must appear in a **`declare target`** directive.

For Fortran: No restrictions on the type except that for derived types:

- All type-bound procedures accessed in any target region must appear in a **`declare target`** directive.

defined For *variables*, the property of having a valid value.

For C: For the contents of *variables*, the property of having a valid value.

For C++: For the contents of *variables* of POD (plain old data) type, the property of having a valid value.

For *variables* of non-POD class type, the property of having been constructed but not subsequently destructed.

For Fortran: For the contents of *variables*, the property of having a valid value. For the allocation or association status of *variables*, the property of having a valid status.

> COMMENT: Programs that rely upon *variables* that are not *defined* are *non-conforming programs*.

class type For C++: *Variables* declared with one of the **`class`**, **`struct`**, or **`union`** keywords

sequentially consistent atomic construct An **`atomic`** construct for which the **`seq_cst`** clause is specified.

non-sequentially consistent atomic construct An **`atomic`** construct for which the **`seq_cst`** clause is not specified

1.2.7 Implementation Terminology

supporting *n* levels of parallelism Implies allowing an *active parallel region* to be enclosed by *n-1 active parallel regions*.

supporting the OpenMP API Supporting at least one level of parallelism.

supporting nested parallelism Supporting more than one level of parallelism.

internal control variable A conceptual variable that specifies runtime behavior of a set of *threads* or *tasks* in an *OpenMP program*.

COMMENT: The acronym ICV is used interchangeably with the term *internal control variable* in the remainder of this specification.

compliant implementation An implementation of the OpenMP specification that compiles and executes any *conforming program* as defined by the specification.

COMMENT: A *compliant implementation* may exhibit *unspecified behavior* when compiling or executing a *non-conforming program*.

unspecified behavior A behavior or result that is not specified by the OpenMP specification or not known prior to the compilation or execution of an *OpenMP program*.

Such *unspecified behavior* may result from:

- Issues documented by the OpenMP specification as having *unspecified behavior*.
- A *non-conforming program*.
- A *conforming program* exhibiting an *implementation defined* behavior.

implementation defined Behavior that must be documented by the implementation, and is allowed to vary among different *compliant implementations*. An implementation is allowed to define this behavior as *unspecified*.

COMMENT: All features that have *implementation defined* behavior are documented in Appendix C.

deprecated Implies a construct, clause, or other feature is normative in the current specification but is considered obsolescent and will be removed in the future.

1
1.2.8 Tool Terminology

2 **tool** Executable code, distinct from application or runtime code, that can observe and/or
3 modify the execution of an application.

4 **first-party tool** A tool that executes in the address space of the program it is monitoring.

5 **third-party tool** A tool that executes as a separate process from that which it is monitoring and
6 potentially controlling.

7 **activated tool** A first-party tool that successfully completed its initialization.

8 **event** A point of interest in the execution of a thread where the condition defining that event
9 is true.

10 **tool callback** A function provided by a tool to an OpenMP implementation that can be invoked
11 when needed.

12 **registering a callback** Providing a callback function to an OpenMP implementation for a particular purpose.

13 **dispatching a callback**
14 **at an event** Processing a callback when an associated event occurs in a manner consistent with
the return code provided when a *first-party* tool registered the callback.

15 **thread state** An enumeration type that describes what an OpenMP thread is currently doing. A
16 thread can be in only one state at any time.

17 **wait identifier** A unique opaque handle associated with each data object (e.g., a lock) used by the
18 OpenMP runtime to enforce mutual exclusion that may cause a thread to wait actively
19 or passively.

20 **frame** A storage area on a thread's stack associated with a procedure invocation. A frame
21 includes space for one or more saved registers and often also includes space for saved
22 arguments, local variables, and padding for alignment.

23 **canonical frame**
24 **address** An address associated with a procedure *frame* on a call stack defined as the value of
the stack pointer immediately prior to calling the procedure whose invocation the
25 frame represents.

26 **runtime entry point** A function interface provided by an OpenMP runtime for use by a tool. A runtime
27 entry point is typically not associated with a global function symbol.

28 **trace record** A data structure to store information associated with an occurrence of an *event*.

29 **native trace record** A *trace record* for an OpenMP device that is in a device-specific format.

30 **signal** A software interrupt delivered to a thread.

31 **signal handler** A function called asynchronously when a *signal* is delivered to a thread.

async signal safe Guaranteed not to interfere with operations that are being interrupted by *signal* delivery. An async signal safe *runtime entry point* is safe to call from a *signal handler*.

code block A contiguous region of memory that contains code of an OpenMP program to be executed on a device.

OMPT An interface that helps a first-party tool monitor the execution of an OpenMP program.

OMPD An interface that helps a third-party tool inspect the OpenMP state of a program that has begun execution.

OMPD library A shared library that implements the OMPD interface.

image file An executable or shared library.

address space A collection of logical, virtual, or physical memory address ranges containing code, stack, and/or data. Address ranges within an address space need not be contiguous. An address space consists of one or more *segments*.

segment A region of an address space associated with a set of address ranges.

OpenMP architecture The architecture on which an OpenMP region executes.

tool architecture The architecture on which an OMPD tool executes.

OpenMP process A collection of one or more threads and address spaces. A process may contain threads and address spaces for multiple OpenMP architectures. At least one thread in an OpenMP process is an OpenMP thread. A process may be live or a core file.

handle An opaque reference provided by an OMPD library implementation to a using tool. A handle uniquely identifies an abstraction.

address space handle A handle that refers to an address space within an OpenMP process.

thread handle A handle that refers to an OpenMP thread.

parallel handle A handle that refers to an OpenMP parallel region.

task handle A handle that refers to an OpenMP task region.

tool context An opaque reference provided by a tool to an OMPD library implementation. A tool context uniquely identifies an abstraction.

address space context A tool context that refers to an address space within a process.

thread context A tool context that refers to a thread.

thread identifier An identifier for a native thread defined by a thread implementation.

1.3 Execution Model

The OpenMP API uses the fork-join model of parallel execution. Multiple threads of execution perform tasks defined implicitly or explicitly by OpenMP directives. The OpenMP API is intended to support programs that will execute correctly both as parallel programs (multiple threads of execution and a full OpenMP support library) and as sequential programs (directives ignored and a simple OpenMP stubs library). However, it is possible and permitted to develop a program that executes correctly as a parallel program but not as a sequential program, or that produces different results when executed as a parallel program compared to when it is executed as a sequential program. Furthermore, using different numbers of threads may result in different numeric results because of changes in the association of numeric operations. For example, a serial addition reduction may have a different pattern of addition associations than a parallel reduction. These different associations may change the results of floating-point addition.

An OpenMP program begins as a single thread of execution, called an initial thread. An initial thread executes sequentially, as if the code encountered is part of an implicit task region, called an initial task region, that is generated by the implicit parallel region surrounding the whole program.

The thread that executes the implicit parallel region that surrounds the whole program executes on the *host device*. An implementation may support other *target devices*. If supported, one or more devices are available to the host device for offloading code and data. Each device has its own threads that are distinct from threads that execute on another device. Threads cannot migrate from one device to another device. The execution model is host-centric such that the host device offloads **`target`** regions to target devices.

When a **`target`** construct is encountered, a new *target task* is generated. The *target task* region encloses the **`target`** region. The *target task* is complete after the execution of the **`target`** region is complete.

When a *target task* executes, the enclosed **`target`** region is executed by an initial thread. The initial thread may execute on a *target device*. The initial thread executes sequentially, as if the target region is part of an initial task region that is generated by an implicit parallel region. If the target device does not exist or the implementation does not support the target device, all **`target`** regions associated with that device execute on the host device.

The implementation must ensure that the **`target`** region executes as if it were executed in the data environment of the target device unless an **`if`** clause is present and the **`if`** clause expression evaluates to *false*.

The **`teams`** construct creates a *league of teams*, where each team is an initial team that comprises an initial thread that executes the **`teams`** region. Each initial thread executes sequentially, as if the code encountered is part of an initial task region that is generated by an implicit parallel region associated with each team.

If a construct creates a data environment, the data environment is created at the time the construct is encountered. Whether a construct creates a data environment is defined in the description of the

construct.

When any thread encounters a **parallel** construct, the thread creates a team of itself and zero or more additional threads and becomes the master of the new team. A set of implicit tasks, one per thread, is generated. The code for each task is defined by the code inside the **parallel** construct. Each task is assigned to a different thread in the team and becomes tied; that is, it is always executed by the thread to which it is initially assigned. The task region of the task being executed by the encountering thread is suspended, and each member of the new team executes its implicit task. There is an implicit barrier at the end of the **parallel** construct. Only the master thread resumes execution beyond the end of the **parallel** construct, resuming the task region that was suspended upon encountering the **parallel** construct. Any number of **parallel** constructs can be specified in a single program.

parallel regions may be arbitrarily nested inside each other. If nested parallelism is disabled, or is not supported by the OpenMP implementation, then the new team that is created by a thread encountering a **parallel** construct inside a **parallel** region will consist only of the encountering thread. However, if nested parallelism is supported and enabled, then the new team can consist of more than one thread. A **parallel** construct may include a **proc_bind** clause to specify the places to use for the threads in the team within the **parallel** region.

When any team encounters a worksharing construct, the work inside the construct is divided among the members of the team, and executed cooperatively instead of being executed by every thread. There is a default barrier at the end of each worksharing construct unless the **nowait** clause is present. Redundant execution of code by every thread in the team resumes after the end of the worksharing construct.

When any thread encounters a *task generating construct*, one or more explicit tasks are generated. Execution of explicitly generated tasks is assigned to one of the threads in the current team, subject to the thread's availability to execute work. Thus, execution of the new task could be immediate, or deferred until later according to task scheduling constraints and thread availability. Threads are allowed to suspend the current task region at a task scheduling point in order to execute a different task. If the suspended task region is for a tied task, the initially assigned thread later resumes execution of the suspended task region. If the suspended task region is for an untied task, then any thread may resume its execution. Completion of all explicit tasks bound to a given parallel region is guaranteed before the master thread leaves the implicit barrier at the end of the region. Completion of a subset of all explicit tasks bound to a given parallel region may be specified through the use of task synchronization constructs. Completion of all explicit tasks bound to the implicit parallel region is guaranteed by the time the program exits.

When any thread encounters a **simd** construct, the iterations of the loop associated with the construct may be executed concurrently using the SIMD lanes that are available to the thread.

When a thread encounters a **concurrent** construct, an implementation may choose to apply optimizations, parallelization, and/or vectorization, including scheduling loop iterations in ways that cannot be directly represented in OpenMP. When a **concurrent** construct is encountered, an implementation must guarantee that the logical iterations of the associated loops are executed

exactly once. When encountering a **concurrent** construct an implementation will not generate a new league of thread teams, but may use any thread teams currently available on which to execute.

The **cancel** construct can alter the previously described flow of execution in an OpenMP region. The effect of the **cancel** construct depends on its *construct-type-clause*. If a task encounters a **cancel** construct with a **taskgroup** *construct-type-clause*, then the task activates cancellation and continues execution at the end of its **task** region, which implies completion of that task. Any other task in that **taskgroup** that has begun executing completes execution unless it encounters a **cancellation point** construct, in which case it continues execution at the end of its **task** region, which implies its completion. Other tasks in that **taskgroup** region that have not begun execution are aborted, which implies their completion.

For all other *construct-type-clause* values, if a thread encounters a **cancel** construct, it activates cancellation of the innermost enclosing region of the type specified and the thread continues execution at the end of that region. Threads check if cancellation has been activated for their region at cancellation points and, if so, also resume execution at the end of the canceled region.

If cancellation has been activated regardless of *construct-type-clause*, threads that are waiting inside a barrier other than an implicit barrier at the end of the canceled region exit the barrier and resume execution at the end of the canceled region. This action can occur before the other threads reach that barrier.

Synchronization constructs and library routines are available in the OpenMP API to coordinate tasks and data access in **parallel** regions. In addition, library routines and environment variables are available to control or to query the runtime environment of OpenMP programs.

The OpenMP specification makes no guarantee that input or output to the same file is synchronous when executed in parallel. In this case, the programmer is responsible for synchronizing input and output statements (or routines) using the provided synchronization constructs or library routines. For the case where each thread accesses a different file, no synchronization by the programmer is necessary.

1.4 Memory Model

1.4.1 Structure of the OpenMP Memory Model

The OpenMP API provides a relaxed-consistency, shared-memory model. All OpenMP threads have access to a place to store and to retrieve variables, called the *memory*. In addition, each thread is allowed to have its own *temporary view* of the memory. The temporary view of memory for each thread is not a required part of the OpenMP memory model, but can represent any kind of intervening structure, such as machine registers, cache, or other local storage, between the thread

and the memory. The temporary view of memory allows the thread to cache variables and thereby to avoid going to memory for every reference to a variable. Each thread also has access to another type of memory that must not be accessed by other threads, called *threadprivate memory*.

A directive that accepts data-sharing attribute clauses determines two kinds of access to variables used in the directive's associated structured block: shared and private. Each variable referenced in the structured block has an original variable, which is the variable by the same name that exists in the program immediately outside the construct. Each reference to a shared variable in the structured block becomes a reference to the original variable. For each private variable referenced in the structured block, a new version of the original variable (of the same type and size) is created in memory for each task or SIMD lane that contains code associated with the directive. Creation of the new version does not alter the value of the original variable. However, the impact of attempts to access the original variable during the region associated with the directive is unspecified; see Section 2.20.3.3 on page 252 for additional details. References to a private variable in the structured block refer to the private version of the original variable for the current task or SIMD lane. The relationship between the value of the original variable and the initial or final value of the private version depends on the exact clause that specifies it. Details of this issue, as well as other issues with privatization, are provided in Section 2.20 on page 239.

The minimum size at which a memory update may also read and write back adjacent variables that are part of another variable (as array or structure elements) is implementation defined but is no larger than required by the base language.

A single access to a variable may be implemented with multiple load or store instructions, and hence is not guaranteed to be atomic with respect to other accesses to the same variable. Accesses to variables smaller than the implementation defined minimum size or to C or C++ bit-fields may be implemented by reading, modifying, and rewriting a larger unit of memory, and may thus interfere with updates of variables or fields in the same unit of memory.

If multiple threads write without synchronization to the same memory unit, including cases due to atomicity considerations as described above, then a data race occurs. Similarly, if at least one thread reads from a memory unit and at least one thread writes without synchronization to that same memory unit, including cases due to atomicity considerations as described above, then a data race occurs. If a data race occurs then the result of the program is unspecified.

Every variable has an associated *modification order*, defined as a sequential ordering of all operations that store a value into the variable without causing a data race to occur.

A private variable in a task region that eventually generates an inner nested **`parallel`** region is permitted to be made shared by implicit tasks in the inner **`parallel`** region. A private variable in a task region can be shared by an explicit task region generated during its execution. However, it is the programmer's responsibility to ensure through synchronization that the lifetime of the variable does not end before completion of the explicit task region sharing it. Any other access by one task to the private variables of another task results in unspecified behavior.

1.4.2 Device Data Environments

When an OpenMP program begins, an implicit **target data** region for each device surrounds the whole program. Each device has a device data environment that is defined by its implicit **target data** region. Any **declare target** directives and the directives that accept data-mapping attribute clauses determine how an original variable in a data environment is mapped to a corresponding variable in a device data environment.

When an original variable is mapped to a device data environment and the associated corresponding variable is not present in the device data environment, a new corresponding variable (of the same type and size as the original variable) is created in the device data environment. The initial value of the new corresponding variable is determined from the clauses and the data environment of the encountering thread.

The corresponding variable in the device data environment may share storage with the original variable. Writes to the corresponding variable may alter the value of the original variable. The impact of this on memory consistency is discussed in Section 1.4.6 on page 25. When a task executes in the context of a device data environment, references to the original variable refer to the corresponding variable in the device data environment. If a corresponding variable does not exist in the device data environment then accesses to the original variable result in unspecified behavior unless the unified_shared_memory requirement is specified.

The relationship between the value of the original variable and the initial or final value of the corresponding variable depends on the *map-type*. Details of this issue, as well as other issues with mapping a variable, are provided in Section 2.20.6.1 on page 280.

The original variable in a data environment and the corresponding variable(s) in one or more device data environments may share storage. Without intervening synchronization data races can occur.

1.4.3 Memory Management

The host device, and target devices that an implementation may support, have attached storage resources where program variables are stored. These resources can be of different kinds.

An OpenMP program can use a memory allocator to allocate storage for its variables. Memory allocators are associated with certain storage resources and use that storage to allocate variables. Memory allocators are also used to deallocate variables and free the storage in the resources. When an OpenMP memory allocator is not used variables can be allocated in any storage resource.

1.4.4 The Flush Operation

The memory model has relaxed-consistency because a thread's temporary view of memory is not required to be consistent with memory at all times. A value written to a variable can remain in the

thread's temporary view until it is forced to memory at a later time. Likewise, a read from a variable may retrieve the value from the thread's temporary view, unless it is forced to read from memory. The OpenMP flush operation enforces consistency between the temporary view and memory.

The flush operation is applied to a set of variables called the *flush-set*. It also has an associated *sync-set* containing a set of synchronization variables that may be used to force synchronization between the flush operation and another flush operation. A flush operation must be a write flush, a read flush, or both a write flush and a read flush; accordingly, it restricts reordering of memory operations performed by the same thread that an implementation might otherwise do as follows:

- With respect to a write flush, an implementation must not reorder any prior memory operation in the thread's program order that reads or writes to a variable in its flush-set or any subsequent memory operation in the thread's program order that writes to a variable in its sync-set.
- With respect to a read flush, an implementation must not reorder any subsequent memory operation in the thread's program order that reads or writes to a variable in its flush-set or any prior memory operation in the thread's program order that reads from a variable in its sync-set.
- With respect to a read flush, an implementation must not reorder any subsequent write flush in the thread's program order that is applied to the same variable.

A strong flush imposes stronger ordering requirements across threads compared to a weak flush, as described in Section 1.4.6 on page 25.

If a thread has performed a write to its temporary view of a shared variable since its last write flush of that variable, then when it executes another write flush of the variable, the flush does not complete until the value of the variable has been written to the variable in memory. If a thread performs multiple writes to the same variable between two write flushes of that variable, the write flush ensures that it is the value of the last write is written to the variable in memory. The completion of a write flush of a set of variables executed by a thread is defined as the point at which all writes to those variables performed by the thread before the flush are visible in memory.

If a thread has not performed a write to its temporary view of a shared variable since its last write flush of that variable, a read flush of the variable executed by a thread causes its temporary view of the variable to be discarded. If its next memory operation following the read flush for that variable is a read, then the thread will read from memory when it may again capture the value in the temporary view. When a thread executes a read flush, no later memory operation by that thread for a variable involved in that flush is allowed to start until the flush completes. The completion of a read flush of a set of variables executed by a thread is defined as the point at which the thread's temporary view of all variables involved is discarded.

Flush operations enable a program to provide a guarantee of consistency between a thread's temporary view and memory. Therefore, the flush operation can be used to guarantee that a value written to a variable by one thread may be read by a second thread. To accomplish this, it is sufficient for the programmer to ensure that the second thread has not written to the variable since its last flush of the variable, and that the following sequence of events are completed in the specified order according to the completion order guarantees defined in Section 1.4.6 on page 25:

1. The value is written to the variable by the first thread.
2. The variable is flushed, with a write flush, by the first thread.
3. The variable is flushed, with a read flush, by the second thread.
4. The value is read from the variable by the second thread.

Note – OpenMP synchronization operations, described in Section 2.18 on page 193 and in Section 3.3 on page 344, are recommended for enforcing this order. Synchronization through variables, described in Section 1.4.5 on page 24, is possible but is not recommended because the proper timing of flushes is difficult.

1.4.5 Flush Synchronization and Happens Before

A release flush is a write flush that is synchronizable, and likewise an acquire flush is a read flush that is synchronizable. A release flush may synchronize with an acquire flush using a variable that is in the sync-sets of both flushes.

For every variable in the sync-set of a release flush, there exists a *release flush sequence* that is a sequence of consecutive modifications to the variable taken from its modification order. It is defined to be the maximal such sequence that is headed by a modification that follows the release flush in its thread's program order and contains only subsequent modifications performed on the same thread or read-modify-write atomic modifications performed by a different thread. A release flush synchronizes with an acquire flush on a different thread if there exists a variable in the sync-sets of both flushes and the value written to it by some modification in a release flush sequence associated with the release flush is read by an access to the variable preceding the acquire flush in its thread's program order.

Note – The conditions under which a release flush synchronizes with an acquire flush are intended to match the conditions under which release operations synchronize with acquire operations in C++11 and C11. A read-modify-write atomic modification includes any atomic operation specified with an **atomic** construct on which neither the **read** or **write** clause appears.

An operation *X* *happens before* an operation *Y* if any of the following conditions are satisfied:

1. *X* happens before *Y* according to the base language's definition of happens before, if such a definition exists.
2. *X* and *Y* are performed by the same thread, and *X* precedes *Y* in the thread's program order.
3. *X* is a release flush, *Y* is an acquire flush, and *X* synchronizes with *Y* according to the flush synchronization conditions explained above.
4. There exists another operation *Z*, such that *X* happens before *Z* and *Z* happens before *Y*.

A variable with an initial value is treated as if the value is stored to the variable by an operation that happens before all operations that access or modify the variable in the program.

1.4.6 OpenMP Memory Consistency

Given the conditions in Section 1.4.5 on page 24 for flush synchronization and happens before, OpenMP guarantees the following in accordance with the restrictions in Section 1.4.4 on page 22 on reordering with respect to flush operations:

- If the intersection of the flush-sets of two strong flushes performed by two different threads is non-empty, then the two flushes must be completed as if in some sequential order, seen by all threads.
- If two operations performed by the same thread access, modify, or, with a strong flush, flush the same variable, then they must be completed as if in that thread's program order, as seen by all threads.
- If two operations access, modify, or flush the same variable and one happens before the other, then they must be completed in that happens before order, as seen by the thread or threads that perform the operations.
- Any two atomic memory operations from different **atomic** regions must be completed as if in the same order as the strong flushes implied in their respective regions, as seen by all threads.

The flush operation can be specified using the **flush** directive, and is also implied at various locations in an OpenMP program: see Section 2.18.8 on page 215 for details.

Note – Since flush operations by themselves cannot prevent data races, explicit flush operations are only useful in combination with non-sequentially consistent atomic directives.

OpenMP programs that:

- do not use non-sequentially consistent atomic directives,
- do not rely on the accuracy of a *false* result from **omp_test_lock** and **omp_test_nest_lock**, and
- correctly avoid data races as required in Section 1.4.1 on page 20

behave as though operations on shared variables were simply interleaved in an order consistent with the order in which they are performed by each thread. The relaxed consistency model is invisible for such programs, and any explicit flush operations in such programs are redundant.

Implementations are allowed to relax the ordering imposed by implicit flush operations when the result is only visible to programs using non-sequentially consistent atomic directives.

1.5 Tool Interface

To enable development of high-quality, portable, tools that support monitoring, performance, or correctness analysis and debugging of OpenMP programs developed using any implementation of the OpenMP API, the OpenMP API includes two tool interfaces: OMPT and OMPD.

1.5.1 OMPT

The OMPT interface, which is intended for *first-party* tools, provides the following:

- a mechanism to initialize a first-party tool,
- routines that enable a tool to determine the capabilities of an OpenMP implementation,
- routines that enable a tool to examine OpenMP state information associated with a thread,
- mechanisms that enable a tool to map implementation-level calling contexts back to their source-level representations,
- a callback interface that enables a tool to receive notification of OpenMP *events*,

- a tracing interface that enables a tool to trace activity on OpenMP target devices, and
- a runtime library routine that an application can use to control a tool.

OpenMP implementations may differ with respect to the *thread states* that they support, the mutual exclusion implementations they employ, and the OpenMP events for which tool callbacks are invoked. For some OpenMP events, OpenMP implementations must guarantee that a registered callback will be invoked for each occurrence of the event. For other OpenMP events, OpenMP implementations are permitted to invoke a registered callback for some or no occurrences of the event; for such OpenMP events, however, OpenMP implementations are encouraged to invoke tool callbacks on as many occurrences of the event as is practical to do so. Section 4.1.1.3 specifies the subset of OMPT callbacks that an OpenMP implementation must support for a minimal implementation of the OMPT interface.

An implementation of the OpenMP API may differ from the abstract execution model described by its specification. The ability of tools using the OMPT interface to observe such differences does not constrain implementations of the OpenMP API in any way.

With the exception of the **omp_control_tool** runtime library routine for tool control, all other routines in the OMPT interface are intended for use only by tools and are not visible to applications. For that reason, a Fortran binding is provided only for **omp_control_tool**; all other OMPT functionality is described with C syntax only.

1.5.2 OMPD

The OMPD interface is intended for a *third-party* tool, which runs as a separate process. An OpenMP implementation must provide an OMPD shared library plugin that can be loaded and used by a third-party tool. A third-party tool, such as a debugger, uses the OMPD plugin library to access OpenMP state of a program that has begun execution. OMPD defines the following:

- an interface that an OMPD plugin shared library exports, which a tool can use to access OpenMP state of a program that has begun execution;
- a callback interface that a tool provides to the OMPD plugin so that the plugin can use it to access OpenMP state of a program that has begun execution; and
- a small number of symbols that must be defined by an OpenMP implementation to help the tool find the correct OMPD plugin to use for that OpenMP implementation and to facilitate notification of events.

OMPD is described in Chapter 4.

1.6 OpenMP Compliance

The OpenMP API defines constructs that operate in the context of the base language that is supported by an implementation. If the implementation of the base language does not support a language construct that appears in this document, a compliant OpenMP implementation is not required to support it, with the exception that for Fortran, the implementation must allow case insensitivity for directive and API routines names, and must allow identifiers of more than six characters. An implementation of the OpenMP API is compliant if and only if it compiles and executes all other conforming programs, and supports the tool interface, according to the syntax and semantics laid out in Chapters 1, 2, 3, 4 and 5. Appendices A, B, C, D, and E, as well as sections designated as Notes (see Section 1.8 on page 31) are for information purposes only and are not part of the specification.

All library, intrinsic and built-in routines provided by the base language must be thread-safe in a compliant implementation. In addition, the implementation of the base language must also be thread-safe. For example, **ALLOCATE** and **DEALLOCATE** statements must be thread-safe in Fortran. Unsynchronized concurrent use of such routines by different threads must produce correct results (although not necessarily the same as serial execution results, as in the case of random number generation routines).

Starting with Fortran 90, variables with explicit initialization have the **SAVE** attribute implicitly. This is not the case in Fortran 77. However, a compliant OpenMP Fortran implementation must give such a variable the **SAVE** attribute, regardless of the underlying base language version.

Appendix C lists certain aspects of the OpenMP API that are implementation defined. A compliant implementation is required to define and document its behavior for each of the items in Appendix C.

1.7 Normative References

- ISO/IEC 9899:1990, *Information Technology - Programming Languages - C*.

  This OpenMP API specification refers to ISO/IEC 9899:1990 as C90.

- ISO/IEC 9899:1999, *Information Technology - Programming Languages - C*.

  This OpenMP API specification refers to ISO/IEC 9899:1999 as C99.

- ISO/IEC 9899:2011, *Information Technology - Programming Languages - C*.

  This OpenMP API specification refers to ISO/IEC 9899:2011 as C11. The following features are not supported:

  - Supporting the noreturn property

- Adding alignment support
- Creation of complex value
- Abandoning a process (adding **quick_exit** and **at_quick_exit**)
- Threads for the C standard library
- Thread-local storage
- Parallel memory sequencing model
- Atomic

- ISO/IEC 14882:1998, *Information Technology - Programming Languages - C++*.

  This OpenMP API specification refers to ISO/IEC 14882:1998 as C++.

- ISO/IEC 14882:2011, *Information Technology - Programming Languages - C++*.

  This OpenMP API specification refers to ISO/IEC 14882:2011 as C++11. The following features are not supported:

  - Rvalue references
  - Variadic templates
  - Extending variadic template template parameter
  - Declared type of an expression
  - Incomplete return types
  - Default template arguments for function templates
  - Alias templates
  - Generalized constant expressions
  - Alignment support
  - Delegating constructors
  - New character types
  - Standard layout types
  - Defaulted functions
  - Local and unnamed types as template arguments
  - Range-based for
  - Explicit virtual overrides
  - Minimal support for garbage collection and reachability-based leak detection

- Allowing move constructs to throw
- Defining move special member functions
- Concurrency
- Data-dependency ordering: atomics and memory model
- Additions to the standard library
- Thread-local storage
- Dynamic initialization and destruction with concurrency
- C++11 library
- Long long support
- Extending integral types

• ISO/IEC 14882:2014, *Information Technology - Programming Languages - C++*.

This OpenMP API specification refers to ISO/IEC 14882:2014 as C++14. The following features are not supported:

- Initialized/Generalized lambda captures (init-capture)
- Variable templates
- Generic (polymorphic) lambda expressions
- Sized deallocation
- Null forward iterators
- Add make_unique
- constexpr library additions: functional
- What signal handlers can do
- Prohibiting "out of thin air"

• ISO/IEC 1539:1980, *Information Technology - Programming Languages - Fortran*.

This OpenMP API specification refers to ISO/IEC 1539:1980 as Fortran 77.

• ISO/IEC 1539:1991, *Information Technology - Programming Languages - Fortran*.

This OpenMP API specification refers to ISO/IEC 1539:1991 as Fortran 90.

• ISO/IEC 1539-1:1997, *Information Technology - Programming Languages - Fortran*.

This OpenMP API specification refers to ISO/IEC 1539-1:1997 as Fortran 95.

- ISO/IEC 1539-1:2004, *Information Technology - Programming Languages - Fortran*.

  This OpenMP API specification refers to ISO/IEC 1539-1:2004 as Fortran 2003. The following features are not supported:

  - IEEE Arithmetic issues covered in Fortran 2003 Section 14
  - Parameterized derived types
  - The **PASS** attribute
  - Procedures bound to a type as operators
  - Overriding a type-bound procedure
  - **SELECT TYPE** construct
  - Deferred bindings and abstract types
  - Controlling IEEE underflow
  - Another IEEE class value

Where this OpenMP API specification refers to C, C++ or Fortran, reference is made to the base language supported by the implementation.

1.8 Organization of this Document

The remainder of this document is structured as follows:

- Chapter 2 “Directives”
- Chapter 3 “Runtime Library Routines”
- Chapter 4 “Tool Interface”
- Chapter 5 “Environment Variables”
- Appendix A “Stubs for Runtime Library Routines”
- Appendix B “Interface Declarations”
- Appendix C “OpenMP Implementation-Defined Behaviors”
- Appendix D “Task Frame Management for the Tool Interface”
- Appendix F “Features History”

Some sections of this document only apply to programs written in a certain base language. Text that applies only to programs for which the base language is C or C++ is shown as follows:

C / C++

1 C/C++ specific text...

C / C++

2 Text that applies only to programs for which the base language is C only is shown as follows:

C

3 C specific text...

C

4 Text that applies only to programs for which the base language is C90 only is shown as follows:

C90

5 C90 specific text...

C90

6 Text that applies only to programs for which the base language is C99 only is shown as follows:

C99

7 C99 specific text...

C99

8 Text that applies only to programs for which the base language is C++ only is shown as follows:

C++

9 C++ specific text...

C++

10 Text that applies only to programs for which the base language is Fortran is shown as follows:

Fortran

11 Fortran specific text......

Fortran

12 Where an entire page consists of base language specific text, a marker is shown at the top of the
13 page. For Fortran-specific text, the marker is:

Fortran (cont.)

For C/C++-specific text, the marker is:

C/C++ (cont.)

Some text is for information only, and is not part of the normative specification. Such text is designated as a note, like this:

Note – Non-normative text...

This page intentionally left blank

CHAPTER 2

Directives

This chapter describes the syntax and behavior of OpenMP directives, and is divided into the following sections:

- The language-specific directive format (Section 2.1 on page 36)
- Mechanisms to control conditional compilation (Section 2.2 on page 44)
- Control of OpenMP API ICVs (Section 2.4 on page 49)
- How to specify and to use array sections for all base languages (Section 2.5 on page 60)
- Details of each OpenMP directive, including associated events and tool callbacks (Section 2.8 on page 66 to Section 2.22 on page 295)

C / C++

In C/C++, OpenMP directives are specified by using the **#pragma** mechanism provided by the C and C++ standards.

C / C++

Fortran

In Fortran, OpenMP directives are specified by using special comments that are identified by unique sentinels. Also, a special comment form is available for conditional compilation.

Fortran

Compilers can therefore ignore OpenMP directives and conditionally compiled code if support of the OpenMP API is not provided or enabled. A compliant implementation must provide an option or interface that ensures that underlying support of all OpenMP directives and OpenMP conditional compilation mechanisms is enabled. In the remainder of this document, the phrase *OpenMP compilation* is used to mean a compilation with these OpenMP features enabled.

Fortran

Restrictions

The following restriction applies to all OpenMP directives:

- OpenMP directives, except SIMD and **declare target** directives, may not appear in pure procedures.

Fortran

2.1 Directive Format

C / C++

OpenMP directives for C/C++ are specified with the **pragma** preprocessing directive. The syntax of an OpenMP directive is as follows:

```
#pragma omp directive-name [clause[ [,] clause] ... ] new-line
```

Each directive starts with **#pragma omp**. The remainder of the directive follows the conventions of the C and C++ standards for compiler directives. In particular, white space can be used before and after the **#**, and sometimes white space must be used to separate the words in a directive. Preprocessing tokens following the **#pragma omp** are subject to macro replacement.

Some OpenMP directives may be composed of consecutive **#pragma** preprocessing directives if specified in their syntax.

Directives are case-sensitive.

An OpenMP executable directive applies to at most one succeeding statement, which must be a structured block.

C / C++

Fortran

OpenMP directives for Fortran are specified as follows:

```
sentinel directive-name [clause[ [ , ] clause]...]
```

All OpenMP compiler directives must begin with a directive *sentinel*. The format of a sentinel differs between fixed and free-form source files, as described in Section 2.1.1 on page 39 and Section 2.1.2 on page 40.

Directives are case insensitive. Directives cannot be embedded within continued statements, and statements cannot be embedded within directives.

In order to simplify the presentation, free form is used for the syntax of OpenMP directives for Fortran in the remainder of this document, except as noted.

Fortran

Only one *directive-name* can be specified per directive (note that this includes combined directives, see Section 2.16 on page 169). The order in which clauses appear on directives is not significant. Clauses on directives may be repeated as needed, subject to the restrictions listed in the description of each clause.

Some data-sharing attribute clauses (Section 2.20.3 on page 249), data copying clauses (Section 2.20.5 on page 275), the **threadprivate** directive (Section 2.20.2 on page 244), the **flush** directive (Section 2.18.8 on page 215), and the **link** clause of the **declare target** directive (Section 2.15.7 on page 150) accept a *list*. The **to** clause of the **declare target** directive (Section 2.15.7 on page 150) accepts an *extended-list*. The **depend** clause (Section 2.18.10 on page 225), when used to specify task dependences, accepts a *locator-list*. A *list* consists of a comma-separated collection of one or more *list items*. A *extended-list* consists of a comma-separated collection of one or more *extended list items*. A *locator-list* consists of a comma-separated collection of one or more *locator list items*.

C / C++

A *list item* is a variable or array section. An *extended list item* is a *list item* or a function name. A *locator list item* is any *lvalue* expression, including variables, or an array section.

C / C++

Fortran

1 A *list item* is a variable, array section or common block name (enclosed in slashes). An *extended*
2 *list item* is a *list item* or a procedure name. A *locator list item* is a *list item*.

3 When a named common block appears in a *list*, it has the same meaning as if every explicit member
4 of the common block appeared in the list. An explicit member of a common block is a variable that
5 is named in a **COMMON** statement that specifies the common block name and is declared in the same
6 scoping unit in which the clause appears.

7 Although variables in common blocks can be accessed by use association or host association,
8 common block names cannot. As a result, a common block name specified in a data-sharing
9 attribute, a data copying or a data-mapping attribute clause must be declared to be a common block
10 in the same scoping unit in which the clause appears.

11 If a list item that appears in a directive or clause is an optional dummy argument that is not present,
12 the directive or clause for that list item is ignored.

13 If the variable referenced inside a construct is an optional dummy argument that is not present, any
14 explicitly determined, implicitly determined, or predetermined data-sharing and data-mapping
15 attribute rules for that variable are ignored. Otherwise, if the variable is an optional dummy
16 argument that is present, it is present inside the construct.

Fortran

17 For all base languages, a *list item* or an *extended list item* is subject to the restrictions specified in
18 Section 2.5 on page 60 and in each of the sections describing clauses and directives for which the
19 *list* or *extended-list* appears.

Fortran

2.1.1 Fixed Source Form Directives

The following sentinels are recognized in fixed form source files:

```
!$omp | c$omp | *$omp
```

Sentinels must start in column 1 and appear as a single word with no intervening characters. Fortran fixed form line length, white space, continuation, and column rules apply to the directive line. Initial directive lines must have a space or zero in column 6, and continuation directive lines must have a character other than a space or a zero in column 6.

Comments may appear on the same line as a directive. The exclamation point initiates a comment when it appears after column 6. The comment extends to the end of the source line and is ignored. If the first non-blank character after the directive sentinel of an initial or continuation directive line is an exclamation point, the line is ignored.

Note – in the following example, the three formats for specifying the directive are equivalent (the first line represents the position of the first 9 columns):

```
c23456789
!$omp parallel do shared(a,b,c)

c$omp parallel do
c$omp+shared(a,b,c)

c$omp paralleldoshared(a,b,c)
```

2.1.2 Free Source Form Directives

The following sentinel is recognized in free form source files:

```
!$omp
```

The sentinel can appear in any column as long as it is preceded only by white space (spaces and tab characters). It must appear as a single word with no intervening character. Fortran free form line length, white space, and continuation rules apply to the directive line. Initial directive lines must have a space after the sentinel. Continued directive lines must have an ampersand (**&**) as the last non-blank character on the line, prior to any comment placed inside the directive. Continuation directive lines can have an ampersand after the directive sentinel with optional white space before and after the ampersand.

Comments may appear on the same line as a directive. The exclamation point (**!**) initiates a comment. The comment extends to the end of the source line and is ignored. If the first non-blank character after the directive sentinel is an exclamation point, the line is ignored.

One or more blanks or horizontal tabs must be used to separate adjacent keywords in directives in free source form, except in the following cases, where white space is optional between the given set of keywords:

concurrent

declare reduction

declare simd

declare target

distribute parallel do

distribute parallel do simd

distribute simd

do simd

end atomic

end concurrent

end critical

end distribute

end distribute parallel do

```
1  end distribute parallel do simd
2  end distribute simd
3  end do
4  end do simd
5  end master
6  end ordered
7  end parallel
8  end parallel do
9  end parallel do simd
10 end parallel sections
11 end parallel workshare
12 end sections
13 end simd
14 end single
15 end target
16 end target data
17 end target parallel
18 end target parallel do
19 end target parallel do simd
20 end target simd
21 end target teams
22 end target teams distribute
23 end target teams distribute parallel do
24 end target teams distribute parallel do simd
25 end target teams distribute simd
26 end task
27 end taskgroup
```

```
1    end taskloop
2    end taskloop simd
3    end teams
4    end teams distribute
5    end teams distribute parallel do
6    end teams distribute parallel do simd
7    end teams distribute simd
8    end workshare
9    parallel do
10   parallel do simd
11   parallel sections
12   parallel workshare
13   target data
14   target enter data
15   target exit data
16   target parallel
17   target parallel do
18   target parallel do simd
19   target simd
20   target teams
21   target teams distribute
22   target teams distribute parallel do
23   target teams distribute parallel do simd
24   target teams distribute simd
25   target update
26   taskloop simd
27   teams distribute
```

```
teams distribute parallel do
teams distribute parallel do simd
teams distribute simd
concurrent
end concurrent
```

Note – in the following example the three formats for specifying the directive are equivalent (the first line represents the position of the first 9 columns):

```
!23456789
       !$omp parallel do &
                 !$omp shared(a,b,c)

       !$omp parallel &
      !$omp&do shared(a,b,c)

!$omp paralleldo shared(a,b,c)
```

Fortran

2.1.3 Stand-Alone Directives

Summary

Stand-alone directives are executable directives that have no associated user code.

Description

Stand-alone directives do not have any associated executable user code. Instead, they represent executable statements that typically do not have succinct equivalent statements in the base languages. There are some restrictions on the placement of a stand-alone directive within a program. A stand-alone directive may be placed only at a point where a base language executable statement is allowed.

Restrictions

C / C++

For C/C++, a stand-alone directive may not be used in place of the statement following an **if**, **while**, **do**, **switch**, or **label**.

C / C++

Fortran

For Fortran, a stand-alone directive may not be used as the action statement in an **if** statement or as the executable statement following a label if the label is referenced in the program.

Fortran

2.2 Conditional Compilation

In implementations that support a preprocessor, the **_OPENMP** macro name is defined to have the decimal value *yyyymm* where *yyyy* and *mm* are the year and month designations of the version of the OpenMP API that the implementation supports.

If this macro is the subject of a **#define** or a **#undef** preprocessing directive, the behavior is unspecified.

Fortran

The OpenMP API requires Fortran lines to be compiled conditionally, as described in the following sections.

2.2.1 Fixed Source Form Conditional Compilation Sentinels

The following conditional compilation sentinels are recognized in fixed form source files:

```
!$ | *$ | c$
```

To enable conditional compilation, a line with a conditional compilation sentinel must satisfy the following criteria:

- The sentinel must start in column 1 and appear as a single word with no intervening white space.
- After the sentinel is replaced with two spaces, initial lines must have a space or zero in column 6 and only white space and numbers in columns 1 through 5.
- After the sentinel is replaced with two spaces, continuation lines must have a character other than a space or zero in column 6 and only white space in columns 1 through 5.

If these criteria are met, the sentinel is replaced by two spaces. If these criteria are not met, the line is left unchanged.

Note – in the following example, the two forms for specifying conditional compilation in fixed source form are equivalent (the first line represents the position of the first 9 columns):

```
c23456789
!$ 10 iam = omp_get_thread_num() +
!$   &          index

#ifdef _OPENMP
   10 iam = omp_get_thread_num() +
     &              index
#endif
```

2.2.2 Free Source Form Conditional Compilation Sentinel

The following conditional compilation sentinel is recognized in free form source files:

```
!$
```

To enable conditional compilation, a line with a conditional compilation sentinel must satisfy the following criteria:

- The sentinel can appear in any column but must be preceded only by white space.
- The sentinel must appear as a single word with no intervening white space.
- Initial lines must have a space after the sentinel.
- Continued lines must have an ampersand as the last non-blank character on the line, prior to any comment appearing on the conditionally compiled line. Continuation lines can have an ampersand after the sentinel, with optional white space before and after the ampersand.

If these criteria are met, the sentinel is replaced by two spaces. If these criteria are not met, the line is left unchanged.

Note – in the following example, the two forms for specifying conditional compilation in free source form are equivalent (the first line represents the position of the first 9 columns):

```
c23456789
 !$ iam = omp_get_thread_num() +     &
 !$&    index

#ifdef _OPENMP
    iam = omp_get_thread_num() +     &
        index
#endif
```

Fortran

2.3 requires Directive

Summary

The **requires** directive specifies the features an implementation must provide in order for the code to compile and to execute correctly.

1 **Syntax**

C / C++

2 The syntax of the **requires** directive is as follows:

```
#pragma omp requires [clause[ [ , ] clause] ... ] new-line
```

C / C++

Fortran

3 The syntax of the **requires** directive is as follows:

```
!$omp requires [clause[ [ , ] clause] ... ]
```

Fortran

4 Where *clause* is either one of the requirement clauses listed below or a clause of the form
5 **ext_***implementation-defined-requirement* for an implementation defined requirement clause.

6 **unified_address**

7 **unified_shared_memory**

Description

The **requires** directive specifies features an implementation must support for correct execution of the current translation unit, or in Fortran the main program, subprogram or module. The behavior specified by a requirement clause may override the normal behavior specified elsewhere in this document.

Note – Use of this directive makes your code less portable. Users should be aware that not all devices or implementations support all requirements.

When the **unified_address** clause appears on a **requires** directive, the implementation guarantees that all devices accessible through OpenMP API routines and directives use a unified address space. In this address space, a pointer will always refer to the same location in memory from all devices accessible through OpenMP. The pointers returned by **omp_target_alloc** and accessed through **use_device_ptr** are guaranteed to be pointer values that can support pointer arithmetic while still being native device pointers. The **is_device_ptr** clause is not necessary for device pointers to be translated in **target** regions, and pointers found not present are not set to null but keep their original value. Memory local to a specific execution context may be exempt from this, following the restrictions of locality to a given execution context, thread or contention group. Target devices may still have discrete memories and dereferencing a device pointer on the host device remains unspecified behavior.

The **unified_shared_memory** clause implies the **unified_address** requirement, inheriting all of its behaviors. Additionally memory in the device data environment of any device visible to OpenMP, including but not limited to the host, is considered part of the device data environment of all devices accessible through OpenMP except as noted below. Every device address allocated through OpenMP device memory routines is a valid host pointer. Memory local to an execution context as defined in **unified_address** above may remain part of distinct device data environments as long as the execution context is local to the device containing that environment.

The **unified_shared_memory** clause makes the **map** clause optional on **target** constructs as well as the **declare target** directive on static lifetime variables accessed as part of **declare target** functions. Scalar variables are still made **firstprivate** by default for **target** regions. Values stored into memory by one device may not be visible to other devices until those two devices synchronize with each other or both synchronize with the host.

Implementers are allowed to include additional implementation defined requirement clauses. Requirement names that do not start with **ext_** are reserved. All implementation-defined requirements should begin with **ext_**.

Restrictions

The restrictions for the **requires** directive are as follows:

- In an application composed of multiple translation units or in Fortran more than one main program, subprogram or module all must have the same requirements.

2.4 Internal Control Variables

An OpenMP implementation must act as if there are internal control variables (ICVs) that control the behavior of an OpenMP program. These ICVs store information such as the number of threads to use for future **parallel** regions, the schedule to use for worksharing loops and whether nested parallelism is enabled or not. The ICVs are given values at various times (described below) during the execution of the program. They are initialized by the implementation itself and may be given values through OpenMP environment variables and through calls to OpenMP API routines. The program can retrieve the values of these ICVs only through OpenMP API routines.

For purposes of exposition, this document refers to the ICVs by certain names, but an implementation is not required to use these names or to offer any way to access the variables other than through the ways shown in Section 2.4.2 on page 51.

2.4.1 ICV Descriptions

The following ICVs store values that affect the operation of **parallel** regions.

- *dyn-var* - controls whether dynamic adjustment of the number of threads is enabled for encountered **parallel** regions. There is one copy of this ICV per data environment.
- *nest-var* - controls whether nested parallelism is enabled for encountered **parallel** regions. There is one copy of this ICV per data environment. The *nest-var* ICV has been deprecated.
- *nthreads-var* - controls the number of threads requested for encountered **parallel** regions. There is one copy of this ICV per data environment.
- *thread-limit-var* - controls the maximum number of threads participating in the contention group. There is one copy of this ICV per data environment.
- *max-active-levels-var* - controls the maximum number of nested active **parallel** regions. There is one copy of this ICV per device.

- *place-partition-var* – controls the place partition available to the execution environment for encountered **parallel** regions. There is one copy of this ICV per implicit task.
- *active-levels-var* - the number of nested, active parallel regions enclosing the current task such that all of the **parallel** regions are enclosed by the outermost initial task region on the current device. There is one copy of this ICV per data environment.
- *levels-var* - the number of nested parallel regions enclosing the current task such that all of the **parallel** regions are enclosed by the outermost initial task region on the current device. There is one copy of this ICV per data environment.
- *bind-var* - controls the binding of OpenMP threads to places. When binding is requested, the variable indicates that the execution environment is advised not to move threads between places. The variable can also provide default thread affinity policies. There is one copy of this ICV per data environment.

The following ICVs store values that affect the operation of loop regions.

- *run-sched-var* - controls the schedule that the **runtime** schedule clause uses for loop regions. There is one copy of this ICV per data environment.
- *def-sched-var* - controls the implementation defined default scheduling of loop regions. There is one copy of this ICV per device.

The following ICVs store values that affect program execution.

- *stacksize-var* - controls the stack size for threads that the OpenMP implementation creates. There is one copy of this ICV per device.
- *wait-policy-var* - controls the desired behavior of waiting threads. There is one copy of this ICV per device.
- *display-affinity-var* - controls whether to display thread affinity. There is one copy of this ICV for the whole program.
- *affinity-format-var* - controls the thread affinity format when displaying thread affinity. There is one copy of this ICV per device.
- *cancel-var* - controls the desired behavior of the **cancel** construct and cancellation points. There is one copy of this ICV for the whole program.
- *default-device-var* - controls the default target device. There is one copy of this ICV per data environment.
- *target-offload-var* - controls the offloading behavior. There is one copy of this ICV for the whole program.
- *max-task-priority-var* - controls the maximum priority value that can be specified in the **priority** clause of the **task** construct. There is one copy of this ICV for the whole program.

The following ICVs store values that affect the operation of the first-party tool interface.

- *tool-var* - determines whether an OpenMP implementation will try to register a tool. There is one copy of this ICV for the whole program.
- *tool-libraries-var* - specifies a list of absolute paths to tool libraries for OpenMP devices. There is one copy of this ICV for the whole program.

The following ICVs store values that relate to the operation of the OMPD tool interface.

- *ompd-tool-var* - determines whether an OpenMP implementation will collect information that an OMPD-plugin can access to satisfy requests from a tool. There is one copy of this ICV for the whole program.

The following ICVs store values that affect default memory allocation.

- *def-allocator-var* - determines the memory allocator to be used by memory allocation routines, directives and clauses when a memory allocator is not specified by the user. There is one copy of this ICV per implicit task.

2.4.2 ICV Initialization

Table 2.1 shows the ICVs, associated environment variables, and initial values.

TABLE 2.1: ICV Initial Values

ICV	Environment Variable	Initial value
dyn-var	**OMP_DYNAMIC**	See description below
nest-var	**OMP_NESTED**	Implementation defined
nthreads-var	**OMP_NUM_THREADS**	Implementation defined
run-sched-var	**OMP_SCHEDULE**	Implementation defined
def-sched-var	(none)	Implementation defined
bind-var	**OMP_PROC_BIND**	Implementation defined
stacksize-var	**OMP_STACKSIZE**	Implementation defined
wait-policy-var	**OMP_WAIT_POLICY**	Implementation defined
thread-limit-var	**OMP_THREAD_LIMIT**	Implementation defined

table continued on next page

table continued from previous page

ICV	Environment Variable	Initial value
max-active-levels-var	**OMP_MAX_ACTIVE_LEVELS**	See description below
active-levels-var	(none)	*zero*
levels-var	(none)	*zero*
place-partition-var	**OMP_PLACES**	Implementation defined
cancel-var	**OMP_CANCELLATION**	*false*
display-affinity-var	**OMP_DISPLAY_AFFINITY**	*false*
affinity-format-var	**OMP_AFFINITY_FORMAT**	Implementation defined
default-device-var	**OMP_DEFAULT_DEVICE**	Implementation defined
target-offload-var	**OMP_TARGET_OFFLOAD**	**DEFAULT**
max-task-priority-var	**OMP_MAX_TASK_PRIORITY**	*zero*
tool-var	**OMP_TOOL**	*enabled*
tool-libraries-var	**OMP_TOOL_LIBRARIES**	*empty string*
ompd-tool-var	**OMPD_ENABLED**	*not set*
def-allocator-var	**OMP_ALLOCATOR**	Implementation defined

1 **Description**

2 • Each device has its own ICVs.

3 • The value of the *nthreads-var* ICV is a list.

4 • The value of the *bind-var* ICV is a list.

5 • The initial value of *dyn-var* is implementation defined if the implementation supports dynamic
6 adjustment of the number of threads; otherwise, the initial value is *false*.

7 • The initial value of *max-active-levels-var* is the number of levels of parallelism that the
8 implementation supports. See the definition of *supporting n levels of parallelism* in Section 1.2.7
9 on page 15 for further details.

10 The host and target device ICVs are initialized before any OpenMP API construct or OpenMP API
11 routine executes. After the initial values are assigned, the values of any OpenMP environment
12 variables that were set by the user are read and the associated ICVs for the host device are modified
13 accordingly. The method for initializing a target device's ICVs is implementation defined.

Cross References

- **OMP_SCHEDULE** environment variable, see Section 5.1 on page 564.
- **OMP_NUM_THREADS** environment variable, see Section 5.2 on page 565.
- **OMP_DYNAMIC** environment variable, see Section 5.3 on page 566.
- **OMP_PROC_BIND** environment variable, see Section 5.4 on page 566.
- **OMP_PLACES** environment variable, see Section 5.5 on page 567.
- **OMP_NESTED** environment variable, see Section 5.6 on page 569.
- **OMP_STACKSIZE** environment variable, see Section 5.7 on page 570.
- **OMP_WAIT_POLICY** environment variable, see Section 5.8 on page 571.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 5.9 on page 572.
- **OMP_THREAD_LIMIT** environment variable, see Section 5.10 on page 572.
- **OMP_CANCELLATION** environment variable, see Section 5.11 on page 572.
- **OMP_DISPLAY_AFFINITY** environment variable, see Section 5.13 on page 574.
- **OMP_AFFINITY_FORMAT** environment variable, see Section 5.14 on page 575.
- **OMP_DEFAULT_DEVICE** environment variable, see Section 5.15 on page 577.
- **OMP_TARGET_OFFLOAD** environment variable, see Section 5.17 on page 578.
- **OMP_MAX_TASK_PRIORITY** environment variable, see Section 5.16 on page 577.
- **OMP_TOOL** environment variable, see Section 5.18 on page 579.
- **OMP_TOOL_LIBRARIES** environment variable, see Section 5.19 on page 579.
- **OMPD_ENABLED** environment variable, see Section 5.20 on page 580.
- **OMP_ALLOCATOR** environment variable, see Section 5.21 on page 580.

2.4.3 Modifying and Retrieving ICV Values

Table 2.2 shows the method for modifying and retrieving the values of ICVs through OpenMP API routines.

TABLE 2.2: Ways to Modify and to Retrieve ICV Values

ICV	Ways to modify value	Ways to retrieve value
dyn-var	**omp_set_dynamic()**	**omp_get_dynamic()**
nest-var	**omp_set_nested()**	**omp_get_nested()**
nthreads-var	**omp_set_num_threads()**	**omp_get_max_threads()**
run-sched-var	**omp_set_schedule()**	**omp_get_schedule()**
def-sched-var	(none)	(none)
bind-var	(none)	**omp_get_proc_bind()**
stacksize-var	(none)	(none)
wait-policy-var	(none)	(none)
thread-limit-var	**thread_limit** clause	**omp_get_thread_limit()**
max-active-levels-var	**omp_set_max_active_levels()**	**omp_get_max_active_levels()**
active-levels-var	(none)	**omp_get_active_level()**
levels-var	(none)	**omp_get_level()**
place-partition-var	(none)	See description below
cancel-var	(none)	**omp_get_cancellation()**
display-affinity-var	(none)	(none)
affinity-format-var	**omp_set_affinity_format()**	**omp_get_affinity_format()**
default-device-var	**omp_set_default_device()**	**omp_get_default_device()**
target-offload-var	(none)	(none)
max-task-priority-var	(none)	**omp_get_max_task_priority()**
tool-var	(none)	(none)
tool-libraries-var	(none)	(none)
ompd-tool-var	(none)	(none)
def-allocator-var	**omp_set_default_allocator()**	**omp_get_default_allocator()**

1 ### Description

2 • The value of the *nthreads-var* ICV is a list. The runtime call **omp_set_num_threads()** sets
3 the value of the first element of this list, and **omp_get_max_threads()** retrieves the value
4 of the first element of this list.

5 • The value of the *bind-var* ICV is a list. The runtime call **omp_get_proc_bind()** retrieves
6 the value of the first element of this list.

- Detailed values in the *place-partition-var* ICV are retrieved using the runtime calls **omp_get_partition_num_places()**, **omp_get_partition_place_nums()**, **omp_get_place_num_procs()**, and **omp_get_place_proc_ids()**.

Cross References

- **thread_limit** clause of the **teams** construct, see Section 2.15.9 on page 157.
- **omp_set_num_threads** routine, see Section 3.2.1 on page 300.
- **omp_get_max_threads** routine, see Section 3.2.3 on page 302.
- **omp_set_dynamic** routine, see Section 3.2.7 on page 306.
- **omp_get_dynamic** routine, see Section 3.2.8 on page 308.
- **omp_get_cancellation** routine, see Section 3.2.9 on page 308.
- **omp_set_nested** routine, see Section 3.2.10 on page 309.
- **omp_get_nested** routine, see Section 3.2.11 on page 311.
- **omp_set_schedule** routine, see Section 3.2.12 on page 311.
- **omp_get_schedule** routine, see Section 3.2.13 on page 313.
- **omp_get_thread_limit** routine, see Section 3.2.14 on page 314.
- **omp_set_max_active_levels** routine, see Section 3.2.15 on page 315.
- **omp_get_max_active_levels** routine, see Section 3.2.16 on page 317.
- **omp_get_level** routine, see Section 3.2.17 on page 318.
- **omp_get_active_level** routine, see Section 3.2.20 on page 321.
- **omp_get_proc_bind** routine, see Section 3.2.22 on page 323.
- **omp_get_place_num_procs()** routine, see Section 3.2.24 on page 326.
- **omp_get_place_proc_ids()** routine, see Section 3.2.25 on page 327.
- **omp_get_partition_num_places()** routine, see Section 3.2.27 on page 329.
- **omp_get_partition_place_nums()** routine, see Section 3.2.28 on page 330.
- **omp_set_affinity_format** routine, see Section 3.2.29 on page 331.
- **omp_get_affinity_format** routine, see Section 3.2.30 on page 332.
- **omp_set_default_device** routine, see Section 3.2.33 on page 336.
- **omp_get_default_device** routine, see Section 3.2.34 on page 337.
- **omp_get_max_task_priority** routine, see Section 3.2.41 on page 343.

1 • **`omp_set_default_allocator`** routine, see Section 3.6.2 on page 369.

2 • **`omp_get_default_allocator`** routine, see Section 3.6.3 on page 370.

3 2.4.4 How ICVs are Scoped

4 Table 2.3 shows the ICVs and their scope.

TABLE 2.3: Scopes of ICVs

ICV	Scope
dyn-var	data environment
nest-var	data environment
nthreads-var	data environment
run-sched-var	data environment
def-sched-var	device
bind-var	data environment
stacksize-var	device
wait-policy-var	device
thread-limit-var	data environment
max-active-levels-var	device
active-levels-var	data environment
levels-var	data environment
place-partition-var	implicit task
cancel-var	global
display-affinity-var	global
affinity-format-var	device

table continued on next page

table continued from previous page

ICV	Scope
default-device-var	data environment
target-offload-var	global
max-task-priority-var	global
tool-var	global
tool-libraries-var	global
ompd-tool-var	global
third-party-tool-var	global
def-allocator-var	implicit task

1 **Description**

2 • There is one copy per device of each ICV with device scope

3 • Each data environment has its own copies of ICVs with data environment scope

4 • Each implicit task has its own copy of ICVs with implicit task scope

5 Calls to OpenMP API routines retrieve or modify data environment scoped ICVs in the data
6 environment of their binding tasks.

7 2.4.4.1 How the Per-Data Environment ICVs Work

8 When a **task** construct or **parallel** construct is encountered, the generated task(s) inherit the
9 values of the data environment scoped ICVs from the generating task's ICV values.

10 When a **parallel** construct is encountered, the value of each ICV witch implicit task scope is
11 inherited, unless otherwise specified, from the implicit binding task of the generating task unless
12 otherwise specified.

13 When a **task** construct is encountered, the generated task inherits the value of *nthreads-var* from
14 the generating task's *nthreads-var* value. When a **parallel** construct is encountered, and the
15 generating task's *nthreads-var* list contains a single element, the generated task(s) inherit that list as
16 the value of *nthreads-var*. When a **parallel** construct is encountered, and the generating task's
17 *nthreads-var* list contains multiple elements, the generated task(s) inherit the value of *nthreads-var*
18 as the list obtained by deletion of the first element from the generating task's *nthreads-var* value.
19 The *bind-var* ICV is handled in the same way as the *nthreads-var* ICV.

When a *target task* executes a **target** region, the generated initial task uses the values of the data environment scoped ICVs from the device data environment ICV values of the device that will execute the region.

If a **teams** construct with a **thread_limit** clause is encountered, the *thread-limit-var* ICV of the construct's data environment is instead set to a value that is less than or equal to the value specified in the clause.

When encountering a loop worksharing region with **schedule(runtime)**, all implicit task regions that constitute the binding parallel region must have the same value for *run-sched-var* in their data environments. Otherwise, the behavior is unspecified.

2.4.5 ICV Override Relationships

Table 2.4 shows the override relationships among construct clauses and ICVs.

TABLE 2.4: ICV Override Relationships

ICV	construct clause, if used
dyn-var	(none)
nest-var	(none)
nthreads-var	**num_threads**
run-sched-var	**schedule**
def-sched-var	**schedule**
bind-var	**proc_bind**
stacksize-var	(none)
wait-policy-var	(none)
thread-limit-var	(none)
max-active-levels-var	(none)
active-levels-var	(none)
levels-var	(none)

table continued on next page

table continued from previous page

ICV	construct clause, if used
place-partition-var	(none)
cancel-var	(none)
display-affinity-var	(none)
affinity-format-var	(none)
default-device-var	(none)
target-offload-var	(none)
max-task-priority-var	(none)
tool-var	(none)
tool-libraries-var	(none)
ompd-tool-var	(none)
def-allocator-var	**allocator**

1 **Description**

2 • The **num_threads** clause overrides the value of the first element of the *nthreads-var* ICV.

3 • If *bind-var* is not set to *false* then the **proc_bind** clause overrides the value of the first element
4 of the *bind-var* ICV; otherwise, the **proc_bind** clause has no effect.

5 **Cross References**

6 • **parallel** construct, see Section 2.8 on page 66.

7 • **proc_bind** clause, Section 2.8 on page 66.

8 • **num_threads** clause, see Section 2.8.1 on page 71.

9 • Loop construct, see Section 2.10.1 on page 78.

10 • **schedule** clause, see Section 2.10.1.1 on page 86.

2.5 Array Sections

An array section designates a subset of the elements in an array. An array section can appear only in clauses where it is explicitly allowed.

C / C++

To specify an array section in an OpenMP construct, array subscript expressions are extended with the following syntax:

[*lower-bound* **:** *length* **]** or

[*lower-bound* **:** **]** or

[**:** *length* **]** or

[:]

The array section must be a subset of the original array.

Array sections are allowed on multidimensional arrays. Base language array subscript expressions can be used to specify length-one dimensions of multidimensional array sections.

The *lower-bound* and *length* are integral type expressions. When evaluated they represent a set of integer values as follows:

{ *lower-bound*, *lower-bound* + 1, *lower-bound* + 2,... , *lower-bound* + *length* - 1 }

The *length* must evaluate to a non-negative integer.

When the size of the array dimension is not known, the *length* must be specified explicitly.

When the *length* is absent, it defaults to the size of the array dimension minus the *lower-bound*.

When the *lower-bound* is absent it defaults to 0.

Note – The following are examples of array sections:

```
a[0:6]
a[:6]
a[1:10]
a[1:]
b[10][:][:0]
c[1:10][42][0:6]
S.c[:100]
p->y[:10]
```

```
this->a[:N]
```

The first two examples are equivalent. If **a** is declared to be an eleven element array, the third and fourth examples are equivalent. The fifth example is a zero-length array section. The sixth example is not contiguous. The remaining examples show array sections that are formed from more general base expressions.

C / C++

Fortran has built-in support for array sections although some restrictions apply to their use, as enumerated in the following section.

Restrictions

Restrictions to array sections are as follows:

- An array section can appear only in clauses where it is explicitly allowed.

C / C++

- The type of the base expression appearing in an array section must be an array or pointer type.
- If the type of the base expression of the array section is a pointer type, the base expression must be an lvalue expression.

C / C++

C++

- If the type of the base expression of an array section is a reference to a type *T* then the type will be considered to be *T* for all purposes of the array section.
- An array section cannot be used in a C++ user-defined `[]`-operator.

C++

Fortran

- A stride expression may not be specified.
- The upper bound for the last dimension of an assumed-size dummy array must be specified.
- If a list item is an array section with vector subscripts, the first array element must be the lowest in the array element order of the array section.

Fortran

2.6 Iterators

Iterators are identifiers that expand to multiple values in the clause on which they appear.

The syntax to define an iterator is the following:

```
iterators-definition
```

where *iterators-definition* is either an *iterator-specifier* or an *iterator-specifier-list*.

The syntax of an *iterator-specifier* is one of the following:

[*iterator-type*] *identifier* = *range-specification*

where:

- *identifier* is a base language identifier.

C / C++

- *iterator-type* is a type name.

C / C++

Fortran

- *iterator-type* is a type specifier.

Fortran

- *range-specification* is of the form *begin, end[, step]* where *begin*, *end* and *step* are expressions for which their types can be converted to the *iterator-type* type.

The syntax of an *iterator-specifier-list* is as follows:

(*iterator-specifier*) [, *iterator-specifier-list*]

C / C++

- The *iterator-type* must be an integral or pointer type.
- In an *iterator-specifier*, if the *iterator-type* is not specified then the type of that iterator is of **int** type.

C / C++

Fortran

- The *iterator-type* must be an integer type.
- In an *iterator-specifier*, if the *iterator-type* is not specified then the type of that iterator is default integer.

Fortran

In a *range-specification*, if the *step* is not specified its value is implicitly defined to be 1.

An iterator only exists in the context of the clause on which it appears. An iterator also hides all accessible symbols with the same name in the context of the clause.

The use of a variable in an expression that appears in the *range-specification* causes an implicit reference to the variable in all enclosing constructs.

C / C++

The values of the iterator are the set of values $i_0 ... i_{N-1}$ where $i_0 = begin$, $i_j = i_{j-1} + step$ and

- $i_{N-1} < end$ and $i_{N-1} + step >= end$ if $step > 0$.
- $i_{N-1} > end$ and $i_{N-1} + step <= end$ if $step < 0$.

C / C++

Fortran

The values of the iterator are the set of values $i_1 ... i_N$ where $i_1 = begin$, $i_j = i_{j-1} + step$ and

- $i_N <= end$ and $i_N + step > end$ if $step > 0$.
- $i_N >= end$ and $i_N + step < end$ if $step < 0$.

Fortran

An expression containing an iterator identifier can only appear in clauses that explicitly allow expressions containing iterators. For those clauses, the effect is as if the list item is instantiated within the clause for each value of the iterator in the set defined above, substituting each occurrence of the iterator identifier in the expression with the iterator value.

Restrictions

- If the *step* expression of a *range-specification* equals zero the behavior is unspecified.
- Each iterator identifier can only be defined once in an *iterators-definition*.
- Iterators cannot appear in the *range-specification*.

1
2.7 Memory Allocators

2 OpenMP memory allocators can be used by a program to make allocation requests. When a
3 memory allocator receives a request to allocate storage of a certain size, it will try to return an
4 allocation of logically consecutive memory in its associated storage resources of at least the size
5 being requested. This allocation will not alias with any other existing allocation from an OpenMP
6 memory allocator. Table 2.5 shows the list of predefined memory allocators. The use of a given
7 memory allocator expresses an intent to use storage with certain traits for the allocations. The
8 actual storage resources that each memory allocator will use are implementation defined.

TABLE 2.5: Predefined Allocators

Allocator name	Storage selection intent
omp_default_mem_alloc	Use default storage for allocations.
omp_large_cap_mem_alloc	Use storage with large capacity.
omp_const_mem_alloc	Use storage optimized for read-only variables. The result of writing to memory returned by this memory allocator is unspecified.
omp_high_bw_mem_alloc	Use storage with high bandwidth.
omp_low_lat_mem_alloc	Use storage with low latency.
omp_cgroup_mem_alloc	Use storage that is close to all threads in the same contention group of the thread requesting the allocation. Attempts to access the memory returned by this allocator from a thread that is not part of the same contention group as the thread that allocated the memory result in unspecified behavior.
omp_pteam_mem_alloc	Use storage that is close to all threads that bind to the same **parallel** region of the thread requesting the allocation. Attempts to access the memory returned by this allocator from a thread that does not bind to the same **parallel** region as the thread that allocated the memory results in unspecified behavior.

table continued on next page

table continued from previous page

Allocator name	Storage selection intent
omp_thread_mem_alloc	Use storage that is close to the *thread* requesting the allocation. Attempts to access the memory returned by this allocator from a thread other than the one that allocated the memory results in unspecified behavior.

For memory allocators other than **omp_cgroup_mem_alloc**, **omp_pteam_mem_alloc** and **omp_thread_mem_alloc** the returned memory must be accessible by all threads in the device where the allocation was requested.

Fortran

If any operation of the base language causes a reallocation of an array that is allocated with a memory allocator then that memory allocator will be used to release the current memory and to allocate the new memory.

Fortran

Cross References

- **omp_set_default_allocator** routine, see Section 3.6.2 on page 369.
- **omp_get_default_allocator** routine, see Section 3.6.3 on page 370.
- **OMP_ALLOCATOR** environment variable, see Section 5.21 on page 580.

2.8 parallel Construct

Summary

This fundamental construct starts parallel execution. See Section 1.3 on page 18 for a general description of the OpenMP execution model.

Syntax

C / C++

The syntax of the **parallel** construct is as follows:

```
#pragma omp parallel [clause[ [,] clause] ... ] new-line
    structured-block
```

where *clause* is one of the following:

if([**parallel :**] *scalar-expression***)**

num_threads(*integer-expression***)**

default(shared | none)

private(*list***)**

firstprivate(*list***)**

shared(*list***)**

copyin(*list***)**

reduction(*reduction-identifier* **:** *list***)**

proc_bind(master | close | spread)

allocate([*allocator:*]*list***)**

C / C++

Fortran

The syntax of the **parallel** construct is as follows:

```
!$omp parallel [clause[ [,] clause] ... ]
    structured-block
!$omp end parallel
```

where *clause* is one of the following:

if([**parallel :**] *scalar-logical-expression***)**

num_threads(*scalar-integer-expression***)**

default(private | firstprivate | shared | none)

private(*list***)**

firstprivate(*list***)**

shared(*list***)**

copyin(*list***)**

reduction(*reduction-identifier* **:** *list***)**

proc_bind(master | close | spread)

allocate([*allocator:*]*list***)**

The **end parallel** directive denotes the end of the **parallel** construct.

Fortran

Binding

The binding thread set for a **parallel** region is the encountering thread. The encountering thread becomes the master thread of the new team.

Description

When a thread encounters a **parallel** construct, a team of threads is created to execute the **parallel** region (see Section 2.8.1 on page 71 for more information about how the number of threads in the team is determined, including the evaluation of the **if** and **num_threads** clauses). The thread that encountered the **parallel** construct becomes the master thread of the new team, with a thread number of zero for the duration of the new **parallel** region. All threads in the new team, including the master thread, execute the region. Once the team is created, the number of threads in the team remains constant for the duration of that **parallel** region.

The optional **proc_bind** clause, described in Section 2.8.2 on page 73, specifies the mapping of OpenMP threads to places within the current place partition, that is, within the places listed in the *place-partition-var* ICV for the implicit task of the encountering thread.

Within a **parallel** region, thread numbers uniquely identify each thread. Thread numbers are consecutive whole numbers ranging from zero for the master thread up to one less than the number of threads in the team. A thread may obtain its own thread number by a call to the **omp_get_thread_num** library routine.

A set of implicit tasks, equal in number to the number of threads in the team, is generated by the encountering thread. The structured block of the **parallel** construct determines the code that will be executed in each implicit task. Each task is assigned to a different thread in the team and becomes tied. The task region of the task being executed by the encountering thread is suspended and each thread in the team executes its implicit task. Each thread can execute a path of statements that is different from that of the other threads

The implementation may cause any thread to suspend execution of its implicit task at a task scheduling point, and switch to execute any explicit task generated by any of the threads in the team, before eventually resuming execution of the implicit task (for more details see Section 2.13 on page 110).

There is an implied barrier at the end of a **parallel** region. After the end of a **parallel** region, only the master thread of the team resumes execution of the enclosing task region.

If a thread in a team executing a **parallel** region encounters another **parallel** directive, it creates a new team, according to the rules in Section 2.8.1 on page 71, and it becomes the master of that new team.

If execution of a thread terminates while inside a **parallel** region, execution of all threads in all teams terminates. The order of termination of threads is unspecified. All work done by a team prior to any barrier that the team has passed in the program is guaranteed to be complete. The amount of work done by each thread after the last barrier that it passed and before it terminates is unspecified.

Events

The *parallel-begin* event occurs in a thread encountering a **parallel** construct before any implicit task is created for the associated parallel region.

Upon creation of each implicit task, an *implicit-task-begin* event occurs in the thread executing the implicit task after the implicit task is fully initialized but before the thread begins to execute the structured block of the **parallel** construct.

If the **parallel** region creates a thread, a *thread-begin* event occurs as the first event in the context of the new thread prior to the *implicit-task-begin*.

If the **parallel** region activates an idle thread to create the implicit task, an *idle-end* event occurs in the newly activated thread prior to the *implicit-task-begin*.

Events associated with implicit barriers occur at the end of a **parallel** region. Section 2.18.4 describes events associated with implicit barriers.

When a thread finishes an implicit task, an *implicit-task-end* event occurs in the thread after events associated with implicit barrier synchronization in the implicit task.

The *parallel-end* event occurs in the thread encountering the **parallel** construct after the thread executes its *implicit-task-end* event but before resuming execution of the encountering task.

If a thread is destroyed at the end of a **parallel** region, a *thread-end* event occurs in the thread as the last event prior to the thread's destruction.

If a non-master thread is not destroyed at the end of a **parallel** region, an *idle-begin* event occurs after the thread's *implicit-task-end* event for the **parallel** region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_parallel_begin** callback for each occurrence of a *parallel-begin* event in that thread. The callback occurs in the task encountering the **parallel** construct. This callback has the type signature **ompt_callback_parallel_begin_t**.

A thread dispatches a registered **ompt_callback_implicit_task** callback for each occurrence of a *implicit-task-begin* and *implicit-task-end* event in that thread. The callback occurs in the context of the implicit task. The callback has type signature **ompt_callback_implicit_task_t**. The callback receives **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate.

A thread dispatches a registered **ompt_callback_parallel_end** callback for each occurrence of a *parallel-end* event in that thread. The callback occurs in the task encountering the **parallel** construct. This callback has the type signature **ompt_callback_parallel_end_t**.

A thread dispatches a registered **ompt_callback_idle** callback for each occurrence of a *idle-begin* and *idle-end* event in that thread. The callback occurs in the context of the idling thread. The callback has type signature **ompt_callback_idle_t**. The callback receives **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate.

1 A thread dispatches a registered **ompt_callback_thread_begin** callback for the
2 *thread-begin* event in that thread. The callback occurs in the context of the thread. The callback has
3 type signature **ompt_callback_thread_begin_t**.

4 A thread dispatches a registered **ompt_callback_thread_end** callback for the *thread-end*
5 event in that thread. The callback occurs in the context of the thread. The callback has type
6 signature **ompt_callback_thread_end_t**.

7 ## Restrictions

8 Restrictions to the **parallel** construct are as follows:

9 • A program that branches into or out of a **parallel** region is non-conforming.

10 • A program must not depend on any ordering of the evaluations of the clauses of the **parallel**
11 directive, or on any side effects of the evaluations of the clauses.

12 • At most one **if** clause can appear on the directive.

13 • At most one **proc_bind** clause can appear on the directive.

14 • At most one **num_threads** clause can appear on the directive. The **num_threads**
15 expression must evaluate to a positive integer value.

C / C++

16 A **throw** executed inside a **parallel** region must cause execution to resume within the same
17 **parallel** region, and the same thread that threw the exception must catch it.

C / C++

Fortran

18 Unsynchronized use of Fortran I/O statements by multiple threads on the same unit has unspecified
19 behavior.

Fortran

Cross References

- **if** clause, see Section 2.17 on page 192.
- **default**, **shared**, **private**, **firstprivate**, and **reduction** clauses, see Section 2.20.3 on page 249.
- **copyin** clause, see Section 2.20.5 on page 275.
- **omp_get_thread_num** routine, see Section 3.2.4 on page 304.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.1.3.4.10 on page 397.
- **ompt_callback_thread_begin_t**, see Section 4.1.4.2.1 on page 406.
- **ompt_callback_thread_end_t**, see Section 4.1.4.2.2 on page 407.
- **ompt_callback_idle_t**, see Section 4.1.4.2.3 on page 407.
- **ompt_callback_parallel_begin_t**, see Section 4.1.4.2.4 on page 408.
- **ompt_callback_parallel_end_t**, see Section 4.1.4.2.5 on page 410.
- **ompt_callback_implicit_task_t**, see Section 4.1.4.2.11 on page 416.

2.8.1 Determining the Number of Threads for a parallel Region

When execution encounters a **parallel** directive, the value of the **if** clause or **num_threads** clause (if any) on the directive, the current parallel context, and the values of the *nthreads-var*, *dyn-var*, *thread-limit-var*, *max-active-levels-var*, and *nest-var* ICVs are used to determine the number of threads to use in the region.

Using a variable in an **if** or **num_threads** clause expression of a **parallel** construct causes an implicit reference to the variable in all enclosing constructs. The **if** clause expression and the **num_threads** clause expression are evaluated in the context outside of the **parallel** construct, and no ordering of those evaluations is specified. It is also unspecified whether, in what order, or how many times any side effects of the evaluation of the **num_threads** or **if** clause expressions occur.

When a thread encounters a **parallel** construct, the number of threads is determined according to Algorithm 2.1.

Algorithm 2.1

let *ThreadsBusy* be the number of OpenMP threads currently executing in this contention group;

let *ActiveParRegions* be the number of enclosing active parallel regions;

if an `if` clause exists

then let *IfClauseValue* be the value of the `if` clause expression;

else let *IfClauseValue* = *true*;

if a `num_threads` clause exists

then let *ThreadsRequested* be the value of the `num_threads` clause expression;

else let *ThreadsRequested* = value of the first element of *nthreads-var*;

if a `thread_limit` clause exists on a `teams` construct associated with an enclosing `teams` region

then let *ThreadLimit* be the value of the `thread_limit` clause expression;

else let *ThreadLimit* = *thread-limit-var*;

let *ThreadsAvailable* = (*ThreadLimit* - *ThreadsBusy* + 1);

if (*IfClauseValue* = *false*)

then number of threads = 1;

else if (*ActiveParRegions* >= 1) **and** (*nest-var* = *false*)

then number of threads = 1;

else if (*ActiveParRegions* = *max-active-levels-var*)

then number of threads = 1;

else if (*dyn-var* = *true*) **and** (*ThreadsRequested* <= *ThreadsAvailable*)

then number of threads = [1 : *ThreadsRequested*];

else if (*dyn-var* = *true*) **and** (*ThreadsRequested* > *ThreadsAvailable*)

then number of threads = [1 : *ThreadsAvailable*];

else if (*dyn-var* = *false*) **and** (*ThreadsRequested* <= *ThreadsAvailable*)

then number of threads = *ThreadsRequested*;

else if (*dyn-var* = *false*) **and** (*ThreadsRequested* > *ThreadsAvailable*)

then behavior is implementation defined;

Note – Since the initial value of the *dyn-var* ICV is implementation defined, programs that depend on a specific number of threads for correct execution should explicitly disable dynamic adjustment of the number of threads.

Cross References

- *nthreads-var*, *dyn-var*, *thread-limit-var*, *max-active-levels-var*, and *nest-var* ICVs, see Section 2.4 on page 49.

2.8.2 Controlling OpenMP Thread Affinity

When a thread encounters a **parallel** directive without a **proc_bind** clause, the *bind-var* ICV is used to determine the policy for assigning OpenMP threads to places within the current place partition, that is, the places listed in the *place-partition-var* ICV for the implicit task of the encountering thread. If the **parallel** directive has a **proc_bind** clause then the binding policy specified by the **proc_bind** clause overrides the policy specified by the first element of the *bind-var* ICV. Once a thread in the team is assigned to a place, the OpenMP implementation should not move it to another place.

The **master** thread affinity policy instructs the execution environment to assign every thread in the team to the same place as the master thread. The place partition is not changed by this policy, and each implicit task inherits the *place-partition-var* ICV of the parent implicit task.

The **close** thread affinity policy instructs the execution environment to assign the threads in the team to places close to the place of the parent thread. The place partition is not changed by this policy, and each implicit task inherits the *place-partition-var* ICV of the parent implicit task. If T is the number of threads in the team, and P is the number of places in the parent's place partition, then the assignment of threads in the team to places is as follows:

- $T \leq P$. The master thread executes on the place of the parent thread. The thread with the next smallest thread number executes on the next place in the place partition, and so on, with wrap around with respect to the place partition of the master thread.

- $T > P$. Each place P will contain S_p threads with consecutive thread numbers, where $\lfloor T/P \rfloor \leq Sp \leq \lceil T/P \rceil$. The first S_0 threads (including the master thread) are assigned to the place of the parent thread. The next S_1 threads are assigned to the next place in the place partition, and so on, with wrap around with respect to the place partition of the master thread. When P does not divide T evenly, the exact number of threads in a particular place is implementation defined.

The purpose of the **spread** thread affinity policy is to create a sparse distribution for a team of T threads among the P places of the parent's place partition. A sparse distribution is achieved by first subdividing the parent partition into T subpartitions if $T \leq P$, or P subpartitions if $T > P$. Then one thread ($T \leq P$) or a set of threads ($T > P$) is assigned to each subpartition. The *place-partition-var* ICV of each implicit task is set to its subpartition. The subpartitioning is not only a mechanism for achieving a sparse distribution, it also defines a subset of places for a thread to use when creating a nested **parallel** region. The assignment of threads to places is as follows:

- $T \leq P$. The parent thread's place partition is split into T subpartitions, where each subpartition contains $\lfloor P/T \rfloor$ or $\lceil P/T \rceil$ consecutive places. A single thread is assigned to each subpartition. The master thread executes on the place of the parent thread and is assigned to the subpartition that includes that place. The thread with the next smallest thread number is assigned to the first place in the next subpartition, and so on, with wrap around with respect to the original place partition of the master thread.

- $T > P$. The parent thread's place partition is split into P subpartitions, each consisting of a single place. Each subpartition is assigned S_p threads with consecutive thread numbers, where $\lfloor T/P \rfloor \leq S_p \leq \lceil T/P \rceil$. The first S_0 threads (including the master thread) are assigned to the subpartition containing the place of the parent thread. The next S_1 threads are assigned to the next subpartition, and so on, with wrap around with respect to the original place partition of the master thread. When P does not divide T evenly, the exact number of threads in a particular subpartition is implementation defined.

The determination of whether the affinity request can be fulfilled is implementation defined. If the affinity request cannot be fulfilled, then the affinity of threads in the team is implementation defined.

Note – Wrap around is needed if the end of a place partition is reached before all thread assignments are done. For example, wrap around may be needed in the case of **close** and $T \leq P$, if the master thread is assigned to a place other than the first place in the place partition. In this case, thread 1 is assigned to the place after the place of the master place, thread 2 is assigned to the place after that, and so on. The end of the place partition may be reached before all threads are assigned. In this case, assignment of threads is resumed with the first place in the place partition.

1

2.9 Canonical Loop Form

C / C++

2 A loop has *canonical loop form* if it conforms to the following:

for (*init-expr*; *test-expr*; *incr-expr*) *structured-block*	
init-expr	One of the following: *var* = *lb* *integer-type var* = *lb* *random-access-iterator-type var* = *lb* *pointer-type var* = *lb*
test-expr	One of the following: *var relational-op b* *b relational-op var*
incr-expr	One of the following: ++*var* *var*++ - - *var* *var* - - *var* += *incr* *var* - = *incr* *var* = *var* + *incr* *var* = *incr* + *var* *var* = *var* - *incr*
var	One of the following: A variable of a signed or unsigned integer type. For C++, a variable of a random access iterator type. For C, a variable of a pointer type. If this variable would otherwise be shared, it is implicitly made private in the loop construct. This variable must not be modified during the execution of the *for-loop* other than in *incr-expr*. Unless the variable is specified **lastprivate** or **linear** on the loop construct, its value after the loop is unspecified.

continued on next page

continued from previous page

relational-op	One of the following: **`<`** **`<=`** **`>`** **`>=`** **`!=`**
lb and *b*	Loop invariant expressions of a type compatible with the type of *var*.
incr	A loop invariant integer expression.

The canonical form allows the iteration count of all associated loops to be computed before executing the outermost loop. The computation is performed for each loop in an integer type. This type is derived from the type of *var* as follows:

- If *var* is of an integer type, then the type is the type of *var*.
- For C++, if *var* is of a random access iterator type, then the type is the type that would be used by *std::distance* applied to variables of the type of *var*.
- For C, if *var* is of a pointer type, then the type is **`ptrdiff_t`**.

The behavior is unspecified if any intermediate result required to compute the iteration count cannot be represented in the type determined above.

There is no implied synchronization during the evaluation of the *lb*, *b*, or *incr* expressions. It is unspecified whether, in what order, or how many times any side effects within the *lb*, *b*, or *incr* expressions occur.

Note – Random access iterators are required to support random access to elements in constant time. Other iterators are precluded by the restrictions since they can take linear time or offer limited functionality. It is therefore advisable to use tasks to parallelize those cases.

Restrictions

The following restrictions also apply:

- If *test-expr* is of the form *var relational-op b* and *relational-op* is < or <= then *incr-expr* must cause *var* to increase on each iteration of the loop. If *test-expr* is of the form *var relational-op b* and *relational-op* is > or >= then *incr-expr* must cause *var* to decrease on each iteration of the loop.
- If *test-expr* is of the form *b relational-op var* and *relational-op* is < or <= then *incr-expr* must cause *var* to decrease on each iteration of the loop. If *test-expr* is of the form *b relational-op var* and *relational-op* is > or >= then *incr-expr* must cause *var* to increase on each iteration of the loop.
- If *test-expr* is of the form *b* != *var* or *var* != *b* then *incr-expr* must cause *var* either to increase on each iteration of the loop or to decrease on each iteration of the loop.
- For C++, in the **`simd`** construct the only random access iterator types that are allowed for *var* are pointer types.
- The *b*, *lb* and *incr* expressions may not reference *var* of any of the associated loops.
- If *relational-op* is != and *incr-expr* is of the form that has *incr* then *incr* must be a constant expression and evaluate to -1 or 1.

C / C++

2.10 Worksharing Constructs

A worksharing construct distributes the execution of the associated region among the members of the team that encounters it. Threads execute portions of the region in the context of the implicit tasks each one is executing. If the team consists of only one thread then the worksharing region is not executed in parallel.

A worksharing region has no barrier on entry; however, an implied barrier exists at the end of the worksharing region, unless a **`nowait`** clause is specified. If a **`nowait`** clause is present, an implementation may omit the barrier at the end of the worksharing region. In this case, threads that finish early may proceed straight to the instructions following the worksharing region without waiting for the other members of the team to finish the worksharing region, and without performing a flush operation.

The OpenMP API defines the following worksharing constructs, and these are described in the sections that follow:

- loop construct

- **`sections`** construct
- **`single`** construct
- **`workshare`** construct

Restrictions

The following restrictions apply to worksharing constructs:

- Each worksharing region must be encountered by all threads in a team or by none at all, unless cancellation has been requested for the innermost enclosing parallel region.
- The sequence of worksharing regions and **`barrier`** regions encountered must be the same for every thread in a team

2.10.1 Loop Construct

Summary

The loop construct specifies that the iterations of one or more associated loops will be executed in parallel by threads in the team in the context of their implicit tasks. The iterations are distributed across threads that already exist in the team executing the **`parallel`** region to which the loop region binds.

Syntax

C / C++

The syntax of the loop construct is as follows:

```
#pragma omp for [clause[ [,] clause] ... ] new-line
    for-loops
```

where clause is one of the following:

1 **private(***list***)**

2 **firstprivate(***list***)**

3 **lastprivate(***[lastprivate-modifier* **:** *] list***)**

4 **linear(***list[* **:** *linear-step]***)**

5 **reduction(***reduction-identifier* **:** *list***)**

6 **schedule(***[modifier [***,** *modifier]***:** *]kind[***,** *chunk_size]***)**

7 **collapse(***n***)**

8 **ordered***[***(***n***)** *]*

9 **nowait**

10 **allocate(***[allocator:]list***)**

11 The **for** directive places restrictions on the structure of all associated *for-loops*. Specifically, all
12 associated *for-loops* must have *canonical loop form* (see Section 2.9 on page 75).

C / C++

Fortran

13 The syntax of the loop construct is as follows:

```
!$omp do [clause[ [,] clause] ... ]
    do-loops
[!$omp end do [nowait]]
```

14 where *clause* is one of the following:

15 **private(***list***)**

16 **firstprivate(***list***)**

17 **lastprivate(***[lastprivate-modifier* **:** *] list***)**

18 **linear(***list[* **:** *linear-step]***)**

19 **reduction(***reduction-identifier* **:** *list***)**

20 **schedule(***[modifier [***,** *modifier]***:** *]kind[***,** *chunk_size]***)**

21 **collapse(***n***)**

22 **ordered***[***(***n***)** *]*

23 **allocate(***[allocator:]list***)**

1 If an **end do** directive is not specified, an **end do** directive is assumed at the end of the *do-loops*.

2 Any associated *do-loop* must be a *do-construct* or an *inner-shared-do-construct* as defined by the
3 Fortran standard. If an **end do** directive follows a *do-construct* in which several loop statements
4 share a **DO** termination statement, then the directive can only be specified for the outermost of these
5 **DO** statements.

6 If any of the loop iteration variables would otherwise be shared, they are implicitly made private on
7 the loop construct.

Fortran

8 ### Binding

9 The binding thread set for a loop region is the current team. A loop region binds to the innermost
10 enclosing **parallel** region. Only the threads of the team executing the binding **parallel**
11 region participate in the execution of the loop iterations and the implied barrier of the loop region if
12 the barrier is not eliminated by a **nowait** clause.

13 ### Description

14 The loop construct is associated with a loop nest consisting of one or more loops that follow the
15 directive.

16 There is an implicit barrier at the end of a loop construct unless a **nowait** clause is specified.

17 The **collapse** clause may be used to specify how many loops are associated with the loop
18 construct. The parameter of the **collapse** clause must be a constant positive integer expression.
19 If a **collapse** clause is specified with a parameter value greater than 1, then the iterations of the
20 associated loops to which the clause applies are collapsed into one larger iteration space that is then
21 divided according to the **schedule** clause. The sequential execution of the iterations in these
22 associated loops determines the order of the iterations in the collapsed iteration space. If no
23 **collapse** clause is present or its parameter is 1, the only loop that is associated with the loop
24 construct for the purposes of determining how the iteration space is divided according to the
25 **schedule** clause is the one that immediately follows the loop directive.

26 If more than one loop is associated with the loop construct then the number of times that any
27 intervening code between any two associated loops will be executed is unspecified but will be at
28 least once per iteration of the loop enclosing the intervening code and at most once per iteration of
29 the innermost loop associated with the construct.

30 The iteration count for each associated loop is computed before entry to the outermost loop. If
31 execution of any associated loop changes any of the values used to compute any of the iteration
32 counts, then the behavior is unspecified.

33 The integer type (or kind, for Fortran) used to compute the iteration count for the collapsed loop is
34 implementation defined.

1 A worksharing loop has logical iterations numbered 0,1,...,N-1 where N is the number of loop
2 iterations, and the logical numbering denotes the sequence in which the iterations would be
3 executed if a set of associated loop(s) were executed sequentially. At the beginning of each logical
4 iteration, the loop iteration variable of each associated loop has the value that it would have if the
5 set of the associated loop(s) were executed sequentially. The **schedule** clause specifies how
6 iterations of these associated loops are divided into contiguous non-empty subsets, called chunks,
7 and how these chunks are distributed among threads of the team. Each thread executes its assigned
8 chunk(s) in the context of its implicit task. The iterations of a given chunk are executed in
9 sequential order by the assigned thread. The *chunk_size* expression is evaluated using the original
10 list items of any variables that are made private in the loop construct. It is unspecified whether, in
11 what order, or how many times, any side effects of the evaluation of this expression occur. The use
12 of a variable in a **schedule** clause expression of a loop construct causes an implicit reference to
13 the variable in all enclosing constructs.

14 Different loop regions with the same schedule and iteration count, even if they occur in the same
15 parallel region, can distribute iterations among threads differently. The only exception is for the
16 **static** schedule as specified in Table 2.6. Programs that depend on which thread executes a
17 particular iteration under any other circumstances are non-conforming.

18 See Section 2.10.1.1 on page 86 for details of how the schedule for a worksharing loop is
19 determined.

20 The schedule *kind* can be one of those specified in Table 2.6.

21 The schedule *modifier* can be one of those specified in Table 2.7. If the **static** schedule kind is
22 specified or if the **ordered** clause is specified, and if the **nonmonotonic** modifier is not
23 specified, the effect is as if the **monotonic** modifier is specified. Otherwise, unless the
24 **monotonic** modifier is specified, the effect is as if the **nonmonotonic** modifier is specified.

25 The **ordered** clause with the parameter may also be used to specify how many loops are
26 associated with the loop construct. The parameter of the **ordered** clause must be a constant
27 positive integer expression if specified. The parameter of the **ordered** clause does not affect how
28 the logical iteration space is then divided. If an **ordered** clause with the parameter is specified for
29 the loop construct, then those associated loops form a *doacross loop nest*.

30 If the value of the parameter in the **collapse** or **ordered** clause is larger than the number of
31 nested loops following the construct, the behavior is unspecified.

TABLE 2.6: schedule Clause *kind* Values

static	When **schedule(static,** *chunk_size***)** is specified, iterations are divided into chunks of size *chunk_size*, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.

table continued on next page

table continued from previous page

	When no *chunk_size* is specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread. The size of the chunks is unspecified in this case. A compliant implementation of the **`static`** schedule must ensure that the same assignment of logical iteration numbers to threads will be used in two loop regions if the following conditions are satisfied: 1) both loop regions have the same number of loop iterations, 2) both loop regions have the same value of *chunk_size* specified, or both loop regions have no *chunk_size* specified, 3) both loop regions bind to the same parallel region, and 4) neither loop is associated with a SIMD construct. A data dependence between the same logical iterations in two such loops is guaranteed to be satisfied allowing safe use of the **`nowait`** clause.
`dynamic`	When **`schedule(dynamic,`** *chunk_size*) is specified, the iterations are distributed to threads in the team in chunks. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed. Each chunk contains *chunk_size* iterations, except for the chunk that contains the sequentially last iteration, which may have fewer iterations. When no *chunk_size* is specified, it defaults to 1.
`guided`	When **`schedule(guided,`** *chunk_size*) is specified, the iterations are assigned to threads in the team in chunks. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned. For a *chunk_size* of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to 1. For a *chunk_size* with value k (greater than 1), the size of each chunk is determined in the same way, with the restriction that the chunks do not contain fewer than k iterations (except for the chunk that contains the sequentially last iteration, which may have fewer than k iterations).

table continued on next page

table continued from previous page

	When no *chunk_size* is specified, it defaults to 1.
auto	When **schedule(auto)** is specified, the decision regarding scheduling is delegated to the compiler and/or runtime system. The programmer gives the implementation the freedom to choose any possible mapping of iterations to threads in the team.
runtime	When **schedule(runtime)** is specified, the decision regarding scheduling is deferred until run time, and the schedule and chunk size are taken from the *run-sched-var* ICV. If the ICV is set to **auto**, the schedule is implementation defined.

1 Note – For a team of p threads and a loop of n iterations, let $\lceil n/p \rceil$ be the integer q that satisfies
2 $n = p * q - r$, with $0 <= r < p$. One compliant implementation of the **static** schedule (with no
3 specified *chunk_size*) would behave as though *chunk_size* had been specified with value q. Another
4 compliant implementation would assign q iterations to the first $p - r$ threads, and $q - 1$ iterations to
5 the remaining r threads. This illustrates why a conforming program must not rely on the details of a
6 particular implementation.

7 A compliant implementation of the **guided** schedule with a *chunk_size* value of k would assign
8 $q = \lceil n/p \rceil$ iterations to the first available thread and set n to the larger of $n - q$ and $p * k$. It would
9 then repeat this process until q is greater than or equal to the number of remaining iterations, at
10 which time the remaining iterations form the final chunk. Another compliant implementation could
11 use the same method, except with $q = \lceil n/(2p) \rceil$, and set n to the larger of $n - q$ and $2 * p * k$.

TABLE 2.7: schedule Clause *modifier* Values

monotonic	When the **monotonic** modifier is specified then each thread executes the chunks that it is assigned in increasing logical iteration order.
nonmonotonic	When the **nonmonotonic** modifier is specified then chunks are assigned to threads in any order and the behavior of an application that depends on any execution order of the chunks is unspecified.

table continued on next page

table continued from previous page

`simd`	When the **`simd`** modifier is specified and the loop is associated with a SIMD construct, the *chunk_size* for all chunks except the first and last chunks is $new_chunk_size = \lceil chunk_size / simd_width \rceil * simd_width$ where *simd_width* is an implementation-defined value. The first chunk will have at least *new_chunk_size* iterations except if it is also the last chunk. The last chunk may have fewer iterations than *new_chunk_size*. If the **`simd`** modifier is specified and the loop is not associated with a SIMD construct, the modifier is ignored.

1 **Events**

2 The *loop-begin* event occurs after an implicit task encounters a **`loop`** construct but before the task
3 starts the execution of the structured block of the **`loop`** region.

4 The *loop-end* event occurs after a **`loop`** region finishes execution but before resuming execution of
5 the encountering task.

6 **Tool Callbacks**

7 A thread dispatches a registered **`ompt_callback_work`** callback for each occurrence of a
8 *loop-begin* and *loop-end* event in that thread. The callback occurs in the context of the implicit
9 task. The callback has type signature **`ompt_callback_work_t`**. The callback receives
10 **`ompt_scope_begin`** or **`ompt_scope_end`** as its *endpoint* argument, as appropriate, and
11 **`ompt_work_loop`** as its *wstype* argument.

12 **Restrictions**

13 Restrictions to the loop construct are as follows:

14 • There must be no OpenMP directive in the region between any associated loops.

15 • If a **`collapse`** clause is specified, exactly one loop must occur in the region at each nesting
16 level up to the number of loops specified by the parameter of the **`collapse`** clause.

17 • If the **`ordered`** clause is present, all loops associated with the construct must be perfectly
18 nested; that is there must be no intervening code between any two loops.

19 • The values of the loop control expressions of the loops associated with the loop construct must
20 be the same for all threads in the team.

21 • Only one **`schedule`** clause can appear on a loop directive.

22 • Only one **`collapse`** clause can appear on a loop directive.

23 • *chunk_size* must be a loop invariant integer expression with a positive value.

- The value of the *chunk_size* expression must be the same for all threads in the team.
- The value of the *run-sched-var* ICV must be the same for all threads in the team.
- When **schedule(runtime)** or **schedule(auto)** is specified, *chunk_size* must not be specified.
- A *modifier* may not be specified on a **linear** clause.
- Only one **ordered** clause can appear on a loop directive.
- The **ordered** clause must be present on the loop construct if any **ordered** region ever binds to a loop region arising from the loop construct.
- The **nonmonotonic** modifier cannot be specified if an **ordered** clause is specified.
- Either the **monotonic** modifier or the **nonmonotonic** modifier can be specified but not both.
- The loop iteration variable may not appear in a **threadprivate** directive.
- If both the **collapse** and **ordered** clause with a parameter are specified, the parameter of the **ordered** clause must be greater than or equal to the parameter of the **collapse** clause.
- A **linear** clause or an **ordered** clause with a parameter can be specified on a loop directive but not both.

C / C++

- The associated *for-loops* must be structured blocks.
- Only an iteration of the innermost associated loop may be curtailed by a **continue** statement.
- No statement can branch to any associated **for** statement.
- Only one **nowait** clause can appear on a **for** directive.
- A throw executed inside a loop region must cause execution to resume within the same iteration of the loop region, and the same thread that threw the exception must catch it.

C / C++

Fortran

- The associated *do-loops* must be structured blocks.
- Only an iteration of the innermost associated loop may be curtailed by a **CYCLE** statement.
- No statement in the associated loops other than the **DO** statements can cause a branch out of the loops.
- The *do-loop* iteration variable must be of type integer.
- The *do-loop* cannot be a **DO WHILE** or a **DO** loop without loop control.

Fortran

1 **Cross References**

2 • **private**, **firstprivate**, **lastprivate**, **linear**, and **reduction** clauses, see
3 Section 2.20.3 on page 249.

4 • **OMP_SCHEDULE** environment variable, see Section 5.1 on page 564.

5 • **ordered** construct, see Section 2.18.9 on page 221.

6 • **depend** clause, see Section 2.18.10 on page 225.

7 • **ompt_scope_begin** and **ompt_scope_end**, see Section 4.1.3.4.10 on page 397.

8 • **ompt_work_loop**, see Section 4.1.3.4.13 on page 398.

9 • **ompt_callback_work_t**, see Section 4.1.4.2.16 on page 423.

10 #### 2.10.1.1 Determining the Schedule of a Worksharing Loop

11 When execution encounters a loop directive, the **schedule** clause (if any) on the directive, and
12 the *run-sched-var* and *def-sched-var* ICVs are used to determine how loop iterations are assigned
13 to threads. See Section 2.4 on page 49 for details of how the values of the ICVs are determined. If
14 the loop directive does not have a **schedule** clause then the current value of the *def-sched-var*
15 ICV determines the schedule. If the loop directive has a **schedule** clause that specifies the
16 **runtime** schedule kind then the current value of the *run-sched-var* ICV determines the schedule.
17 Otherwise, the value of the **schedule** clause determines the schedule. Figure 2.1 describes how
18 the schedule for a worksharing loop is determined.

19 **Cross References**

20 • ICVs, see Section 2.4 on page 49

21 ### 2.10.2 `sections` Construct

22 **Summary**

23 The **sections** construct is a non-iterative worksharing construct that contains a set of structured
24 blocks that are to be distributed among and executed by the threads in a team. Each structured
25 block is executed once by one of the threads in the team in the context of its implicit task.

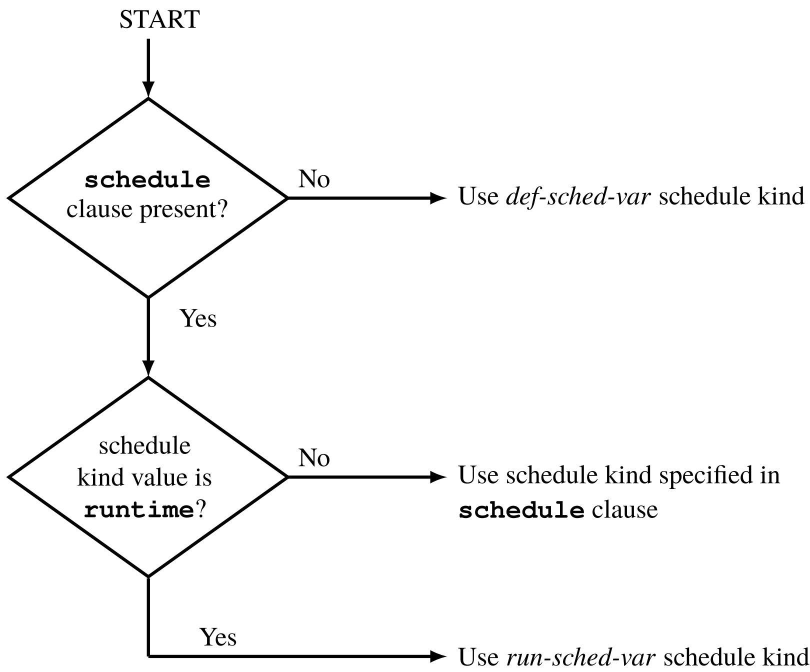


FIGURE 2.1: Determining the **schedule** for a Worksharing Loop

1 **Syntax**

C / C++

2 The syntax of the **sections** construct is as follows:

```
#pragma omp sections [clause[ [,] clause] ... ] new-line
    {
    [#pragma omp section new-line]
        structured-block
    [#pragma omp section new-line
        structured-block]
    ...
    }
```

3 where *clause* is one of the following:

1 **private(***list***)**

2 **firstprivate(***list***)**

3 **lastprivate(***[lastprivate-modifier***:** *] list***)**

4 **reduction(***reduction-identifier* **:** *list***)**

5 **nowait**

6 **allocate(***[allocator:]list***)**

C / C++

Fortran

7 The syntax of the **sections** construct is as follows:

```
!$omp sections [clause[ [,] clause] ... ]
   [!$omp section]
      structured-block
   [!$omp section
      structured-block]
   ...
!$omp end sections [nowait]
```

8 where *clause* is one of the following:

9 **private(***list***)**

10 **firstprivate(***list***)**

11 **lastprivate(***[lastprivate-modifier***:** *] list***)**

12 **reduction(***reduction-identifier* **:** *list***)**

13 **allocate(***[allocator:]list***)**

Fortran

14 Binding

15 The binding thread set for a **sections** region is the current team. A **sections** region binds to
16 the innermost enclosing **parallel** region. Only the threads of the team executing the binding
17 **parallel** region participate in the execution of the structured blocks and the implied barrier of
18 the **sections** region if the barrier is not eliminated by a **nowait** clause.

Description

Each structured block in the **sections** construct is preceded by a **section** directive except possibly the first block, for which a preceding **section** directive is optional.

The method of scheduling the structured blocks among the threads in the team is implementation defined.

There is an implicit barrier at the end of a **sections** construct unless a **nowait** clause is specified.

Events

The *sections-begin* event occurs after an implicit task encounters a **sections** construct but before the task starts the execution of the structured block of the **sections** region.

The *sections-end* event occurs after a **sections** region finishes execution but before resuming execution of the encountering task.

Tool Callbacks

A thread dispatches a registered **ompt_callback_work** callback for each occurrence of a *sections-begin* and *sections-end* event in that thread. The callback occurs in the context of the implicit task. The callback has type signature **ompt_callback_work_t**. The callback receives **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate, and **ompt_work_sections** as its *wstype* argument.

Restrictions

Restrictions to the **sections** construct are as follows:

- Orphaned **section** directives are prohibited. That is, the **section** directives must appear within the **sections** construct and must not be encountered elsewhere in the **sections** region.
- The code enclosed in a **sections** construct must be a structured block.
- Only a single **nowait** clause can appear on a **sections** directive.

C++

- A throw executed inside a **sections** region must cause execution to resume within the same section of the **sections** region, and the same thread that threw the exception must catch it.

C++

Cross References

- **private**, **firstprivate**, **lastprivate**, and **reduction** clauses, see Section 2.20.3 on page 249.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.1.3.4.10 on page 397.
- **ompt_work_sections**, see Section 4.1.3.4.13 on page 398.
- **ompt_callback_work_t**, see Section 4.1.4.2.16 on page 423.

2.10.3 single Construct

Summary

The **single** construct specifies that the associated structured block is executed by only one of the threads in the team (not necessarily the master thread), in the context of its implicit task. The other threads in the team, which do not execute the block, wait at an implicit barrier at the end of the **single** construct unless a **nowait** clause is specified.

Syntax

C / C++

The syntax of the single construct is as follows:

```
#pragma omp single [clause[ [,] clause] ... ] new-line
    structured-block
```

where *clause* is one of the following:

private(*list***)**

firstprivate(*list***)**

copyprivate(*list***)**

nowait

allocate(*[allocator:]list***)**

C / C++

Fortran

The syntax of the **single** construct is as follows:

```
!$omp single [clause[ [,] clause] ... ]
    structured-block
!$omp end single [end_clause[ [,] end_clause] ... ]
```

where *clause* is one of the following:

private(*list***)**

firstprivate(*list***)**

allocate(*[allocator:]list***)**

and *end_clause* is one of the following:

copyprivate(*list***)**

nowait

Fortran

Binding

The binding thread set for a **single** region is the current team. A **single** region binds to the innermost enclosing **parallel** region. Only the threads of the team executing the binding **parallel** region participate in the execution of the structured block and the implied barrier of the **single** region if the barrier is not eliminated by a **nowait** clause.

Description

Only one of the encountering threads will execute the structured block associated with the **single** construct. The method of choosing a thread to execute the structured block each time the team encounters the construct is implementation defined. There is an implicit barrier at the end of the **single** construct unless a **nowait** clause is specified.

Events

The *single-begin* event occurs after an **implicit task** encounters a **single** construct but before the task starts the execution of the structured block of the **single** region.

The *single-end* event occurs after a **single** region finishes execution of the structured block but before resuming execution of the encountering implicit task.

1
Tool Callbacks

2 A thread dispatches a registered **ompt_callback_work** callback for each occurrence of
3 *single-begin* and *single-end* events in that thread. The callback has type signature
4 **ompt_callback_work_t**. The callback receives **ompt_scope_begin** or
5 **ompt_scope_end** as its *endpoint* argument, as appropriate, and
6 **ompt_work_single_executor** or **ompt_work_single_other** as its *wstype* argument.

7
Restrictions

8 Restrictions to the **single** construct are as follows:

9 • The **copyprivate** clause must not be used with the **nowait** clause.

10 • At most one **nowait** clause can appear on a **single** construct.

C++

11 • A throw executed inside a **single** region must cause execution to resume within the same
12 **single** region, and the same thread that threw the exception must catch it.

C++

13
Cross References

14 • **private** and **firstprivate** clauses, see Section 2.20.3 on page 249.

15 • **copyprivate** clause, see Section 2.20.5.2 on page 277.

16 • **ompt_scope_begin** and **ompt_scope_end**, see Section 4.1.3.4.10 on page 397.

17 • **ompt_work_single_executor** and **ompt_work_single_other**, see
18 Section 4.1.3.4.13 on page 398.

19 • **ompt_callback_work_t**, Section 4.1.4.2.16 on page 423.

2.10.4 **workshare** Construct

Summary

The **workshare** construct divides the execution of the enclosed structured block into separate units of work, and causes the threads of the team to share the work such that each unit is executed only once by one thread, in the context of its implicit task.

Syntax

The syntax of the **workshare** construct is as follows:

```
!$omp workshare
    structured-block
!$omp end workshare [nowait]
```

The enclosed structured block must consist of only the following:

- array assignments
- scalar assignments
- **FORALL** statements
- **FORALL** constructs
- **WHERE** statements
- **WHERE** constructs
- **atomic** constructs
- **critical** constructs
- **parallel** constructs

Statements contained in any enclosed **critical** construct are also subject to these restrictions. Statements in any enclosed **parallel** construct are not restricted.

Binding

The binding thread set for a **workshare** region is the current team. A **workshare** region binds to the innermost enclosing **parallel** region. Only the threads of the team executing the binding **parallel** region participate in the execution of the units of work and the implied barrier of the **workshare** region if the barrier is not eliminated by a **nowait** clause.

Description

There is an implicit barrier at the end of a **workshare** construct unless a **nowait** clause is specified.

An implementation of the **workshare** construct must insert any synchronization that is required to maintain standard Fortran semantics. For example, the effects of one statement within the structured block must appear to occur before the execution of succeeding statements, and the evaluation of the right hand side of an assignment must appear to complete prior to the effects of assigning to the left hand side.

The statements in the **workshare** construct are divided into units of work as follows:

- For array expressions within each statement, including transformational array intrinsic functions that compute scalar values from arrays:
  - Evaluation of each element of the array expression, including any references to **ELEMENTAL** functions, is a unit of work.
  - Evaluation of transformational array intrinsic functions may be freely subdivided into any number of units of work.
- For an array assignment statement, the assignment of each element is a unit of work.
- For a scalar assignment statement, the assignment operation is a unit of work.
- For a **WHERE** statement or construct, the evaluation of the mask expression and the masked assignments are each a unit of work.
- For a **FORALL** statement or construct, the evaluation of the mask expression, expressions occurring in the specification of the iteration space, and the masked assignments are each a unit of work
- For an **atomic** construct, the atomic operation on the storage location designated as *x* is a unit of work.
- For a **critical** construct, the construct is a single unit of work.
- For a **parallel** construct, the construct is a unit of work with respect to the **workshare** construct. The statements contained in the **parallel** construct are executed by a new thread team.
- If none of the rules above apply to a portion of a statement in the structured block, then that portion is a unit of work.

The transformational array intrinsic functions are **MATMUL**, **DOT_PRODUCT**, **SUM**, **PRODUCT**, **MAXVAL**, **MINVAL**, **COUNT**, **ANY**, **ALL**, **SPREAD**, **PACK**, **UNPACK**, **RESHAPE**, **TRANSPOSE**, **EOSHIFT**, **CSHIFT**, **MINLOC**, and **MAXLOC**.

It is unspecified how the units of work are assigned to the threads executing a **workshare** region.

If an array expression in the block references the value, association status, or allocation status of private variables, the value of the expression is undefined, unless the same value would be computed by every thread.

If an array assignment, a scalar assignment, a masked array assignment, or a **FORALL** assignment assigns to a private variable in the block, the result is unspecified.

The **workshare** directive causes the sharing of work to occur only in the **workshare** construct, and not in the remainder of the **workshare** region.

Events

The *workshare-begin* event occurs after an implicit task encounters a **workshare** construct but before the task starts the execution of the structured block of the **workshare** region.

The *workshare-end* event occurs after a **workshare** region finishes execution but before resuming execution of the encountering task.

Tool Callbacks

A thread dispatches a registered **ompt_callback_work** callback for each occurrence of a *workshare-begin* and *workshare-end* event in that thread. The callback occurs in the context of the implicit task. The callback has type signature **ompt_callback_work_t**. The callback receives **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate, and **ompt_work_workshare** as its *wstype* argument.

Restrictions

The following restrictions apply to the **workshare** construct:

- All array assignments, scalar assignments, and masked array assignments must be intrinsic assignments.
- The construct must not contain any user defined function calls unless the function is **ELEMENTAL**.

Cross References

- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.1.3.4.10 on page 397.
- **ompt_work_workshare**, see Section 4.1.3.4.13 on page 398.
- **ompt_callback_work_t**, see Section 4.1.4.2.16 on page 423.

Fortran

2.11 SIMD Constructs

2.11.1 simd Construct

Summary

The **simd** construct can be applied to a loop to indicate that the loop can be transformed into a SIMD loop (that is, multiple iterations of the loop can be executed concurrently using SIMD instructions).

Syntax

The syntax of the **simd** construct is as follows:

C / C++

```
#pragma omp simd [clause[ [,] clause] ... ] new-line
    for-loops
```

where *clause* is one of the following:

safelen(*length***)**

simdlen(*length***)**

linear(*list*[**:** *linear-step*]**)**

aligned(*list*[**:** *alignment*]**)**

nontemporal(*list***)**

private(*list***)**

lastprivate([*lastprivate-modifier* **:**] *list***)**

reduction(*reduction-identifier* **:** *list***)**

collapse(*n***)**

The **simd** directive places restrictions on the structure of the associated *for-loops*. Specifically, all associated *for-loops* must have *canonical loop form* (Section 2.9 on page 75).

C / C++

Fortran

```
!$omp simd [clause[ [,] clause ... ]
    do-loops
[!$omp end simd]
```

1 where *clause* is one of the following:

2 **safelen(***length***)**

3 **simdlen(***length***)**

4 **linear(***list[: linear-step]***)**

5 **aligned(***list[: alignment]***)**

6 **nontemporal(***list***)**

7 **private(***list***)**

8 **lastprivate(***[lastprivate-modifier* **:** *] list***)**

9 **reduction(***reduction-identifier* **:** *list***)**

10 **collapse(***n***)**

11 If an **end simd** directive is not specified, an **end simd** directive is assumed at the end of the
12 *do-loops*.

13 Any associated *do-loop* must be a *do-construct* or an *inner-shared-do-construct* as defined by the
14 Fortran standard. If an **end simd** directive follows a *do-construct* in which several loop statements
15 share a **DO** termination statement, then the directive can only be specified for the outermost of these
16 **DO** statements.

Fortran

17 Binding

18 A **simd** region binds to the current task region. The binding thread set of the **simd** region is the
19 current team.

Description

The **simd** construct enables the execution of multiple iterations of the associated loops concurrently by means of SIMD instructions.

The **collapse** clause may be used to specify how many loops are associated with the construct. The parameter of the **collapse** clause must be a constant positive integer expression. If no **collapse** clause is present, the only loop that is associated with the loop construct is the one that immediately follows the directive.

If more than one loop is associated with the **simd** construct, then the iterations of all associated loops are collapsed into one larger iteration space that is then executed with SIMD instructions. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

If more than one loop is associated with the **simd** construct then the number of times that any intervening code between any two associated loops will be executed is unspecified but will be at least once per iteration of the loop enclosing the intervening code and at most once per iteration of the innermost loop associated with the construct.

The iteration count for each associated loop is computed before entry to the outermost loop. If execution of any associated loop changes any of the values used to compute any of the iteration counts, then the behavior is unspecified.

The integer type (or kind, for Fortran) used to compute the iteration count for the collapsed loop is implementation defined.

A SIMD loop has logical iterations numbered 0,1,...,N-1 where N is the number of loop iterations, and the logical numbering denotes the sequence in which the iterations would be executed if the associated loop(s) were executed with no SIMD instructions. At the beginning of each logical iteration, the loop iteration variable of each associated loop has the value that it would have if the set of the associated loop(s) were executed sequentially. The number of iterations that are executed concurrently at any given time is implementation defined. Each concurrent iteration will be executed by a different SIMD lane. Each set of concurrent iterations is a SIMD chunk. Lexical forward dependencies in the iterations of the original loop must be preserved within each SIMD chunk.

The **safelen** clause specifies that no two concurrent iterations within a SIMD chunk can have a distance in the logical iteration space that is greater than or equal to the value given in the clause. The parameter of the **safelen** clause must be a constant positive integer expression. The **simdlen** clause specifies the preferred number of iterations to be executed concurrently. The parameter of the **simdlen** clause must be a constant positive integer expression.

C / C++

The **aligned** clause declares that the object to which each list item points is aligned to the number of bytes expressed in the optional parameter of the **aligned** clause.

C / C++

Fortran

The **aligned** clause declares that the location of each list item is aligned to the number of bytes expressed in the optional parameter of the **aligned** clause.

Fortran

The optional parameter of the **aligned** clause, *alignment*, must be a constant positive integer expression. If no optional parameter is specified, implementation-defined default alignments for SIMD instructions on the target platforms are assumed.

The **nontemporal** clause specifies that accesses to the storage locations to which the list items refer have low temporal locality across the iterations in which those storage locations are accessed.

Restrictions

- There must be no OpenMP directive in the region between any associated loops.
- If a **collapse** clause is specified, exactly one loop must occur in the region at each nesting level up to the number of loops specified by the parameter of the **collapse** clause.
- If the **ordered** clause is present, all loops associated with the construct must be perfectly nested; that is there must be no intervening code between any two loops.
- The associated loops must be structured blocks.
- A program that branches into or out of a **simd** region is non-conforming.
- Only one **collapse** clause can appear on a **simd** directive.
- A *list-item* cannot appear in more than one **aligned** clause.
- Only one **safelen** clause can appear on a **simd** directive.
- Only one **simdlen** clause can appear on a **simd** directive.
- If both **simdlen** and **safelen** clauses are specified, the value of the **simdlen** parameter must be less than or equal to the value of the **safelen** parameter.
- A *modifier* may not be specified on a **linear** clause.
- An **atomic** construct or an **ordered** construct with the **simd** clause is the only OpenMP construct that can be encountered during execution of a **simd** region.

C / C++

- The **simd** region cannot contain calls to the **longjmp** or **setjmp** functions.

C / C++

C

1 • The type of list items appearing in the **aligned** clause must be array or pointer.

C

C++

2 • The type of list items appearing in the **aligned** clause must be array, pointer, reference to
3 array, or reference to pointer.

4 • No exception can be raised in the **simd** region.

C++

Fortran

5 • The *do-loop* iteration variable must be of type **integer**.

6 • The *do-loop* cannot be a **DO WHILE** or a **DO** loop without loop control.

7 • If a list item on the **aligned** clause has the **ALLOCATABLE** attribute, the allocation status must
8 be allocated.

9 • If a list item on the **aligned** clause has the **POINTER** attribute, the association status must be
10 associated.

11 • If the type of a list item on the **aligned** clause is either **C_PTR** or Cray pointer, the list item
12 must be defined.

Fortran

13 **Cross References**

14 • **private**, **lastprivate**, **linear** and **reduction** clauses, see Section 2.20.3 on page 249.

15 ## 2.11.2 declare simd Directive

16 **Summary**

17 The **declare simd** directive can be applied to a function (C, C++ and Fortran) or a subroutine
18 (Fortran) to enable the creation of one or more versions that can process multiple arguments using
19 SIMD instructions from a single invocation in a SIMD loop. The **declare simd** directive is a
20 declarative directive. There may be multiple **declare simd** directives for a function (C, C++,
21 Fortran) or subroutine (Fortran).

Syntax

The syntax of the **declare simd** directive is as follows:

C / C++

```
#pragma omp declare simd [clause[ [,] clause] ... ] new-line
[#pragma omp declare simd [clause[ [,] clause] ... ] new-line]
[ ... ]
    function definition or declaration
```

where *clause* is one of the following:

simdlen(*length***)**

linear(*linear-list*[**:** *linear-step*]**)**

aligned(*argument-list*[**:** *alignment*]**)**

uniform(*argument-list***)**

inbranch

notinbranch

C / C++

Fortran

```
!$omp declare simd [ (proc-name) ] [clause[ [,] clause] ... ]
```

where *clause* is one of the following:

simdlen(*length***)**

linear(*linear-list*[**:** *linear-step*]**)**

aligned(*argument-list*[**:** *alignment*]**)**

uniform(*argument-list***)**

inbranch

notinbranch

Fortran

Description

C / C++

The use of one or more **declare simd** directives immediately prior to a function declaration or definition enables the creation of corresponding SIMD versions of the associated function that can be used to process multiple arguments from a single invocation in a SIMD loop concurrently.

The expressions appearing in the clauses of each directive are evaluated in the scope of the arguments of the function declaration or definition.

C / C++

Fortran

The use of one or more **declare simd** directives for a specified subroutine or function enables the creation of corresponding SIMD versions of the subroutine or function that can be used to process multiple arguments from a single invocation in a SIMD loop concurrently.

Fortran

If a SIMD version is created, the number of concurrent arguments for the function is determined by the **simdlen** clause. If the **simdlen** clause is used its value corresponds to the number of concurrent arguments of the function. The parameter of the **simdlen** clause must be a constant positive integer expression. Otherwise, the number of concurrent arguments for the function is implementation defined.

C++

The special *this* pointer can be used as if was one of the arguments to the function in any of the **linear**, **aligned**, or **uniform** clauses.

C++

The **uniform** clause declares one or more arguments to have an invariant value for all concurrent invocations of the function in the execution of a single SIMD loop.

C / C++

The **aligned** clause declares that the object to which each list item points is aligned to the number of bytes expressed in the optional parameter of the **aligned** clause.

C / C++

Fortran

1 The **aligned** clause declares that the target of each list item is aligned to the number of bytes
2 expressed in the optional parameter of the **aligned** clause.

Fortran

3 The optional parameter of the **aligned** clause, *alignment*, must be a constant positive integer
4 expression. If no optional parameter is specified, implementation-defined default alignments for
5 SIMD instructions on the target platforms are assumed.

6 The **inbranch** clause specifies that the SIMD version of the function will always be called from
7 inside a conditional statement of a SIMD loop. The **notinbranch** clause specifies that the SIMD
8 version of the function will never be called from inside a conditional statement of a SIMD loop. If
9 neither clause is specified, then the SIMD version of the function may or may not be called from
10 inside a conditional statement of a SIMD loop.

11 Restrictions

12 • Each argument can appear in at most one **uniform** or **linear** clause.

13 • At most one **simdlen** clause can appear in a **declare simd** directive.

14 • Either **inbranch** or **notinbranch** may be specified, but not both.

15 • When a *linear-step* expression is specified in a **linear** clause it must be either a constant integer
16 expression or an integer-typed parameter that is specified in a **uniform** clause on the directive.

17 • The function or subroutine body must be a structured block.

18 • The execution of the function or subroutine, when called from a SIMD loop, cannot result in the
19 execution of an OpenMP construct except for an **ordered** construct with the **simd** clause or an
20 **atomic** construct.

21 • The execution of the function or subroutine cannot have any side effects that would alter its
22 execution for concurrent iterations of a SIMD chunk.

23 • A program that branches into or out of the function is non-conforming.

C / C++

24 • If the function has any declarations, then the **declare simd** construct for any declaration that
25 has one must be equivalent to the one specified for the definition. Otherwise, the result is
26 unspecified.

27 • The function cannot contain calls to the **longjmp** or **setjmp** functions.

C / C++

C

1 • The type of list items appearing in the **aligned** clause must be array or pointer.

C

C++

2 • The function cannot contain any calls to **throw**.

3 • The type of list items appearing in the **aligned** clause must be array, pointer, reference to
4 array, or reference to pointer.

C++

Fortran

5 • *proc-name* must not be a generic name, procedure pointer or entry name.

6 • If *proc-name* is omitted, the **declare simd** directive must appear in the specification part of a
7 subroutine subprogram or a function subprogram for which creation of the SIMD versions is
8 enabled.

9 • Any **declare simd** directive must appear in the specification part of a subroutine subprogram,
10 function subprogram or interface body to which it applies.

11 • If a **declare simd** directive is specified in an interface block for a procedure, it must match a
12 **declare simd** directive in the definition of the procedure.

13 • If a procedure is declared via a procedure declaration statement, the procedure *proc-name* should
14 appear in the same specification.

15 • If a **declare simd** directive is specified for a procedure name with explicit interface and a
16 **declare simd** directive is also specified for the definition of the procedure then the two
17 **declare simd** directives must match. Otherwise the result is unspecified.

18 • Procedure pointers may not be used to access versions created by the **declare simd** directive.

19 • The type of list items appearing in the **aligned** clause must be **C_PTR** or Cray pointer, or the
20 list item must have the **POINTER** or **ALLOCATABLE** attribute.

Fortran

Cross References

- **reduction** clause, see Section 2.20.4.4 on page 272.
- **linear** clause, see Section 2.20.3.6 on page 262.

2.11.3 Loop SIMD Construct

Summary

The loop SIMD construct specifies that the iterations of one or more associated loops will be distributed across threads that already exist in the team and that the iterations executed by each thread can also be executed concurrently using SIMD instructions. The loop SIMD construct is a composite construct.

Syntax

C / C++

```
#pragma omp for simd [clause[ [,] clause] ... ] new-line
    for-loops
```

where *clause* can be any of the clauses accepted by the **for** or **simd** directives with identical meanings and restrictions.

C / C++

Fortran

```
!$omp do simd [clause[ [,] clause] ... ]
    do-loops
[!$omp end do simd [nowait] ]
```

where *clause* can be any of the clauses accepted by the **simd** or **do** directives, with identical meanings and restrictions.

If an **end do simd** directive is not specified, an **end do simd** directive is assumed at the end of the *do-loops*.

Fortran

Description

The loop SIMD construct will first distribute the iterations of the associated loop(s) across the implicit tasks of the parallel region in a manner consistent with any clauses that apply to the loop construct. The resulting chunks of iterations will then be converted to a SIMD loop in a manner consistent with any clauses that apply to the **`simd`** construct. The effect of any clause that applies to both constructs is as if it were applied to both constructs separately except the **`collapse`** clause, which is applied once.

Events

This composite construct generates the same events as the loop construct.

Tool Callbacks

This composite construct dispatches the same callbacks as the loop construct.

Restrictions

All restrictions to the loop construct and the **`simd`** construct apply to the loop SIMD construct. In addition, the following restrictions apply:

- No **`ordered`** clause with a parameter can be specified.
- A list item may appear in a **`linear`** or **`firstprivate`** clause but not both.

Cross References

- loop construct, see Section 2.10.1 on page 78.
- **`simd`** construct, see Section 2.11.1 on page 96.
- Data attribute clauses, see Section 2.20.3 on page 249.
- Events and tool callbacks for the loop construct, see Section 2.10.1 on page 78.

2.12 concurrent Construct

Summary

A **concurrent** construct asserts to the compiler that all iterations of the associated loops are free of data dependencies and can be safely run concurrently in any order.

Syntax

C / C++

The syntax of the **concurrent** construct is as follows:

```
#pragma omp concurrent [clause[ [,] clause] ... ] new-line
    for-loops
```

where *clause* is one of the following:

private(*list***)**

firstprivate(*list***)**

reduction(*reduction-identifier* **:** *list***)**

levels(*n***)**

The **concurrent** directive places restrictions on the structure of all associated *for-loops*. Specifically, all associated *for-loops* must have *canonical loop form* (see Section 2.9 on page 75).

C / C++

Fortran

The syntax of the **concurrent** construct is as follows:

```
!$omp concurrent [clause[ [,] clause] ... ]
    do-loops
[!$omp end concurrent]
```

where *clause* is one of the following:

private(*list***)**

firstprivate(*list***)**

reduction(*reduction-identifier* **:** *list***)**

levels(*n***)**

If an **end concurrent** directive is not specified, an **end concurrent** directive is assumed at the end of the *do-loops*.

Any associated *do-loop* must be a *do-construct* or an *inner-shared-do-construct* as defined by the Fortran standard. If an **end concurrent** directive follows a *do-construct* in which several loop statements share a **DO** termination statement, then the directive can only be specified for the outermost of these **DO** statements.

If any of the loop iteration variables would otherwise be shared, they are implicitly made private on the **concurrent** construct.

Fortran

If a **private** or **firstprivate** clause appears on the **concurrent** construct and the iterations of the associated loops are executed concurrently, then a private copy of the listed variables must be created for each thread team, thread, or SIMD lane that executes the iterations.

If a **reduction** clause appears on the **concurrent** construct and an implementation executes the iterations of the associated loops concurrently, then it must perform a parallel reduction with respect to the parallel execution units (teams, threads, or SIMD lanes) used to execute the loop iterations. The result of the reduction operation must be available immediately following the end of the associated loops.

Binding

If the iterations of the loops associated with the **concurrent** construct are distributed to multiple thread teams, then the binding thread set for the **concurrent** region is the set of master threads executing an enclosing **teams** region and the **concurrent** region binds to this **teams** region. Otherwise the **concurrent** region binds to the current task region and the binding thread set is the current team.

Description

The **concurrent** construct is associated with a loop nest consisting of one or more loops that follow the directive. The construct asserts that all iterations of the associated loop nest are independent with respect to each other, given that privatization and reduction clauses are honored, and may be executed concurrently in any order.

If more than one thread encounters an **concurrent** construct, the implementation must ensure that each logical iteration of the corresponding loops is executed exactly once.

The **levels** clause may be used to specify how many loops are associated with the **concurrent** construct. The parameter of the **levels** clause must be a constant positive integer expression. If a **levels** clause is present then for the next N nested loops all iterations of the loops may be executed concurrently with any other iteration in the combined iteration space of the associated loops in any order. If no **levels** clause appears on the construct, the **concurrent** directive applies to only the loop that immediately follows the directive.

Restrictions

Restrictions to the **concurrent** construct are as follows:

- An implementation may not generate a new league of thread teams as the result of a **concurrent** construct.
- Only one **levels** clause can appear on a **concurrent** directive.
- A **concurrent** region may not contain calls to functions containing OpenMP directives.
- A **concurrent** region may not contain calls to the OpenMP Runtime API.

C / C++

- The associated for-loops must be structured blocks.
- No statement can branch to any associated **concurrent** statement.

C / C++

Fortran

- The associated do-loops must be structured blocks.
- No statement in the associated loops other than the DO statements can cause a branch out of the loops.

Fortran

1 **Cross References**

2 • The Loop construct, see Section 2.10.1 on page 78.
3 • **distribute** constrict, see Section 2.15.10 on page 160.
4 • SIMD constructs, see Section 2.11 on page 96.
5 • The **single** construct, see Section 2.10.3 on page 90.

6 # 2.13 Tasking Constructs

7 ## 2.13.1 `task` Construct

8 **Summary**

9 The **task** construct defines an explicit task.

10 **Syntax**

C / C++

11 The syntax of the **task** construct is as follows:

```
#pragma omp task [clause[ [,] clause] ... ] new-line
    structured-block
```

1 where *clause* is one of the following:

2 **if(**[**task** **:**] *scalar-expression***)**

3 **final(***scalar-expression***)**

4 **untied**

5 **default(shared | none)**

6 **mergeable**

7 **private(***list***)**

8 **firstprivate(***list***)**

9 **shared(***list***)**

10 **in_reduction(***reduction-identifier* **:** *list***)**

11 **depend(***dependence-type* **:** *locator-list* **[:** *iterators-definition***])**

12 **priority(***priority-value***)**

13 **allocate(**[*allocator:*]*list***)**

C / C++

Fortran

14 The syntax of the **task** construct is as follows:

```
!$omp task [clause[ [,] clause] ... ]
    structured-block
!$omp end task
```

15 where *clause* is one of the following:

16 **if(**[**task** **:**] *scalar-logical-expression***)**

17 **final(***scalar-logical-expression***)**

18 **untied**

19 **default(private | firstprivate | shared | none)**

20 **mergeable**

21 **private(***list***)**

22 **firstprivate(***list***)**

23 **shared(***list***)**

in_reduction(*reduction-identifier* **:** *list***)**

depend(*dependence-type* **:** *locator-list* **[:***iterators-definition***])**

priority(*priority-value***)**

allocate(*[allocator:]list***)**

Fortran

Binding

The binding thread set of the **task** region is the current team. A **task** region binds to the innermost enclosing **parallel** region.

Description

The **task** construct is a *task generating construct*. When a thread encounters a **task** construct, an explicit task is generated from the code for the associated structured block. The data environment of the task is created according to the data-sharing attribute clauses on the **task** construct, per-data environment ICVs, and any defaults that apply.

The encountering thread may immediately execute the task, or defer its execution. In the latter case, any thread in the team may be assigned the task. Completion of the task can be guaranteed using task synchronization constructs. If a **task** construct is encountered during execution of an outer task, the generated **task** region associated with this construct is not a part of the outer task region unless the generated task is an included task.

When an **if** clause is present on a **task** construct, and the **if** clause expression evaluates to *false*, an undeferred task is generated, and the encountering thread must suspend the current task region, for which execution cannot be resumed until the generated task is completed. The use of a variable in an **if** clause expression of a **task** construct causes an implicit reference to the variable in all enclosing constructs.

When a **final** clause is present on a **task** construct and the **final** clause expression evaluates to *true*, the generated task will be a final task. All **task** constructs encountered during execution of a final task will generate final and included tasks. Note that the use of a variable in a **final** clause expression of a **task** construct causes an implicit reference to the variable in all enclosing constructs.

The **if** clause expression and the **final** clause expression are evaluated in the context outside of the **task** construct, and no ordering of those evaluations is specified.

A thread that encounters a task scheduling point within the **task** region may temporarily suspend the **task** region. By default, a task is tied and its suspended **task** region can only be resumed by the thread that started its execution. If the **untied** clause is present on a **task** construct, any thread in the team can resume the **task** region after a suspension. The **untied** clause is ignored

if a **final** clause is present on the same **task** construct and the **final** clause expression evaluates to *true*, or if a task is an included task.

The **task** construct includes a task scheduling point in the task region of its generating task, immediately following the generation of the explicit task. Each explicit **task** region includes a task scheduling point at its point of completion.

When the **mergeable** clause is present on a **task** construct, the generated task is a *mergeable task*.

The **priority** clause is a hint for the priority of the generated task. The *priority-value* is a non-negative integer expression that provides a hint for task execution order. Among all tasks ready to be executed, higher priority tasks (those with a higher numerical value in the **priority** clause expression) are recommended to execute before lower priority ones. The default *priority-value* when no **priority** clause is specified is zero (the lowest priority). If a value is specified in the **priority** clause that is higher than the *max-task-priority-var* ICV then the implementation will use the value of that ICV. A program that relies on task execution order being determined by this *priority-value* may have unspecified behavior.

Note – When storage is shared by an explicit **task** region, the programmer must ensure, by adding proper synchronization, that the storage does not reach the end of its lifetime before the explicit **task** region completes its execution.

Events

The *task-create* event occurs when a thread encounters a construct that causes a new task to be created. The event occurs after the task is initialized but before it begins execution or is deferred.

Tool Callbacks

A thread dispatches a registered **ompt_callback_task_create** callback for each occurrence of a *task-create* event in the context of the encountering task. This callback has the type signature **ompt_callback_task_create_t**. In the dispatched callback, **(task_type & ompt_task_explicit)** always evaluates to *true*. If the task is an undeferred task, then **(task_type & ompt_task_undeferred)** evaluates to *true*. If the task is a final task, **(task_type & ompt_task_final)** evaluates to *true*. If the task is an untied task, **(task_type & ompt_task_untied)** evaluates to *true*. If the task is a mergeable task, **(task_type & ompt_task_mergeable)** evaluates to *true*. If the task is a merged task, **(task_type & ompt_task_merged)** evaluates to *true*.

Restrictions

Restrictions to the **task** construct are as follows:

- A program that branches into or out of a **task** region is non-conforming.
- A program must not depend on any ordering of the evaluations of the clauses of the **task** directive, or on any side effects of the evaluations of the clauses.
- At most one **if** clause can appear on the directive.
- At most one **final** clause can appear on the directive.
- At most one **priority** clause can appear on the directive.

C / C++

- A throw executed inside a **task** region must cause execution to resume within the same **task** region, and the same thread that threw the exception must catch it.

C / C++

Fortran

- Unsynchronized use of Fortran I/O statements by multiple tasks on the same unit has unspecified behavior

Fortran

Cross References

- Task scheduling constraints, see Section 2.13.6 on page 125.
- **depend** clause, see Section 2.18.10 on page 225.
- **if** Clause, see Section 2.17 on page 192.
- Data-sharing attribute clauses, Section 2.20.3 on page 249.
- **ompt_callback_task_create_t**, see Section 4.1.4.2.7 on page 412.

2.13.2 taskloop Construct

Summary

The **taskloop** construct specifies that the iterations of one or more associated loops will be executed in parallel using explicit tasks. The iterations are distributed across tasks generated by the construct and scheduled to be executed.

1 **Syntax**

C / C++

2 The syntax of the **taskloop** construct is as follows:

```
#pragma omp taskloop [clause[[,] clause] ...] new-line
    for-loops
```

3 where *clause* is one of the following:

4 **if(**[**taskloop :**] *scalar-expr***)**

5 **shared(***list***)**

6 **private(***list***)**

7 **firstprivate(***list***)**

8 **lastprivate(***list***)**

9 **reduction(***reduction-identifier* **:** *list***)**

10 **in_reduction(***reduction-identifier* **:** *list***)**

11 **default(shared | none)**

12 **grainsize(***grain-size***)**

13 **num_tasks(***num-tasks***)**

14 **collapse(***n***)**

15 **final(***scalar-expr***)**

16 **priority(***priority-value***)**

17 **untied**

18 **mergeable**

19 **nogroup**

20 **allocate(**[*allocator:*]*list***)**

21 The **taskloop** directive places restrictions on the structure of all associated *for-loops*.
22 Specifically, all associated *for-loops* must have canonical loop form (see Section 2.9 on page 75).

C / C++

Fortran

The syntax of the **taskloop** construct is as follows:

```
!$omp taskloop [clause[[,] clause] ...]
    do-loops
[!$omp end taskloop]
```

where *clause* is one of the following:

if([**taskloop :**] *scalar-logical-expr***)**

shared(*list***)**

private(*list***)**

firstprivate(*list***)**

lastprivate(*list***)**

reduction(*reduction-identifier* **:** *list***)**

in_reduction(*reduction-identifier* **:** *list***)**

default(private | firstprivate | shared | none)

grainsize(*grain-size***)**

num_tasks(*num-tasks***)**

collapse(*n***)**

final(*scalar-logical-expr***)**

priority(*priority-value***)**

untied

mergeable

nogroup

allocate([*allocator:*]*list***)**

If an **end taskloop** directive is not specified, an **end taskloop** directive is assumed at the end of the *do-loops*.

Any associated *do-loop* must be *do-construct* or an *inner-shared-do-construct* as defined by the Fortran standard. If an **end taskloop** directive follows a *do-construct* in which several loop statements share a **DO** termination statement, then the directive can only be specified for the outermost of these **DO** statements.

If any of the loop iteration variables would otherwise be shared, they are implicitly made private for the loop-iteration tasks generated by the **taskloop** construct. Unless the loop iteration variables are specified in a **lastprivate** clause on the **taskloop** construct, their values after the loop are unspecified.

Fortran

Binding

The binding thread set of the **taskloop** region is the current team. A **taskloop** region binds to the innermost enclosing **parallel** region.

Description

The **taskloop** construct is a *task generating construct*. When a thread encounters a **taskloop** construct, the construct partitions the associated loops into explicit tasks for parallel execution of the loops' iterations. The data environment of each generated task is created according to the data-sharing attribute clauses on the **taskloop** construct, per-data environment ICVs, and any defaults that apply. The order of the creation of the loop tasks is unspecified. Programs that rely on any execution order of the logical loop iterations are non-conforming.

By default, the **taskloop** construct executes as if it was enclosed in a **taskgroup** construct with no statements or directives outside of the **taskloop** construct. Thus, the **taskloop** construct creates an implicit **taskgroup** region. If the **nogroup** clause is present, no implicit **taskgroup** region is created.

If a **reduction** clause is present on the **taskloop** construct, the behavior is as if a **task_reduction** clause with the same reduction operator and list items was applied to the implicit **taskgroup** construct enclosing the **taskloop** construct. Furthermore, the **taskloop** construct executes as if each generated task was defined by a **task** construct on which an **in_reduction** clause with the same reduction operator and list items is present. Thus, the generated tasks are participants of the reduction defined by the **task_reduction** clause that was applied to the implicit **taskgroup** construct.

If an **in_reduction** clause is present on the **taskloop** construct, the behavior is as if each generated task was defined by a **task** construct on which an **in_reduction** clause with the same reduction operator and list items is present. Thus, the generated tasks are participants of a reduction previously defined by a reduction scoping clause.

If a **grainsize** clause is present on the **taskloop** construct, the number of logical loop iterations assigned to each generated task is greater than or equal to the minimum of the value of the *grain-size* expression and the number of logical loop iterations, but less than two times the value of the *grain-size* expression.

The parameter of the **grainsize** clause must be a positive integer expression. If **num_tasks** is specified, the **taskloop** construct creates as many tasks as the minimum of the *num-tasks* expression and the number of logical loop iterations. Each task must have at least one logical loop iteration. The parameter of the **num_tasks** clause must evaluate to a positive integer. If neither a **grainsize** nor **num_tasks** clause is present, the number of loop tasks generated and the number of logical loop iterations assigned to these tasks is implementation defined.

The **collapse** clause may be used to specify how many loops are associated with the **taskloop** construct. The parameter of the **collapse** clause must be a constant positive integer expression. If no **collapse** clause is present or its parameter is 1, the only loop that is associated with the **taskloop** construct is the one that immediately follows the **taskloop** directive. If a **collapse** clause is specified with a parameter value greater than 1 and more than one loop is associated with the **taskloop** construct, then the iterations of all associated loops are collapsed into one larger iteration space that is then divided according to the **grainsize** and **num_tasks** clauses. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

If more than one loop is associated with the **taskloop** construct then the number of times that any intervening code between any two associated loops will be executed is unspecified but will be at least once per iteration of the loop enclosing the intervening code and at most once per iteration of the innermost loop associated with the construct.

A taskloop loop has logical iterations numbered 0,1,...,N-1 where N is the number of loop iterations, and the logical numbering denotes the sequence in which the iterations would be executed if the set of associated loop(s) were executed sequentially. At the beginning of each logical iteration, the loop iteration variable of each associated loop has the value that it would have if the set of the associated loop(s) were executed sequentially.

The iteration count for each associated loop is computed before entry to the outermost loop. If execution of any associated loop changes any of the values used to compute any of the iteration counts, then the behavior is unspecified.

The integer type (or kind, for Fortran) used to compute the iteration count for the collapsed loop is implementation defined.

When an **if** clause is present on a **taskloop** construct, and if the **if** clause expression evaluates to *false*, undeferred tasks are generated. The use of a variable in an **if** clause expression of a **taskloop** construct causes an implicit reference to the variable in all enclosing constructs.

When a **final** clause is present on a **taskloop** construct and the **final** clause expression evaluates to *true*, the generated tasks will be final tasks. The use of a variable in a **final** clause expression of a **taskloop** construct causes an implicit reference to the variable in all enclosing

constructs.

When a **priority** clause is present on a **taskloop** construct, the generated tasks have the *priority-value* as if it was specified for each individual task. If the **priority** clause is not specified, tasks generated by the **taskloop** construct have the default task priority (zero).

If the **untied** clause is specified, all tasks generated by the **taskloop** construct are untied tasks.

When the **mergeable** clause is present on a **taskloop** construct, each generated task is a *mergeable task*.

C++

For **firstprivate** variables of class type, the number of invocations of copy constructors to perform the initialization is implementation-defined.

C++

Note – When storage is shared by a **taskloop** region, the programmer must ensure, by adding proper synchronization, that the storage does not reach the end of its lifetime before the **taskloop** region and its descendant tasks complete their execution.

Events

The *taskloop-begin* event occurs after a task encounters a **taskloop** construct but before any other events that may trigger as a consequence of executing the **taskloop**. Specifically, a *taskloop-begin* event for a **taskloop** will precede the *taskgroup-begin* that occurs unless a **nogroup** clause is present. Regardless of whether an implicit taskgroup is present, a *taskloop-begin* will always precede any *task-create* events for generated tasks.

The *taskloop-end* event occurs after a **taskloop** region finishes execution but before resuming execution of the encountering task.

Tool Callbacks

A thread dispatches a registered **ompt_callback_work** callback for each occurrence of a *taskloop-begin* and *taskloop-end* event in that thread. The callback occurs in the context of the encountering task. The callback has type signature **ompt_callback_work_t**. The callback receives **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate, and **ompt_work_taskloop** as its *wstype* argument.

Restrictions

The restrictions of the **taskloop** construct are as follows:

- A program that branches into or out of a **taskloop** region is non-conforming.
- There must be no OpenMP directive in the region between any associated loops.
- If a **collapse** clause is specified, exactly one loop must occur in the region at each nesting level up to the number of loops specified by the parameter of the **collapse** clause.
- If the **ordered** clause is present, all loops associated with the construct must be perfectly nested; that is there must be no intervening code between any two loops.
- If a **reduction** clause is present on the **taskloop** directive, the **nogroup** clause must not be specified.
- The same list item cannot appear in both a **reduction** and an **in_reduction** clause.
- At most one **grainsize** clause can appear on a **taskloop** directive.
- At most one **num_tasks** clause can appear on a **taskloop** directive.
- The **grainsize** clause and **num_tasks** clause are mutually exclusive and may not appear on the same **taskloop** directive.
- At most one **collapse** clause can appear on a **taskloop** directive.
- At most one **if** clause can appear on the directive.
- At most one **final** clause can appear on the directive.
- At most one **priority** clause can appear on the directive.

Cross References

- **task** construct, Section 2.13.1 on page 110.
- **taskgroup** construct, Section 2.18.6 on page 204.
- Data-sharing attribute clauses, Section 2.20.3 on page 249.
- **if** Clause, see Section 2.17 on page 192.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.1.3.4.10 on page 397.
- **ompt_work_taskloop**, see Section 4.1.3.4.13 on page 398.
- **ompt_callback_work_t**, see Section 4.1.4.2.16 on page 423.

2.13.3 `taskloop simd` Construct

Summary

The **`taskloop simd`** construct specifies a loop that can be executed concurrently using SIMD instructions and that those iterations will also be executed in parallel using explicit tasks. The **`taskloop simd`** construct is a composite construct.

Syntax

C / C++

The syntax of the **`taskloop simd`** construct is as follows:

```
#pragma omp taskloop simd [clause[[,] clause] ...] new-line
      for-loops
```

where *clause* can be any of the clauses accepted by the **`taskloop`** or **`simd`** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **`taskloop simd`** construct is as follows:

```
!$omp taskloop simd [clause[[,] clause] ...]
      do-loops
[!$omp end taskloop simd]
```

where *clause* can be any of the clauses accepted by the **`taskloop`** or **`simd`** directives with identical meanings and restrictions.

If an **`end taskloop simd`** directive is not specified, an **`end taskloop simd`** directive is assumed at the end of the *do-loops*.

Fortran

Binding

The binding thread set of the **`taskloop simd`** region is the current team. A **`taskloop simd`** region binds to the innermost enclosing parallel region.

Description

The **taskloop simd** construct will first distribute the iterations of the associated loop(s) across tasks in a manner consistent with any clauses that apply to the **taskloop** construct. The resulting tasks will then be converted to a SIMD loop in a manner consistent with any clauses that apply to the **simd** construct, except for the **collapse** clause. For the purposes of each task's conversion to a SIMD loop, the **collapse** clause is ignored and the effect of any **in_reduction** clause is as if a **reduction** clause with the same reduction operator and list items is present on the construct.

Events

This composite construct generates the same events as the **taskloop** construct.

Tool Callbacks

This composite construct dispatches the same callbacks as the **taskloop** construct.

Restrictions

- The restrictions for the **taskloop** and **simd** constructs apply.

Cross References

- **taskloop** construct, see Section 2.13.2 on page 114.
- **simd** construct, see Section 2.11.1 on page 96.
- Data-sharing attribute clauses, see Section 2.20.3 on page 249.
- Events and tool callbacks for **taskloop** construct, see Section 2.13.2 on page 114.

2.13.4 `taskyield` Construct

Summary

The **`taskyield`** construct specifies that the current task can be suspended in favor of execution of a different task. The **`taskyield`** construct is a stand-alone directive.

Syntax

C / C++

The syntax of the **`taskyield`** construct is as follows:

```
#pragma omp taskyield new-line
```

C / C++

Fortran

The syntax of the **`taskyield`** construct is as follows:

```
!$omp taskyield
```

Fortran

Binding

A **`taskyield`** region binds to the current task region. The binding thread set of the **`taskyield`** region is the current team.

Description

The **`taskyield`** region includes an explicit task scheduling point in the current task region.

Cross References

- Task scheduling, see Section 2.13.6 on page 125.

2.13.5 Initial Task

Events

No events are associated with the implicit parallel region in each initial thread.

The *initial-thread-begin* event occurs in an initial thread after the OpenMP runtime invokes the tool initializer but before the initial thread begins to execute the first OpenMP region in the initial task.

The *initial-task-create* event occurs after an *initial-thread-begin* event but before the first OpenMP region in the initial task begins to execute.

The *initial-thread-end* event occurs as the final event in an initial thread at the end of an initial task immediately prior to invocation of the tool finalizer.

Tool Callbacks

A thread dispatches a registered **ompt_callback_thread_begin** callback for the *initial-thread-begin* event in an initial thread. The callback occurs in the context of the initial thread. The callback has type signature **ompt_callback_thread_begin_t**. The callback receives **ompt_thread_initial** as its *thread_type* argument.

A thread dispatches a registered **ompt_callback_task_create** callback for each occurrence of a *initial-task-create* event in the context of the encountering task. This callback has the type signature **ompt_callback_task_create_t**. The callback receives **ompt_task_initial** as its *type* argument. The implicit parallel region does not dispatch a **ompt_callback_parallel_begin** callback; however, the implicit parallel region can be initialized within this **ompt_callback_task_create** callback.

A thread dispatches a registered **ompt_callback_thread_end** callback for the *initial-thread-end* event in that thread. The callback occurs in the context of the thread. The callback has type signature **ompt_callback_thread_end_t**. The implicit parallel region does not dispatch a **ompt_callback_parallel_end** callback; however, the implicit parallel region can be finalized within this **ompt_callback_thread_end** callback.

Cross References

- **ompt_task_initial**, see Section 4.1.3.4.16 on page 400.
- **ompt_callback_thread_begin_t**, see Section 4.1.4.2.1 on page 406.
- **ompt_callback_thread_end_t**, see Section 4.1.4.2.2 on page 407.
- **ompt_callback_task_create_t**, see Section 4.1.4.2.7 on page 412.

2.13.6 Task Scheduling

Whenever a thread reaches a task scheduling point, the implementation may cause it to perform a task switch, beginning or resuming execution of a different task bound to the current team. Task scheduling points are implied at the following locations:

- the point immediately following the generation of an explicit task;
- after the point of completion of a **task** region;
- in a **taskyield** region;
- in a **taskwait** region;
- at the end of a **taskgroup** region;
- in an implicit and explicit **barrier** region;
- the point immediately following the generation of a **target** region;
- at the beginning and end of a **target data** region;
- in a **target update** region;
- in a **target enter data** region;
- in a **target exit data** region;
- in the **omp_target_memcpy** routine;
- in the **omp_target_memcpy_rect** routine;

When a thread encounters a task scheduling point it may do one of the following, subject to the *Task Scheduling Constraints* (below):

- begin execution of a tied task bound to the current team
- resume any suspended task region, bound to the current team, to which it is tied
- begin execution of an untied task bound to the current team
- resume any suspended untied task region bound to the current team.

If more than one of the above choices is available, it is unspecified as to which will be chosen.

Task Scheduling Constraints are as follows:

1. An included task is executed immediately after generation of the task.
2. Scheduling of new tied tasks is constrained by the set of task regions that are currently tied to the thread, and that are not suspended in a **barrier** region. If this set is empty, any new tied task may be scheduled. Otherwise, a new tied task may be scheduled only if it is a descendent task of every task in the set.
3. A dependent task shall not be scheduled until its task dependences are fulfilled.

4. A task shall not be scheduled while any task with which it is mutually exclusive has been scheduled, but has not yet completed.

5. When an explicit task is generated by a construct containing an **if** clause for which the expression evaluated to *false*, and the previous constraints are already met, the task is executed immediately after generation of the task.

A program relying on any other assumption about task scheduling is non-conforming.

Note – Task scheduling points dynamically divide task regions into parts. Each part is executed uninterrupted from start to end. Different parts of the same task region are executed in the order in which they are encountered. In the absence of task synchronization constructs, the order in which a thread executes parts of different schedulable tasks is unspecified.

A correct program must behave correctly and consistently with all conceivable scheduling sequences that are compatible with the rules above.

For example, if **threadprivate** storage is accessed (explicitly in the source code or implicitly in calls to library routines) in one part of a task region, its value cannot be assumed to be preserved into the next part of the same task region if another schedulable task exists that modifies it.

As another example, if a lock acquire and release happen in different parts of a task region, no attempt should be made to acquire the same lock in any part of another task that the executing thread may schedule. Otherwise, a deadlock is possible. A similar situation can occur when a **critical** region spans multiple parts of a task and another schedulable task contains a **critical** region with the same name.

The use of threadprivate variables and the use of locks or critical sections in an explicit task with an **if** clause must take into account that when the **if** clause evaluates to *false*, the task is executed immediately, without regard to *Task Scheduling Constraint* 2.

Events

The *task-schedule* event occurs in a thread when the thread switches tasks at a task scheduling point; no event occurs when switching to or from a merged task.

Tool Callbacks

A thread dispatches a registered **ompt_callback_task_schedule** callback for each occurrence of a *task-schedule* event in the context of the task that begins or resumes. This callback has the type signature **ompt_callback_task_schedule_t**. The argument *prior_task_status* is used to indicate the cause for suspending the prior task. This cause may be the completion of the prior task region, the encountering of a **taskyield** construct, or the encountering of an active cancellation point.

Cross References

- **ompt_callback_task_schedule_t**, see Section 4.1.4.2.10 on page 415.

2.14 Memory Management Directives

2.14.1 allocate Directive

Summary

The **allocate** directive specifies how a set of variables are allocated. The **allocate** directive is a declarative directive if it is not associated with an allocation statement.

Syntax

C / C++

The syntax of the **allocate** directive is as follows:

```
#pragma omp allocate(list) [clause[ [ [,] clause] ... ]] new-line
```

where *clause* is one of the following:

allocator(*allocator***)**

where *allocator* is an expression of **const omp_allocator_t *** type.

C / C++

Fortran

The syntax of the **allocate** directive is as follows:

```
!$omp allocate(list) [clause[ [ [,] clause] ... ]]
```

or

```
!$omp allocate[(list)] clause[ [ [,] clause] ... ]
[!$omp allocate(list) clause[ [ [,] clause] ... ]]
[...]
    allocate statement
```

where *clause* is one of the following:

allocator(*allocator***)**

where *allocator* is an integer expression of **omp_allocator_kind** *kind*.

Fortran

Description

If the directive is not associated with a statement, the storage for each *list item* that appears in the directive will be provided by an allocation through a memory allocator. If no clause is specified then the memory allocator specified by the *def-allocator-var* ICV will be used. If the **allocator** clause is specified, the memory allocator specified in the clause will be used. If a memory allocator is unable to fulfill the allocation request for any list item the result of the behavior is implementation defined.

The scope of this allocation is that of the list item in the base language. At the end of the scope for a given list item the memory allocator used to allocate that list item deallocates the storage.

Fortran

If the directive is associated with an **allocate** statement, the same list items appearing in the directive list and the **allocate** statement list are allocated with the memory allocator of the directive. If no list items are specified then all variables listed in the **allocate** statement are allocated with the memory allocator of the directive.

Fortran

1

Restrictions

2 3 • A variable that is part of another variable (as an array or structure element) cannot appear in an **allocate** directive.

4 • The directive must appear in the same scope of the *list item* declaration and before its first use.

5 • At most one **allocator** clause can appear on the **allocate** directive.

C / C++

6 7 • If a list item has a static storage type, only predefined memory allocator variables can be used in the **allocator** clause.

C / C++

Fortran

8 9 • List items specified in the **allocate** directive must not have the **ALLOCATABLE** attribute unless the directive is associated with an **allocate** statement.

10 11 12 • List items specified in an **allocate** directive that is associated with an **allocate** statement must be **ALLOCATABLE** variables or **POINTER** variables allocated by the **allocate** statement.

13 14 • Multiple directives can only be associated with an **allocate** statement if list items are specified on each **allocate** directive.

15 16 17 • If a list item has the **SAVE** attribute, or if the list item is a common block name, or if a list item is declared in the scope of a module, then only predefined memory allocator variables can be used in the **allocator** clause.

18 19 • **allocate** directives appearing in a **target** region must specify an **allocator** clause and the *allocator* must be a constant expression.

Fortran

20

Cross References

21 • Memory allocators, see Section 2.7 on page 64.

22 • **omp_allocator_t** and **omp_allocator_kind**, see Section 3.6.1 on page 368.

23 • *def-allocator-var* ICV, see Section 2.4.1 on page 49.

1
2.14.2 `allocate` Clause

2
Summary

3 The **`allocate`** clause specifies the memory allocator to be used to obtain storage for private
4 variables of a directive.

5
Syntax

6 The syntax of the **`allocate`** clause is as follows:

```
allocate([allocator :] list)
```

C / C++

7 where *allocator* is an expression of the **`const omp_allocator_t *`** type.

C / C++

Fortran

8 where *allocator* is an integer expression of the **`omp_allocator_kind`** *kind*.

Fortran

9
Description

10 The storage for new list items that arise from *list items* that appear in the directive will be provided
11 through a memory allocator. If an *allocator* is specified in the clause this will be the memory
12 allocator used for allocations. For all directives except for the **`target`** directive, if no *allocator* is
13 specified in the clause then the memory allocator specified by the *def-allocator-var* ICV will be
14 used for the *list items* specified in the **`allocate`** clause. If a memory allocator is unable to fulfill
15 the allocation request for any list item the result of the behavior is implementation defined.

16
Restrictions

17 • For any list item that is specified in the **`allocate`** clause on a directive, a data-sharing attribute
18 clause that may create a private copy of that list item must be specified on the same directive.

19 • For **`task`**, **`taskloop`** or **`target`** directives, allocation requests to the
20 **`omp_thread_mem_alloc`** memory allocator result in unspecified behavior.

21 • **`allocate`** clauses appearing in a **`target`** construct or in a **`target`** region must specify an
22 *allocator* and it must be a constant expression.

1 **Cross References**

2 • Memory allocators, see Section 2.7 on page 64.

3 • **omp_allocator_t** and **omp_allocator_kind**, see Section 3.6.1 on page 368.

4 • *def-allocator-var* ICV, see Section 2.4.1 on page 49.

5 2.15 Device Constructs

6 2.15.1 Device Initialization

7 **Events**

8 The *device-initialize* event occurs in a thread that encounters the first **target**, **target data**, or
9 **target enter data** construct associated with a particular target device after the thread
10 initiates initialization of OpenMP on the device and the device's OpenMP initialization, which may
11 include device-side tool initialization, completes.

12 The *device-load* event for a code block for a target device occurs in some thread before any thread
13 executes code from that code block on that target device.

14 The *device-unload* event for a target device occurs in some thread whenever a code block is
15 unloaded from the device.

16 The *device-finalize* event for a target device that has been initialized occurs in some thread before
17 an OpenMP implementation shuts down.

18 **Tool Callbacks**

19 A thread dispatches a registered **ompt_callback_device_initialize** callback for each
20 occurrence of a *device-initialize* event in that thread. This callback has type signature
21 **ompt_callback_device_initialize_t**.

22 A thread dispatches a registered **ompt_callback_device_load** callback for each occurrence
23 of a *device-load* event in that thread. This callback has type signature
24 **ompt_callback_device_load_t**.

25 A thread dispatches a registered **ompt_callback_device_unload** callback for each
26 occurrence of a *device-unload* event in that thread. This callback has type signature
27 **ompt_callback_device_unload_t**.

28 A thread dispatches a registered **ompt_callback_device_finalize** callback for each
29 occurrence of a *device-finalize* event in that thread. This callback has type signature
30 **ompt_callback_device_finalize_t**.

Restrictions

No thread may offload execution of an OpenMP construct to a device until a dispatched **ompt_callback_device_initialize** callback completes.

No thread may offload execution of an OpenMP construct to a device after a dispatched **ompt_callback_device_finalize** callback occurs.

Cross References

- **ompt_callback_device_initialize_t**, see Section 4.1.4.2.28 on page 438.
- **ompt_callback_device_load_t**, see Section 4.1.4.2.19 on page 426.
- **ompt_callback_device_unload_t**, see Section 4.1.4.2.20 on page 428.
- **ompt_callback_device_finalize_t**, see Section 4.1.4.2.29 on page 440.

2.15.2 target data Construct

Summary

Map variables to a device data environment for the extent of the region.

Syntax

C / C++

The syntax of the **target data** construct is as follows:

```
#pragma omp target data clause[ [ [,] clause] ... ] new-line
    structured-block
```

where *clause* is one of the following:

if([**target data :**] *scalar-expression***)**

device(*integer-expression***)**

map([[*map-type-modifier*[,]] *map-type* **:**] *list***)**

use_device_ptr(*list***)**

C / C++

Fortran

The syntax of the **target data** construct is as follows:

```
!$omp target data clause[ [ [,] clause] ... ]
    structured-block
!$omp end target data
```

where *clause* is one of the following:

if([**target data :**] *scalar-logical-expression***)**

device(*scalar-integer-expression***)**

map([[*map-type-modifier*[,]] *map-type* **:**] *list***)**

use_device_ptr(*list***)**

The **end target data** directive denotes the end of the **target data** construct.

Fortran

Binding

The binding task set for a **target data** region is the generating task. The **target data** region binds to the region of the generating task.

Description

When a **target data** construct is encountered, the encountering task executes the region. If there is no **device** clause, the default device is determined by the *default-device-var* ICV. Variables are mapped for the extent of the region, according to any data-mapping attribute clauses, from the data environment of the encountering task to the device data environment. When an **if** clause is present and the **if** clause expression evaluates to *false*, the device is the host.

List items that appear in a **use_device_ptr** clause are converted into device pointers to the corresponding list items in the device data environment. If a **use_device_ptr** clause and one or more **map** clauses are present on the same construct, this conversion will occur as if performed after all variables are mapped according to those **map** clauses.

Events

The *target-data-begin* event occurs when a thread enters a **target data** region.

The *target-data-end* event occurs when a thread exits a **target data** region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_target** callback for each occurrence of a *target-data-begin* and *target-data-end* event in that thread in the context of the task encountering the construct. The callback has type signature **ompt_callback_target_t**. The callback receives **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate, and **ompt_target_enter_data** as its *kind* argument.

Restrictions

- A program must not depend on any ordering of the evaluations of the clauses of the **target data** directive, or on any side effects of the evaluations of the clauses.
- At most one **device** clause can appear on the directive. The **device** expression must evaluate to a non-negative integer value less than the value of **omp_get_num_devices()**.
- At most one **if** clause can appear on the directive.
- A *map-type* in a **map** clause must be **to**, **from**, **tofrom** or **alloc**.
- At least one **map** or **use_device_ptr** clause must appear on the directive.
- A list item in a **use_device_ptr** clause must have a corresponding list item in the device data environment.
- A list item that specifies a given variable may not appear in more than one **use_device_ptr** clause.
- References in the construct to a list item that appears in a **use_device_ptr** clause must be to the address of the list item.

Cross References

- *default-device-var*, see Section 2.4 on page 49.
- **if** Clause, see Section 2.17 on page 192.
- **map** clause, see Section 2.20.6.1 on page 280.
- **omp_get_num_devices** routine, see Section 3.2.35 on page 337.
- **ompt_callback_target_t**, see Section 4.1.4.2.18 on page 425.

2.15.3 target enter data Construct

Summary

The **target enter data** directive specifies that variables are mapped to a device data environment. The **target enter data** directive is a stand-alone directive.

Syntax

C / C++

The syntax of the **target enter data** construct is as follows:

```
#pragma omp target enter data [ clause[ [,] clause]...] new-line
```

where *clause* is one of the following:

```
if([ target enter data :] scalar-expression)
device(integer-expression)
map([ [map-type-modifier[,]] map-type : ] list)
depend(dependence-type : locator-list [:iterators-definition])
nowait
```

C / C++

Fortran

The syntax of the **target enter data** is as follows:

```
!$omp target enter data [ clause[ [,] clause]...]
```

where clause is one of the following:

```
if([ target enter data :] scalar-logical-expression)
device(scalar-integer-expression)
map([ [map-type-modifier[,]] map-type : ] list)
depend(dependence-type : locator-list [:iterators-definition])
nowait
```

Fortran

Binding

The binding task set for a **target enter data** region is the generating task, which is the *target task* generated by the **target enter data** construct. The **target enter data** region binds to the corresponding *target task* region.

Description

When a **target enter data** construct is encountered, the list items are mapped to the device data environment according to the **map** clause semantics.

The **target enter data** construct is a task generating construct. The generated task is a *target task*. The generated task region encloses the **target enter data** region.

All clauses are evaluated when the **target enter data** construct is encountered. The data environment of the *target task* is created according to the data-sharing attribute clauses on the **target enter data** construct, per-data environment ICVs, and any default data-sharing attribute rules that apply to the **target enter data** construct. A variable that is mapped in the **target enter data** construct has a default data-sharing attribute of shared in the data environment of the *target task*.

Assignment operations associated with mapping a variable (see Section 2.20.6.1 on page 280) occur when the *target task* executes.

If the **nowait** clause is present, execution of the *target task* may be deferred. If the **nowait** clause is not present, the *target task* is an included task.

If a **depend** clause is present, it is associated with the *target task*.

If there is no **device** clause, the default device is determined by the *default-device-var* ICV.

When an **if** clause is present and the **if** clause expression evaluates to *false*, the device is the host.

Events

Events associated with a *target task* are the same as for the **task** construct defined in Section 2.13.1 on page 110.

The *target-enter-data-begin* event occurs when a thread enters a **target enter data** region.

The *target-enter-data-end* event occurs when a thread exits a **target enter data** region.

Tool Callbacks

Callbacks associated with events for *target tasks* are the same as for the **task** construct defined in Section 2.13.1 on page 110.

A thread dispatches a registered **ompt_callback_target** callback for each occurrence of a *target-enter-data-begin* and *target-enter-data-end* event in that thread in the context of the target task on the host. The callback has type signature **ompt_callback_target_t**. The callback receives **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate, and **ompt_target_enter_data** as its *kind* argument.

1 ### Restrictions

2 • A program must not depend on any ordering of the evaluations of the clauses of the
3 **target enter data** directive, or on any side effects of the evaluations of the clauses.

4 • At least one **map** clause must appear on the directive.

5 • At most one **device** clause can appear on the directive. The **device** expression must evaluate
6 to a non-negative integer value less than the value of **omp_get_num_devices()**.

7 • At most one **if** clause can appear on the directive.

8 • A *map-type* must be specified in all **map** clauses and must be either **to** or **alloc**.

9 • At most one **nowait** clause can appear on the directive.

10 ### Cross References

11 • *default-device-var*, see Section 2.4.1 on page 49.

12 • **task**, see Section 2.13.1 on page 110.

13 • **task scheduling constraints**, see Section 2.13.6 on page 125.

14 • **target data**, see Section 2.15.2 on page 132.

15 • **target exit data**, see Section 2.15.4 on page 137.

16 • **if** Clause, see Section 2.17 on page 192.

17 • **map** clause, see Section 2.20.6.1 on page 280.

18 • **omp_get_num_devices** routine, see Section 3.2.35 on page 337

19 • **ompt_callback_target_t**, see Section 4.1.4.2.18 on page 425.

20 ## 2.15.4 target exit data Construct

21 ### Summary

22 The **target exit data** directive specifies that list items are unmapped from a device data
23 environment. The **target exit data** directive is a stand-alone directive.

Syntax

C / C++

The syntax of the **target exit data** construct is as follows:

```
#pragma omp target exit data [ clause[ [,] clause]...] new-line
```

where *clause* is one of the following:

if([**target exit data** **:**] *scalar-expression***)**

device(*integer-expression***)**

map([[*map-type-modifier*[,]] *map-type* **:**] *list***)**

depend(*dependence-type* **:** *locator-list* **[:***iterators-definition***])**

nowait

C / C++

Fortran

The syntax of the **target exit data** is as follows:

```
!$omp target exit data [ clause[ [,] clause]...]
```

where clause is one of the following:

if([**target exit data** **:**] *scalar-logical-expression***)**

device(*scalar-integer-expression***)**

map([[*map-type-modifier*[,]] *map-type* **:**] *list***)**

depend(*dependence-type* **:** *locator-list* **[:***iterators-definition***])**

nowait

Fortran

Binding

The binding task set for a **target exit data** region is the generating task, which is the *target task* generated by the **target exit data** construct. The **target exit data** region binds to the corresponding *target task* region.

Description

When a **target exit data** construct is encountered, the list items in the **map** clauses are unmapped from the device data environment according to the **map** clause semantics.

The **target exit data** construct is a task generating construct. The generated task is a *target task*. The generated task region encloses the **target exit data** region.

All clauses are evaluated when the **target exit data** construct is encountered. The data environment of the *target task* is created according to the data-sharing attribute clauses on the **target exit data** construct, per-data environment ICVs, and any default data-sharing attribute rules that apply to the **target exit data** construct. A variable that is mapped in the **target exit data** construct has a default data-sharing attribute of shared in the data environment of the *target task*.

Assignment operations associated with mapping a variable (see Section 2.20.6.1 on page 280) occur when the *target task* executes.

If the **nowait** clause is present, execution of the *target task* may be deferred. If the **nowait** clause is not present, the *target task* is an included task.

If a **depend** clause is present, it is associated with the *target task*.

If there is no **device** clause, the default device is determined by the *default-device-var* ICV.

When an **if** clause is present and the **if** clause expression evaluates to *false*, the device is the host.

Events

Events associated with a *target task* are the same as for the **task** construct defined in Section 2.13.1 on page 110.

The *target-exit-begin* event occurs when a thread enters a **target exit data** region.

The *target-exit-end* event occurs when a thread exits a **target exit data** region.

Tool Callbacks

Callbacks associated with events for *target tasks* are the same as for the **task** construct defined in Section 2.13.1 on page 110.

A thread dispatches a registered **ompt_callback_target** callback for each occurrence of a *target-exit-begin* and *target-exit-end* event in that thread in the context of the target task on the host. The callback has type signature **ompt_callback_target_t**. The callback receives **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate, and **ompt_target_exit_data** as its *kind* argument.

Restrictions

- A program must not depend on any ordering of the evaluations of the clauses of the **target exit data** directive, or on any side effects of the evaluations of the clauses.
- At least one **map** clause must appear on the directive.
- At most one **device** clause can appear on the directive. The **device** expression must evaluate to a non-negative integer value less than the value of **omp_get_num_devices()**.
- At most one **if** clause can appear on the directive.
- A *map-type* must be specified in all **map** clauses and must be either **from**, **release**, or **delete**.
- At most one **nowait** clause can appear on the directive.

Cross References

- *default-device-var*, see Section 2.4.1 on page 49.
- **task**, see Section 2.13.1 on page 110.
- **task scheduling constraints**, see Section 2.13.6 on page 125.
- **target data**, see Section 2.15.2 on page 132.
- **target enter data**, see Section 2.15.3 on page 134.
- **if** Clause, see Section 2.17 on page 192.
- **map** clause, see Section 2.20.6.1 on page 280.
- **omp_get_num_devices** routine, see Section 3.2.35 on page 337.
- **ompt_callback_target_t**, see Section 4.1.4.2.18 on page 425.

2.15.5 `target` Construct

Summary

Map variables to a device data environment and execute the construct on that device.

Syntax

C / C++

The syntax of the **`target`** construct is as follows:

```
#pragma omp target [clause[ [,] clause] ... ] new-line
    structured-block
```

where *clause* is one of the following:

`if(`[**`target`** **`:`**] *scalar-expression***`)`**

`device(`*integer-expression***`)`**

`private(`*list***`)`**

`firstprivate(`*list***`)`**

`in_reduction(`*reduction-identifier* **`:`** *list***`)`**

`reduction(`*reduction-identifier* **`:`** *list***`)`**

`map(`[[*map-type-modifier*[,]] *map-type* **`:`**] *list***`)`**

`is_device_ptr(`*list***`)`**

`defaultmap(tofrom:scalar | none)`

`nowait`

`depend(`*dependence-type* **`:`** *locator-list* **`[:`***iterators-definition***`])`**

`allocate(`*allocator:list***`)`**

C / C++

Fortran

The syntax of the **target** construct is as follows:

```
!$omp target [clause[ [,] clause] ... ]
      structured-block
!$omp end target
```

where *clause* is one of the following:

if([**target** **:**] *scalar-logical-expression***)**

device(*scalar-integer-expression***)**

private(*list***)**

firstprivate(*list***)**

in_reduction(*reduction-identifier* **:** *list***)**

reduction(*reduction-identifier* **:** *list***)**

map([[*map-type-modifier*[,]] *map-type* **:**] *list***)**

is_device_ptr(*list***)**

defaultmap(tofrom:scalar)

nowait

depend(*dependence-type* **:** *locator-list* **[:** *iterators-definition* **])**

allocate(*allocator:list***)**

The **end target** directive denotes the end of the **target** construct

Fortran

Binding

The binding task set for a **target** region is the generating task, which is the *target task* generated by the **target** construct. The **target** region binds to the corresponding *target task* region.

Description

The **target** construct provides a superset of the functionality provided by the **target data** directive, except for the **use_device_ptr** clause.

The functionality added to the **target** directive is the inclusion of an executable region to be executed by a device. That is, the **target** directive is an executable directive.

The **target** construct is a task generating construct. The generated task is a *target task*. The generated task region encloses the **target** region.

All clauses are evaluated when the **target** construct is encountered. The data environment of the *target task* is created according to the data-sharing attribute clauses on the **target** construct, per-data environment ICVs, and any default data-sharing attribute rules that apply to the **target** construct. If a variable or part of a variable appears as a list item in a **reduction** clause on the **target** construct and does not appear as a list item in an **in_reduction** clause on the construct, the variable has a default data-sharing attribute of shared in the data environment of the *target task*. Likewise, if a variable or part of a variable is mapped by the **target** construct, the variable has a default data-sharing attribute of shared in the data environment of the *target task*.

If an **in_reduction** clause is present on the **target** construct, the behavior is as if a **reduction** clause with the same reduction operator and list items is present on the construct.

Assignment operations associated with mapping a variable (see Section 2.20.6.1 on page 280) occur when the *target task* executes.

If the **nowait** clause is present, execution of the *target task* may be deferred. If the **nowait** clause is not present, the *target task* is an included task.

If a **depend** clause is present, it is associated with the *target task*.

When an **if** clause is present and the **if** clause expression evaluates to *false*, the **target** region is executed by the host device in the host data environment.

The **is_device_ptr** clause is used to indicate that a list item is a device pointer already in the device data environment and that it should be used directly. Support for device pointers created outside of OpenMP, specifically outside of the **omp_target_alloc** routine and the **use_device_ptr** clause, is implementation defined.

If a function (C, C++, Fortran) or subroutine (Fortran) is referenced in a **target** construct then that function or subroutine is treated as if its name had appeared in a **to** clause on a **declare target** directive.

C / C++

If an array section is a list item in a **map** clause and it has a named pointer that is a scalar variable with a predetermined data-sharing attribute of firstprivate (see Section 2.20.1.1 on page 240) then on entry to the **target** region:

- If the list item is not a zero-length array section, the corresponding private variable is initialized relative to the address of the storage location of the corresponding array section in the device data environment that is created by the **map** clause.
- If the list item is a zero-length array section, the corresponding private variable is initialized relative to the address of the corresponding storage location in the device data environment. If the corresponding storage location is not present in the device data environment, the corresponding private variable is initialized to NULL.

C / C++

Events

The *target-begin* event occurs when a thread enters a **target** region.

The *target-end* event occurs when a thread exits a **target** region.

The *target-submit* event occurs prior to creating an initial task on a target device for a target region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_target** callback for each occurrence of a *target-begin* and *target-end* event in that thread in the context of target task on the host. The callback has type signature **ompt_callback_target_t**. The callback receives **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate, and **ompt_target** as its *kind* argument.

A thread dispatches a registered **ompt_callback_target_submit** callback for each occurrence of a *target-submit* event in that thread. The callback has type signature **ompt_callback_target_submit_t**.

Restrictions

- If a **target**, **target update**, **target data**, **target enter data**, or **target exit data** construct is encountered during execution of a **target** region, the behavior is unspecified.
- The result of an **omp_set_default_device**, **omp_get_default_device**, or **omp_get_num_devices** routine called within a **target** region is unspecified.
- The effect of an access to a **threadprivate** variable in a target region is unspecified.
- If a list item in a **map** clause is a structure element, any other element of that structure that is referenced in the **target** construct must also appear as a list item in a **map** clause.
- A variable referenced in a **target** region but not the **target** construct that is not declared in the **target** region must appear in a **declare target** directive.

1 • At most one **defaultmap** clause can appear on the directive.

2 • At most one **nowait** clause can appear on the directive.

3 • A *map-type* in a **map** clause must be **to**, **from**, **tofrom** or **alloc**.

4 • A list item that appears in an **is_device_ptr** clause must be a valid device pointer in the
5 device data environment.

6 • At most one **device** clause can appear on the directive. The **device** expression must evaluate
7 to a non-negative integer value less than the value of **omp_get_num_devices()**.

C / C++

8 • An attached pointer may not be modified in a **target** region.

C / C++

C

9 • A list item that appears in an **is_device_ptr** clause must have a type of pointer or array.

C

C++

10 • A list item that appears in an **is_device_ptr** clause must have a type of pointer, array,
11 reference to pointer or reference to array.

12 • The effect of invoking a virtual member function of an object on a device other than the device
13 on which the object was constructed is implementation defined.

14 • A throw executed inside a **target** region must cause execution to resume within the same
15 **target** region, and the same thread that threw the exception must catch it.

C++

Fortran

16 • A list item that appears in an **is_device_ptr** clause must be a dummy argument that does
17 not have the **ALLOCATABLE**, **POINTER** or **VALUE** attribute.

18 • If a list item in a **map** clause is an array section, and the array section is derived from a variable
19 with a **POINTER** or **ALLOCATABLE** attribute then the behavior is unspecified if the
20 corresponding list item's variable is modified in the region.

Fortran

Cross References

- *default-device-var*, see Section 2.4 on page 49.
- **task** construct, see Section 2.13.1 on page 110.
- **task** scheduling constraints, see Section 2.13.6 on page 125
- **target data** construct, see Section 2.15.2 on page 132.
- **if** Clause, see Section 2.17 on page 192.
- **private** and **firstprivate** clauses, see Section 2.20.3 on page 249.
- Data-mapping Attribute Rules and Clauses, see Section 2.20.6 on page 279.
- **omp_get_num_devices** routine, see Section 3.2.35 on page 337
- **ompt_callback_target_t**, see Section 4.1.4.2.18 on page 425.
- **ompt_callback_target_submit_t**, Section 4.1.4.2.23 on page 432.

2.15.6 target update Construct

Summary

The **target update** directive makes the corresponding list items in the device data environment consistent with their original list items, according to the specified motion clauses. The **target update** construct is a stand-alone directive.

Syntax

C / C++

The syntax of the **target update** construct is as follows:

```
#pragma omp target update clause[ [ [,] clause] ... ] new-line
```

where *clause* is either *motion-clause* or one of the following:

if([**target update :**] *scalar-expression***)**

device(*integer-expression***)**

nowait

depend(*dependence-type* **:** *locator-list* **[:** *iterators-definition***])**

and *motion-clause* is one of the following:

to([**mapper(***mapper-identifier***):**]*list***)**

from([**mapper(***mapper-identifier***):**]*list***)**

C / C++

Fortran

The syntax of the **target update** construct is as follows:

```
!$omp target update clause[ [ [,] clause] ... ]
```

where *clause* is either *motion-clause* or one of the following:

if([**target update :**] *scalar-logical-expression***)**

device(*scalar-integer-expression***)**

nowait

depend(*dependence-type* **:** *locator-list*[**:***iterators-definition*]**)**

and *motion-clause* is one of the following:

to([**mapper(***mapper-identifier***):**]*list***)**

from([**mapper(***mapper-identifier***):**]*list***)**

Fortran

Binding

The binding task set for a **target update** region is the generating task, which is the *target task* generated by the **target update** construct. The **target update** region binds to the corresponding *target task* region.

Description

For each list item in a **to** or **from** clause there is a corresponding list item and an original list item. If the corresponding list item is not present in the device data environment then no assignment occurs to or from the original list item. Otherwise, each corresponding list item in the device data environment has an original list item in the current task's data environment. If a **mapper()** modifier appears in a **to** clause, each list item is replaced with the list items that the given mapper specifies are to be mapped with a **to** or **tofrom** map-type. If a **mapper()** modifier appears in a **from** clause, each list item is replaced with the list items that the given mapper specifies are to be mapped with a **from** or **tofrom** map-type.

For each list item in a **from** or a **to** clause:

- For each part of the list item that is an attached pointer:
  - On exit from the region that part of the original list item will have the value it had on entry to the region;
  - On exit from the region that part of the corresponding list item will have the value it had on entry to the region;
- For each part of the list item that is not an attached pointer:
  - If the clause is **from**, the value of that part of the corresponding list item is assigned to that part of the original list item;
  - If the clause is **to**, the value of that part of the original list item is assigned to that part of the corresponding list item.
- To avoid race conditions:
  - Concurrent reads or updates of any part of the original list item must be synchronized with the update of the original list item that occurs as a result of the **from** clause;
  - Concurrent reads or updates of any part of the corresponding list item must be synchronized with the update of the corresponding list item that occurs as a result of the **to** clause.

The list items that appear in the **to** or **from** clauses may include array sections.

The **target update** construct is a task generating construct. The generated task is a *target task*. The generated task region encloses the **target update** region.

All clauses are evaluated when the **target update** construct is encountered. The data environment of the *target task* is created according to the data-sharing attribute clauses on the **target update** construct, per-data environment ICVs, and any default data-sharing attribute rules that apply to the **target update** construct. A variable that is mapped in the **target update** construct has a default data-sharing attribute of shared in the data environment of the *target task*.

Assignment operations associated with mapping a variable (see Section 2.20.6.1 on page 280) occur when the *target task* executes.

If the **nowait** clause is present, execution of the *target task* may be deferred. If the **nowait** clause is not present, the *target task* is an included task.

If a **depend** clause is present, it is associated with the *target task*.

The device is specified in the **device** clause. If there is no **device** clause, the device is determined by the *default-device-var* ICV. When an **if** clause is present and the **if** clause expression evaluates to *false* then no assignments occur.

Events

Events associated with a *target task* are the same as for the **task** construct defined in Section 2.13.1 on page 110.

The *target-update-begin* event occurs when a thread enters a **target update** region.

The *target-update-end* event occurs when a thread exits a **target update** region.

Tool Callbacks

Callbacks associated with events for *target tasks* are the same as for the **task** construct defined in Section 2.13.1 on page 110.

A thread dispatches a registered **ompt_callback_target** callback for each occurrence of a *target-update-begin* and *target-update-end* event in that thread in the context of the target task on the host. The callback has type signature **ompt_callback_target_t**. The callback receives **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate, and **ompt_target_update** as its *kind* argument.

Restrictions

- A program must not depend on any ordering of the evaluations of the clauses of the **target update** directive, or on any side effects of the evaluations of the clauses.
- At least one *motion-clause* must be specified.
- If a list item is an array section it must specify contiguous storage.
- A list item can only appear in a **to** or **from** clause, but not both.
- A list item in a **to** or **from** clause must have a mappable type.
- At most one **device** clause can appear on the directive. The **device** expression must evaluate to a non-negative integer value less than the value of **omp_get_num_devices()**.
- At most one **if** clause can appear on the directive.
- At most one **nowait** clause can appear on the directive.

Cross References

- *default-device-var*, see Section 2.4 on page 49.
- Array sections, Section 2.5 on page 60
- **task** construct, see Section 2.13.1 on page 110.
- **task** scheduling constraints, see Section 2.13.6 on page 125
- **target data**, see Section 2.15.2 on page 132.
- **if** Clause, see Section 2.17 on page 192.
- **omp_get_num_devices** routine, see Section 3.2.35 on page 337
- **ompt_callback_task_create_t**, see Section 4.1.4.2.7 on page 412.
- **ompt_callback_target_t**, see Section 4.1.4.2.18 on page 425.

2.15.7 declare target Directive

Summary

The **declare target** directive specifies that variables, functions (C, C++ and Fortran), and subroutines (Fortran) are mapped to a device. The **declare target** directive is a declarative directive.

Syntax

C / C++

The syntax of the **declare target** directive takes either of the following forms:

```
#pragma omp declare target new-line
declaration-definition-seq
#pragma omp end declare target new-line
```

or

```
#pragma omp declare target (extended-list) new-line
```

or

```
#pragma omp declare target clause[ [ , ] clause ... ] new-line
```

1 where *clause* is one of the following:

2 **to**(*extended-list*)

3 **link**(*list*)

4 **implements**(*function-name*)

5 **device_type(host | nohost | any)**

C / C++

Fortran

6 The syntax of the **declare target** directive is as follows:

```
!$omp declare target (extended-list)
```

7 or

```
!$omp declare target [clause[ [ , ] clause] ... ]
```

8 where *clause* is one of the following:

9 **to**(*extended-list*)

10 **link**(*list*)

11 **implements**(*subroutine-name*)

12 **device_type(host | nohost | any)**

Fortran

Description

The **declare target** directive ensures that procedures and global variables can be executed or accessed on a device. Variables are mapped for all device executions, or for specific device executions through a **link** clause.

If an *extended-list* is present with no clause then the **to** clause is assumed.

The **implements** clause specifies that an alternate version of a procedure should be used.

The **device_type** clause specifies if a version of the procedure should be made available on host, device or both. If **host** is specified only host version of the procedure is made available. If **nohost** is specified then only device version of the procedure is made available. If **any** is specified then both device and host version of the procedure is made available.

C / C++

If a function is treated as if it appeared as a list item in a **to** clause on a **declare target** directive in the same translation unit in which the definition of the function occurs then a device-specific version of the function is created.

If a variable is treated as if it appeared as a list item in a **to** clause on a **declare target** directive in the same translation unit in which the definition of the variable occurs then the original list item is allocated a corresponding list item in the device data environment of all devices.

All calls in **target** constructs to the function in the **implements** clause are replaced by the function following the **declare target** constructs.

C / C++

Fortran

If a procedure that is host associated is treated as if it appeared as a list item in a **to** clause on a **declare target** directive then a device-specific version of the procedure is created.

If a variable that is host associated is treated as if it appeared as a list item in a **to** clause on a **declare target** directive then the original list item is allocated a corresponding list item in the device data environment of all devices.

1 All calls in **target** constructs to the procedure in the **implements** clause are replaced by the
2 procedure in which **declare target** construct appeared.

Fortran

3 If a variable is treated as if it appeared as a list item in a **to** clause on a **declare target**
4 directive then the corresponding list item in the device data environment of each device is
5 initialized once, in the manner specified by the program, but at an unspecified point in the program
6 prior to the first reference to that list item. The list item is never removed from those device data
7 environments as if its reference count is initialized to positive infinity.

8 The list items of a **link** clause are not mapped by the **declare target** directive. Instead, their
9 mapping is deferred until they are mapped by **target data** or **target** constructs. They are
10 mapped only for such regions.

C / C++

11 If a function is referenced in a function that is treated as if it appeared as a list item in a **to** clause
12 on a **declare target** directive then the name of the referenced function is treated as if it had
13 appeared in a **to** clause on a **declare target** directive.

14 If a variable with static storage duration or a function (except *lambda* for C++) is referenced in the
15 initializer expression list of a variable with static storage duration that is treated as if it appeared as a
16 list item in a **to** clause on a **declare target** directive then the name of the referenced variable
17 or function is treated as if it had appeared in a **to** clause on a **declare target** directive.

18 The form of the **declare target** directive that has no clauses and requires a matching
19 **end declare target** directive defines an implicit *extended-list* to an implicit **to** clause. The
20 implicit *extended-list* consists of the variable names of any variable declarations at file or
21 namespace scope that appear between the two directives and of the function names of any function
22 declarations at file, namespace or class scope that appear between the two directives.

23 The *declaration-definition-seq* defined by a **declare target** directive and an
24 **end declare target** directive may contain **declare target** directives without the
25 **device_type** clause. If a list item appears both in an implicit and explicit list, the explicit list
26 takes precedence.

C / C++

Fortran

1 If a procedure is referenced in a procedure that is treated as if it appeared as a list item in a **to**
2 clause on a **declare target** directive then the name of the procedure is treated as if it had
3 appeared in a **to** clause on a **declare target** directive.

4 If a **declare target** does not have any clauses then an implicit *extended-list* to an implicit **to**
5 clause of one item is formed from the name of the enclosing subroutine subprogram, function
6 subprogram or interface body to which it applies.

7 If a **declare target** directive has an **implements** or **device_type** clause then any
8 enclosed internal procedures cannot contain any **declare target** directives. The enclosing
9 **device_type** clause implicitly applies to internal procedures.

Fortran

10 ## Restrictions

11 • A threadprivate variable cannot appear in a **declare target** directive.

12 • A variable declared in a **declare target** directive must have a mappable type.

13 • The same list item must not appear multiple times in clauses on the same directive.

14 • The same list item must not explicitly appear in both a **to** clause on one **declare target**
15 directive and a **link** clause on another **declare target** directive.

16 • The **implements** clause and **device_type** clause are the only valid combination that can
17 appear if either one of them is present.

C++

18 • The function names of overloaded functions or template functions may only be specified within
19 an implicit *extended-list*.

20 • If a *lambda declaration and definition* appears between a **declare target** directive and the
21 matching **end declare target** directive, all the variables that are captured by the *lambda*
22 expression must also be variables that are treated as if they appear in a **to** clause.

C++

Fortran

23 • If a list item is a procedure name, it must not be a generic name, procedure pointer or entry name.

24 • Any **declare target** directive with clauses must appear in a specification part of a
25 subroutine subprogram, function subprogram, program or module.

26 • Any **declare target** directive without clauses must appear in a specification part of a
27 subroutine subprogram, function subprogram or interface body to which it applies.

- If a **declare target** directive is specified in an interface block for a procedure, it must match a **declare target** directive in the definition of the procedure.
- If an external procedure is a type-bound procedure of a derived type and a **declare target** directive is specified in the definition of the external procedure, such a directive must appear in the interface block that is accessible to the derived type definition.
- If any procedure is declared via a procedure declaration statement that is not in the type-bound procedure part of a derived-type definition, any **declare target** with the procedure name must appear in the same specification part.
- A variable that is part of another variable (as an array or structure element) cannot appear in a **declare target** directive.
- The **declare target** directive must appear in the declaration section of a scoping unit in which the common block or variable is declared. Although variables in common blocks can be accessed by use association or host association, common block **declare target** directive must be declared to be a common block in the same scoping unit in which the **declare target** directive appears.
- If a **declare target** directive specifying a common block name appears in one program unit, then such a directive must also appear in every other program unit that contains a **COMMON** statement specifying the same name. It must appear after the last such **COMMON** statement in the program unit.
- If a list item is declared with the **BIND** attribute, the corresponding C entities must also be specified in a **declare target** directive in the C program.
- A blank common block cannot appear in a **declare target** directive.
- A variable can only appear in a **declare target** directive in the scope in which it is declared. It must not be an element of a common block or appear in an **EQUIVALENCE** statement.
- A variable that appears in a **declare target** directive must be declared in the Fortran scope of a module or have the **SAVE** attribute, either explicitly or implicitly.

Fortran

2.15.8 declare mapper Directive

Summary

The **declare mapper** directive declares a user-defined mapper for a given type, and may define a *mapper-identifier* that can be used in a **map** clause. The **declare mapper** directive is a declarative directive.

Syntax

C / C++

The syntax of the **declare mapper** directive is as follows:

```
#pragma omp declare mapper([mapper-identifier:]type var)
[clause[ [,] clause] ... ] new-line
```

C / C++

Fortran

The syntax of the **declare mapper** directive is as follows:

```
!$omp declare mapper([mapper-identifier:] type :: var)
[clause[ [,] clause] ... ]
```

Fortran

where:

- *mapper-identifier* is a base-language identifier
- *type* is a valid type in scope
- *var* is a valid base-language identifier
- *clause* is one of the following:
  - **map(**[[*map-type-modifier*[,]] *map-type* **:**] *list***)**
  - **use_by_default**
- *map-type* is one of the following:
  - **alloc**
  - **to**
  - **from**
  - **tofrom**
- and *map-type-modifier* is **always**

1 ### Description

2 User-defined mappers can be defined using the **declare mapper** directive. The type and, if
3 specified, the *mapper-identifier*, uniquely identify the mapper for use in a **map** clause later in the
4 program. The visibility and accessibility of this declaration are the same as those of a variable
5 declared at the same point in the program.

6 The **use_by_default** clause makes this mapper the default for all variables of *type*, so that all
7 map clauses with this construct in scope that map a list item of *type* will use this mapper unless
8 another is explicitly specified.

9 The variable declared by *var* is available for use in all **map** clauses on the directive, and no part of
10 the variable to be mapped is mapped by default.

11 The default mapper for all types *T*, designated by the pre-defined mapper-identifier **default**, is as
12 follows unless a user-defined mapper is specified for that type.

```
declare mapper(T v) map(tofrom: v) use_by_default
```

13 All **map** clauses on the dirctive are expanded into corresponding **map** clauses wherever this mapper
14 is invoked, either by matching type or by being explicitly named in a **map** clause. A **map** clause
15 with list item *var* maps var as though no mapper were specified.

C++

16 The **declare mapper** directive can also appear at points in the program at which a static data
17 member could be declared. In this case, the visibility and accessibility of the declaration are the
18 same as those of a static data member declared at the same point in the program.

C++

19 ### Restrictions

20 • If the **use_by_default** clause is not specified, then a mapper-identifier is required.

21 • No instance of the mapper type can be mapped as part of the mapper, either directly or indirectly
22 through another type, except the instance passed as the list item.

23 ## 2.15.9 teams Construct

24 ### Summary

25 The **teams** construct creates a league of initial teams and the initial thread in each team executes
26 the region.

Syntax

C / C++

The syntax of the **teams** construct is as follows:

```
#pragma omp teams [clause[ [,] clause] ... ] new-line
    structured-block
```

where *clause* is one of the following:

num_teams(*integer-expression***)**

thread_limit(*integer-expression***)**

default(shared | none)

private(*list***)**

firstprivate(*list***)**

shared(*list***)**

reduction(*reduction-identifier* **:** *list***)**

allocate(*[allocator:]list***)**

C / C++

Fortran

The syntax of the **teams** construct is as follows:

```
!$omp teams [clause[ [,] clause] ... ]
    structured-block
!$omp end teams
```

where *clause* is one of the following:

```
num_teams(scalar-integer-expression)
thread_limit(scalar-integer-expression)
default(shared | firstprivate | private | none)
private(list)
firstprivate(list)
shared(list)
reduction(reduction-identifier : list)
allocate([allocator: ]list)
```

The **end teams** directive denotes the end of the **teams** construct.

Fortran

Binding

The binding thread set for a **teams** region is the encountering thread, which is the initial thread of the **target** region.

Description

When a thread encounters a **teams** construct, a league of teams is created. Each team is an initial team, and the initial thread in each team executes the **teams** region.

The number of teams created is implementation defined, but is less than or equal to the value specified in the **num_teams** clause. A thread may obtain the number of initial teams created by the construct by a call to the **omp_get_num_teams** routine.

The maximum number of threads participating in the contention group that each team initiates is implementation defined, but is less than or equal to the value specified in the **thread_limit** clause.

On a combined or composite construct that includes **target** and **teams** constructs, the expressions in **num_teams** and **thread_limit** clauses are evaluated on the host device on entry to the **target** construct.

Once the teams are created, the number of initial teams remains constant for the duration of the **teams** region.

Within a **teams** region, initial team numbers uniquely identify each initial team. Initial team numbers are consecutive whole numbers ranging from zero to one less than the number of initial teams. A thread may obtain its own initial team number by a call to the **omp_get_team_num** library routine.

After the teams have completed execution of the **teams** region, the encountering thread resumes execution of the enclosing **target** region.

There is no implicit barrier at the end of a **teams** construct.

Restrictions

Restrictions to the **teams** construct are as follows:

- A program that branches into or out of a **teams** region is non-conforming.
- A program must not depend on any ordering of the evaluations of the clauses of the **teams** directive, or on any side effects of the evaluation of the clauses.
- At most one **thread_limit** clause can appear on the directive. The **thread_limit** expression must evaluate to a positive integer value.
- At most one **num_teams** clause can appear on the directive. The **num_teams** expression must evaluate to a positive integer value.
- If specified, a **teams** construct must be contained within a **target** construct. That **target** construct must contain no statements, declarations or directives outside of the **teams** construct.
- **distribute**, **distribute simd**, distribute parallel loop, distribute parallel loop SIMD, and **parallel** regions, including any **parallel** regions arising from combined constructs, are the only OpenMP regions that may be strictly nested inside the **teams** region.

Cross References

- Data-sharing attribute clauses, see Section 2.20.3 on page 249.
- **omp_get_num_teams** routine, see Section 3.2.37 on page 339.
- **omp_get_team_num** routine, see Section 3.2.38 on page 340.

2.15.10 distribute Construct

Summary

The **distribute** construct specifies that the iterations of one or more loops will be executed by the initial teams in the context of their implicit tasks. The iterations are distributed across the initial threads of all initial teams that execute the **teams** region to which the **distribute** region binds.

Syntax

C / C++

The syntax of the **distribute** construct is as follows:

```
#pragma omp distribute [clause[ [,] clause] ... ] new-line
    for-loops
```

Where *clause* is one of the following:

private(*list***)**

firstprivate(*list***)**

lastprivate(*list***)**

collapse(*n***)**

dist_schedule(*kind*[**,** *chunk_size*]**)**

allocate([*allocator:*]*list***)**

All associated *for-loops* must have the canonical form described in Section 2.9 on page 75.

C / C++

Fortran

The syntax of the **distribute** construct is as follows:

```
!$omp distribute [clause[ [,] clause] ... ]
    do-loops
[!$omp end distribute]
```

Where *clause* is one of the following:

private(*list***)**

firstprivate(*list***)**

lastprivate(*list***)**

collapse(*n***)**

dist_schedule(*kind*[**,** *chunk_size*]**)**

allocate([*allocator:*]*list***)**

If an **end distribute** directive is not specified, an **end distribute** directive is assumed at the end of the *do-loops*.

Any associated *do-loop* must be a *do-construct* or an *inner-shared-do-construct* as defined by the Fortran standard. If an **end distribute** directive follows a *do-construct* in which several loop statements share a **DO** termination statement, then the directive can only be specified for the outermost of these **DO** statements.

Fortran

Binding

The binding thread set for a **distribute** region is the set of initial threads executing an enclosing **teams** region. A **distribute** region binds to this **teams** region.

Description

The **distribute** construct is associated with a loop nest consisting of one or more loops that follow the directive.

There is no implicit barrier at the end of a **distribute** construct. To avoid data races the original list items modified due to **lastprivate** or **linear** clauses should not be accessed between the end of the **distribute** construct and the end of the **teams** region to which the **distribute** binds.

The **collapse** clause may be used to specify how many loops are associated with the **distribute** construct. The parameter of the **collapse** clause must be a constant positive integer expression. If no **collapse** clause is present or its paraemter is 1, the only loop that is associated with the **distribute** construct is the one that immediately follows the **distribute** construct. If a **collapse** clause is specified with a parameter value greater than 1 and more than one loop is associated with the **distribute** construct, then the iteration of all associated loops are collapsed into one larger iteration space. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

A distribute loop has logical iterations numbered 0,1,...,N-1 where N is the number of loop iterations, and the logical numbering denotes the sequence in which the iterations would be executed if the set of associated loop(s) were executed sequentially. At the beginning of each logical iteration, the loop iteration variable of each associated loop has the value that it would have if the set of the associated loop(s) were executed sequentially.

If more than one loop is associated with the **distribute** construct then the number of times that any intervening code between any two associated loops will be executed is unspecified but will be at least once per iteration of the loop enclosing the intervening code and at most once per iteration of the innermost loop associated with the construct.

The iteration count for each associated loop is computed before entry to the outermost loop. If execution of any associated loop changes any of the values used to compute any of the iteration counts, then the behavior is unspecified.

The integer type (or kind, for Fortran) used to compute the iteration count for the collapsed loop is implementation defined.

If **dist_schedule** is specified, *kind* must be **static**. If specified, iterations are divided into chunks of size *chunk_size*, chunks are assigned to the initial teams of the league in a round-robin fashion in the order of the initial team number. When no *chunk_size* is specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each initial team of the league. The size of the chunks is unspecified in this case.

When no **dist_schedule** clause is specified, the schedule is implementation defined.

Events

The *distribute-begin* event occurs after an implicit task encounters a **distribute** construct but before the task starts the execution of the structured block of the **distribute** region.

The *distribute-end* event occurs after a **distribute** region finishes execution but before resuming execution of the encountering task.

Tool Callbacks

A thread dispatches a registered **ompt_callback_work** callback for each occurrence of a *distribute-begin* and *distribute-end* event in that thread. The callback occurs in the context of the implicit task. The callback has type signature **ompt_callback_work_t**. The callback receives **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate, and **ompt_work_distribute** as its *wstype* argument.

Restrictions

Restrictions to the **distribute** construct are as follows:

- The **distribute** construct inherits the restrictions of the loop construct.
- The region associated with the **distribute** construct must be strictly nested inside a **teams** region.
- A list item may appear in a **firstprivate** or **lastprivate** clause but not both.

Cross References

- loop construct, see Section 2.10.1 on page 78.
- **teams** construct, see Section 2.15.9 on page 157
- **ompt_work_distribute**, see Section 4.1.3.4.13 on page 398.
- **ompt_callback_work_t**, see Section 4.1.4.2.16 on page 423.

2.15.11 distribute simd Construct

Summary

The **distribute simd** construct specifies a loop that will be distributed across the master threads of the **teams** region and executed concurrently using SIMD instructions. The **distribute simd** construct is a composite construct.

Syntax

The syntax of the **distribute simd** construct is as follows:

C / C++

```
#pragma omp distribute simd [clause[ [,] clause] ... ] newline
    for-loops
```

where *clause* can be any of the clauses accepted by the **distribute** or **simd** directives with identical meanings and restrictions.

C / C++

Fortran

```
!$omp distribute simd [clause[ [,] clause] ... ]
    do-loops
[!$omp end distribute simd]
```

where *clause* can be any of the clauses accepted by the **distribute** or **simd** directives with identical meanings and restrictions.

If an **end distribute simd** directive is not specified, an **end distribute simd** directive is assumed at the end of the *do-loops*.

Fortran

Description

The **`distribute simd`** construct will first distribute the iterations of the associated loop(s) according to the semantics of the **`distribute`** construct and any clauses that apply to the distribute construct. The resulting chunks of iterations will then be converted to a SIMD loop in a manner consistent with any clauses that apply to the **`simd`** construct. The effect of any clause that applies to both constructs is as if it were applied to both constructs separately except the **`collapse`** clause, which is applied once.

Events

This composite construct generates the same events as the **`distribute`** construct.

Tool Callbacks

This composite construct dispatches the same callbacks as the **`distribute`** construct.

Restrictions

- The restrictions for the **`distribute`** and **`simd`** constructs apply.
- A list item may not appear in a **`linear`** clause, unless it is the loop iteration variable.
- The **`conditional`** modifier may not appear in a **`lastprivate`** clause.

Cross References

- **`simd`** construct, see Section 2.11.1 on page 96.
- **`distribute`** construct, see Section 2.15.10 on page 160.
- Data attribute clauses, see Section 2.20.3 on page 249.
- Events and tool callbacks for the **`distribute`** construct, see Section 2.11.1 on page 96.

2.15.12 Distribute Parallel Loop Construct

Summary

The distribute parallel loop construct specifies a loop that can be executed in parallel by multiple threads that are members of multiple teams. The distribute parallel loop construct is a composite construct.

Syntax

The syntax of the distribute parallel loop construct is as follows:

C / C++

```
#pragma omp distribute parallel for [clause[ [,] clause] ... ] newline
    for-loops
```

where *clause* can be any of the clauses accepted by the **distribute** or parallel loop directives with identical meanings and restrictions.

C / C++

Fortran

```
!$omp distribute parallel do [clause[ [,] clause] ... ]
    do-loops
[!$omp end distribute parallel do]
```

where *clause* can be any of the clauses accepted by the **distribute** or parallel loop directives with identical meanings and restrictions.

If an **end distribute parallel do** directive is not specified, an **end distribute parallel do** directive is assumed at the end of the *do-loops*.

Fortran

Description

The distribute parallel loop construct will first distribute the iterations of the associated loop(s) into chunks according to the semantics of the **distribute** construct and any clauses that apply to the **distribute** construct. Each of these chunks will form a loop. Each resulting loop will then be distributed across the threads within the teams region to which the **distribute** construct binds in a manner consistent with any clauses that apply to the parallel loop construct. The effect of any clause that applies to both constructs is as if it were applied to both constructs separately except the **collapse** clause, which is applied once.

Events

This composite construct generates the same events as the **distribute** and parallel loop constructs.

Tool Callbacks

This composite construct dispatches the same callbacks as the **distribute** and parallel loop constructs.

1 ### Restrictions

2 • The restrictions for the **distribute** and parallel loop constructs apply.

3 • No **ordered** clause can be specified.

4 • No **linear** clause can be specified.

5 • The **conditional** modifier may not appear in a **lastprivate** clause.

6 ### Cross References

7 • **distribute** construct, see Section 2.15.10 on page 160.

8 • Parallel loop construct, see Section 2.16.1 on page 169.

9 • Data attribute clauses, see Section 2.20.3 on page 249.

10 • Events and tool callbacks for **distribute** construct, see Section 2.15.10 on page 160.

11 • Events and tool callbacks for parallel loop construct, see Section 2.16.1 on page 169.

12 ## 2.15.13 Distribute Parallel Loop SIMD Construct

13 ### Summary

14 The distribute parallel loop SIMD construct specifies a loop that can be executed concurrently
15 using SIMD instructions in parallel by multiple threads that are members of multiple teams. The
16 distribute parallel loop SIMD construct is a composite construct.

17 ### Syntax

C / C++

18 The syntax of the distribute parallel loop SIMD construct is as follows:

```
#pragma omp distribute parallel for simd [clause[ [,] clause] ... ] newline
    for-loops
```

19 where *clause* can be any of the clauses accepted by the **distribute** or parallel loop SIMD
20 directives with identical meanings and restrictions

C / C++

Fortran

21 The syntax of the distribute parallel loop SIMD construct is as follows:

```
!$omp distribute parallel do simd [clause[ [,] clause] ... ]
    do-loops
[!$omp end distribute parallel do simd]
```

where *clause* can be any of the clauses accepted by the **distribute** or parallel loop SIMD directives with identical meanings and restrictions.

If an **end distribute parallel do simd** directive is not specified, an **end distribute parallel do simd** directive is assumed at the end of the *do-loops*.

Fortran

Description

The distribute parallel loop SIMD construct will first distribute the iterations of the associated loop(s) according to the semantics of the **distribute** construct and any clauses that apply to the **distribute** construct. The resulting loops will then be distributed across the threads contained within the **teams** region to which the **distribute** construct binds in a manner consistent with any clauses that apply to the parallel loop construct. The resulting chunks of iterations will then be converted to a SIMD loop in a manner consistent with any clauses that apply to the **simd** construct. The effect of any clause that applies to both constructs is as if it were applied to both constructs separately except the **collapse** clause, which is applied once.

Events

This composite construct generates the same events as the **distribute** and parallel loop constructs.

Tool Callbacks

This composite construct dispatches the same callbacks as the **distribute** and parallel loop constructs.

Restrictions

- The restrictions for the **distribute** and parallel loop SIMD constructs apply.
- No **ordered** clause can be specified.
- A list item may not appear in a **linear** clause, unless it is the loop iteration variable.
- The **conditional** modifier may not appear in a **lastprivate** clause.

Cross References

- **`distribute`** construct, see Section 2.15.10 on page 160.
- Parallel loop SIMD construct, see Section 2.16.4 on page 173.
- Data attribute clauses, see Section 2.20.3 on page 249.
- Events and tool callbacks for **`distribute`** construct, see Section 2.15.10 on page 160.
- Events and tool callbacks for parallel loop construct, see Section 2.16.1 on page 169.

2.16 Combined Constructs

Combined constructs are shortcuts for specifying one construct immediately nested inside another construct. The semantics of the combined constructs are identical to that of explicitly specifying the first construct containing one instance of the second construct and no other statements.

Some combined constructs have clauses that are permitted on both constructs that were combined. Where specified, the effect is as if applying the clauses to one or both constructs. If not specified and applying the clause to one construct would result in different program behavior than applying the clause to the other construct then the program's behavior is unspecified.

For combined constructs, tool callbacks shall be invoked as if the constructs were explicitly nested.

2.16.1 Parallel Loop Construct

Summary

The parallel loop construct is a shortcut for specifying a **`parallel`** construct containing one loop constuct with one or more associated loops and no other statements.

Syntax

C / C++

The syntax of the parallel loop construct is as follows:

```
#pragma omp parallel for [clause[ [,] clause] ... ] new-line
    for-loops
```

where *clause* can be any of the clauses accepted by the **parallel** or **for** directives, except the **nowait** clause, with identical meanings and restrictions.

C / C++

Fortran

The syntax of the parallel loop construct is as follows:

```
!$omp parallel do [clause[ [,] clause] ... ]
    do-loops
[!$omp end parallel do]
```

where *clause* can be any of the clauses accepted by the **parallel** or **do** directives, with identical meanings and restrictions.

If an **end parallel do** directive is not specified, an **end parallel do** directive is assumed at the end of the *do-loops*. **nowait** may not be specified on an **end parallel do** directive.

Fortran

Description

The semantics are identical to explicitly specifying a **parallel** directive immediately followed by a loop directive.

Restrictions

- The restrictions for the **parallel** construct and the loop construct apply.

Cross References

- **parallel** construct, see Section 2.8 on page 66.
- loop SIMD construct, see Section 2.11.3 on page 105.
- Data attribute clauses, see Section 2.20.3 on page 249.

2.16.2 parallel sections Construct

Summary

The **parallel sections** construct is a shortcut for specifying a **parallel** construct containing one **sections** construct and no other statements.

Syntax

C / C++

The syntax of the **parallel sections** construct is as follows:

```
#pragma omp parallel sections [clause[ [,] clause] ... ] new-line
    {
    [#pragma omp section new-line]
        structured-block
    [#pragma omp section new-line
        structured-block]
    ...
    }
```

where *clause* can be any of the clauses accepted by the **parallel** or **sections** directives, except the **nowait** clause, with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **parallel sections** construct is as follows:

```
!$omp parallel sections [clause[ [,] clause] ... ]
    [!$omp section]
        structured-block
    [!$omp section
        structured-block]
    ...
!$omp end parallel sections
```

where *clause* can be any of the clauses accepted by the **parallel** or **sections** directives, with identical meanings and restrictions.

The last section ends at the **end parallel sections** directive. **nowait** cannot be specified on an **end parallel sections** directive.

Fortran

1 **Description**

C / C++

2 The semantics are identical to explicitly specifying a **parallel** directive immediately followed
3 by a **sections** directive.

C / C++

Fortran

4 The semantics are identical to explicitly specifying a **parallel** directive immediately followed
5 by a **sections** directive, and an **end sections** directive immediately followed by an
6 **end parallel** directive.

Fortran

7 **Restrictions**

8 The restrictions for the **parallel** construct and the **sections** construct apply.

9 **Cross References**

10 • **parallel** construct, see Section 2.8 on page 66.

11 • **sections** construct, see Section 2.10.2 on page 86.

12 • Data attribute clauses, see Section 2.20.3 on page 249.

Fortran

13 ## 2.16.3 `parallel workshare` Construct

14 **Summary**

15 The **parallel workshare** construct is a shortcut for specifying a **parallel** construct
16 containing one **workshare** construct and no other statements.

17 **Syntax**

18 The syntax of the **parallel workshare** construct is as follows:

```
!$omp parallel workshare [clause[ [ , ] clause] ... ]
    structured-block
!$omp end parallel workshare
```

where *clause* can be any of the clauses accepted by the **parallel** directive, with identical meanings and restrictions. **nowait** may not be specified on an **end parallel workshare** directive.

Description

The semantics are identical to explicitly specifying a **parallel** directive immediately followed by a **workshare** directive, and an **end workshare** directive immediately followed by an **end parallel** directive.

Restrictions

The restrictions for the **parallel** construct and the **workshare** construct apply.

Cross References

- **parallel** construct, see Section 2.8 on page 66.
- **workshare** construct, see Section 2.10.4 on page 93.
- Data attribute clauses, see Section 2.20.3 on page 249.

Fortran

2.16.4 Parallel Loop SIMD Construct

Summary

The parallel loop SIMD construct is a shortcut for specifying a **parallel** construct containing one loop SIMD construct and no other statement.

Syntax

C / C++

The syntax of the parallel loop SIMD construct is as follows:

```
#pragma omp parallel for simd [clause[ [ , ] clause] ... ] new-line
    for-loops
```

where *clause* can be any of the clauses accepted by the **parallel** or **for simd** directives, except the **nowait** clause, with identical meanings and restrictions.

C / C++

Fortran

The syntax of the parallel loop SIMD construct is as follows:

```
!$omp parallel do simd [clause[ [ , ] clause] ... ]
    do-loops
[!$omp end parallel do simd]
```

where *clause* can be any of the clauses accepted by the **parallel** or **do simd** directives, with identical meanings and restrictions.

If an **end parallel do simd** directive is not specified, an **end parallel do simd** directive is assumed at the end of the *do-loops*. **nowait** may not be specified on an **end parallel do simd** directive.

Fortran

Description

The semantics of the parallel loop SIMD construct are identical to explicitly specifying a **parallel** directive immediately followed by a loop SIMD directive. The effect of any clause that applies to both constructs is as if it were applied to the loop SIMD construct and not to the **parallel** construct.

Restrictions

The restrictions for the **parallel** construct and the loop SIMD construct apply.

Cross References

- **parallel** construct, see Section 2.8 on page 66.
- loop SIMD construct, see Section 2.11.3 on page 105.
- Data attribute clauses, see Section 2.20.3 on page 249.

2.16.5 `target parallel` Construct

Summary

The **target parallel** construct is a shortcut for specifying a **target** construct containing a **parallel** construct and no other statements.

Syntax

C / C++

The syntax of the **target parallel** construct is as follows:

```
#pragma omp target parallel [clause[ [ , ] clause] ... ] new-line
    structured-block
```

where *clause* can be any of the clauses accepted by the **target** or **parallel** directives, except for **copyin**, with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **target parallel** construct is as follows:

```
!$omp target parallel [clause[ [ , ] clause] ... ]
    structured-block
!$omp end target parallel
```

where *clause* can be any of the clauses accepted by the **target** or **parallel** directives, except for **copyin**, with identical meanings and restrictions.

Fortran

Description

The semantics are identical to explicitly specifying a **target** directive immediately followed by a **parallel** directive.

Restrictions

The restrictions for the **target** and **parallel** constructs apply except for the following explicit modifications:

- If any **if** clause on the directive includes a *directive-name-modifier* then all **if** clauses on the directive must include a *directive-name-modifier*.
- At most one **if** clause without a *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **parallel** *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **target** *directive-name-modifier* can appear on the directive.
- If an **allocator** clause specifies an *allocator* it can only be a predefined allocator variable.

Cross References

- **parallel** construct, see Section 2.8 on page 66.
- **target** construct, see Section 2.15.5 on page 141.
- **if** Clause, see Section 2.17 on page 192.
- Data attribute clauses, see Section 2.20.3 on page 249.

2.16.6 Target Parallel Loop Construct

Summary

The target parallel loop construct is a shortcut for specifying a **target** construct containing a parallel loop construct and no other statements.

Syntax

C / C++

The syntax of the target parallel loop construct is as follows:

```
#pragma omp target parallel for [clause[ [,] clause] ... ] new-line
    for-loops
```

where *clause* can be any of the clauses accepted by the **target** or **parallel for** directives, except for **copyin**, with identical meanings and restrictions.

C / C++

Fortran

The syntax of the target parallel loop construct is as follows:

```
!$omp target parallel do [clause[ [,] clause] ... ]
    do-loops
[!$omp end target parallel do]
```

where *clause* can be any of the clauses accepted by the **target** or **parallel do** directives, except for **copyin**, with identical meanings and restrictions.

If an **end target parallel do** directive is not specified, an **end target parallel do** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **target** directive immediately followed by a parallel loop directive.

Restrictions

The restrictions for the **target** and parallel loop constructs apply except for the following explicit modifications:

- If any **if** clause on the directive includes a *directive-name-modifier* then all **if** clauses on the directive must include a *directive-name-modifier*.
- At most one **if** clause without a *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **parallel** *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **target** *directive-name-modifier* can appear on the directive.

Cross References

- **target** construct, see Section 2.15.5 on page 141.
- Parallel loop construct, see Section 2.16.1 on page 169.
- **if** Clause, see Section 2.17 on page 192.
- Data attribute clauses, see Section 2.20.3 on page 249.

2.16.7 Target Parallel Loop SIMD Construct

Summary

The target parallel loop SIMD construct is a shortcut for specifying a **target** construct containing a parallel loop SIMD construct and no other statements.

Syntax

C / C++

The syntax of the target parallel loop SIMD construct is as follows:

```
#pragma omp target parallel for simd [clause[[,] clause] ... ] new-line
    for-loops
```

where *clause* can be any of the clauses accepted by the **target** or **parallel for simd** directives, except for **copyin**, with identical meanings and restrictions.

C / C++

Fortran

The syntax of the target parallel loop SIMD construct is as follows:

```
!$omp target parallel do simd [clause[ [,] clause] ... ]
    do-loops
[!$omp end target parallel do simd]
```

where *clause* can be any of the clauses accepted by the **target** or **parallel do simd** directives, except for **copyin**, with identical meanings and restrictions.

If an **end target parallel do simd** directive is not specified, an **end target parallel do simd** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **target** directive immediately followed by a parallel loop SIMD directive.

Restrictions

The restrictions for the **target** and parallel loop SIMD constructs apply except for the following explicit modifications:

- If any **if** clause on the directive includes a *directive-name-modifier* then all **if** clauses on the directive must include a *directive-name-modifier*.
- At most one **if** clause without a *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **parallel** *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **target** *directive-name-modifier* can appear on the directive.

Cross References

- **target** construct, see Section 2.15.5 on page 141.
- Parallel loop SIMD construct, see Section 2.16.4 on page 173.
- **if** Clause, see Section 2.17 on page 192.
- Data attribute clauses, see Section 2.20.3 on page 249.

2.16.8 target simd Construct

Summary

The **target simd** construct is a shortcut for specifying a **target** construct containing a **simd** construct and no other statements.

Syntax

C / C++

The syntax of the **target simd** construct is as follows:

```
#pragma omp target simd [clause[ [,] clause] ... ] new-line
    for-loops
```

where *clause* can be any of the clauses accepted by the **target** or **simd** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **target simd** construct is as follows:

```
!$omp target simd [clause[ [,] clause] ... ]
      do-loops
[!$omp end target simd]
```

where *clause* can be any of the clauses accepted by the **target** or **simd** directives with identical meanings and restrictions.

If an **end target simd** directive is not specified, an **end target simd** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **target** directive immediately followed by a **simd** directive.

Restrictions

The restrictions for the **target** and **simd** constructs apply.

Cross References

- **simd** construct, see Section 2.11.1 on page 96.
- **target** construct, see Section 2.15.5 on page 141.
- Data attribute clauses, see Section 2.20.3 on page 249.

2.16.9 target teams Construct

Summary

The **target teams** construct is a shortcut for specifying a **target** construct containing a **teams** construct and no other statements.

1 Syntax

C / C++

2 The syntax of the **target teams** construct is as follows:

```
#pragma omp target teams [clause[ [,] clause] ... ] new-line
    structured-block
```

3 where *clause* can be any of the clauses accepted by the **target** or **teams** directives with identical
4 meanings and restrictions.

C / C++

Fortran

5 The syntax of the **target teams** construct is as follows:

```
!$omp target teams [clause[ [,] clause] ... ]
    structured-block
!$omp end target teams
```

6 where *clause* can be any of the clauses accepted by the **target** or **teams** directives with identical
7 meanings and restrictions.

Fortran

8 Description

9 The semantics are identical to explicitly specifying a **target** directive immediately followed by a
10 **teams** directive.

11 Restrictions

12 The restrictions for the **target** and **teams** constructs apply except for the following explicit
13 modifications:

14 • If an **allocator** clause specifies an *allocator* it can only be a predefined allocator variable.

15 Cross References

16 • **target** construct, see Section 2.15.5 on page 141.

17 • **teams** construct, see Section 2.15.9 on page 157.

18 • Data attribute clauses, see Section 2.20.3 on page 249.

2.16.10 **teams distribute** Construct

Summary

The **teams distribute** construct is a shortcut for specifying a **teams** construct containing a **distribute** construct and no other statements.

Syntax

C / C++

The syntax of the **teams distribute** construct is as follows:

```
#pragma omp teams distribute [clause[ [,] clause] ... ] new-line
    for-loops
```

where *clause* can be any of the clauses accepted by the **teams** or **distribute** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **teams distribute** construct is as follows:

```
!$omp teams distribute [clause[ [,] clause] ... ]
    do-loops
[!$omp end teams distribute]
```

where *clause* can be any of the clauses accepted by the **teams** or **distribute** directives with identical meanings and restrictions.

If an **end teams distribute** directive is not specified, an **end teams distribute** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **teams** directive immediately followed by a **distribute** directive. The effect of any clause that applies to both constructs is as if it were applied to both constructs separately.

Restrictions

The restrictions for the **teams** and **distribute** constructs apply.

1 Cross References

2 • **teams** construct, see Section 2.15.9 on page 157.

3 • **distribute** construct, see Section 2.15.10 on page 160.

4 • Data attribute clauses, see Section 2.20.3 on page 249.

5 2.16.11 teams distribute simd Construct

6 Summary

7 The **teams distribute simd** construct is a shortcut for specifying a **teams** construct
8 containing a **distribute simd** construct and no other statements.

9 Syntax

C / C++

10 The syntax of the **teams distribute simd** construct is as follows:

```
#pragma omp teams distribute simd [clause[ [ , ] clause] ... ] new-line
      for-loops
```

11 where *clause* can be any of the clauses accepted by the **teams** or **distribute simd** directives
12 with identical meanings and restrictions.

C / C++

Fortran

13 The syntax of the **teams distribute simd** construct is as follows:

```
!$omp teams distribute simd [clause[ [ , ] clause] ... ]
      do-loops
[!$omp end teams distribute simd]
```

14 where *clause* can be any of the clauses accepted by the **teams** or **distribute simd** directives
15 with identical meanings and restrictions.

16 If an **end teams distribute simd** directive is not specified, an
17 **end teams distribute simd** directive is assumed at the end of the *do-loops*.

Fortran

1 **Description**

2 The semantics are identical to explicitly specifying a **teams** directive immediately followed by a
3 **distribute simd** directive. The effect of any clause that applies to both constructs is as if it
4 were applied to both constructs separately.

5 **Restrictions**

6 The restrictions for the **teams** and **distribute simd** constructs apply.

7 **Cross References**

8 • **teams** construct, see Section 2.15.9 on page 157.

9 • **distribute simd** construct, see Section 2.15.11 on page 164.

10 • Data attribute clauses, see Section 2.20.3 on page 249.

11 ## 2.16.12 `target teams distribute` Construct

12 **Summary**

13 The **target teams distribute** construct is a shortcut for specifying a **target** construct
14 containing a **teams distribute** construct and no other statements.

15 **Syntax**

C / C++

16 The syntax of the **target teams distribute** construct is as follows:

```
#pragma omp target teams distribute [clause[ [,] clause] ... ] new-line
    for-loops
```

17 where *clause* can be any of the clauses accepted by the **target** or **teams distribute**
18 directives with identical meanings and restrictions.

C / C++

Fortran

19 The syntax of the **target teams distribute** construct is as follows:

```
!$omp target teams distribute [clause[ [,] clause] ... ]
    do-loops
[!$omp end target teams distribute]
```

where *clause* can be any of the clauses accepted by the **target** or **teams distribute** directives with identical meanings and restrictions.

If an **end target teams distribute** directive is not specified, an **end target teams distribute** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **target** directive immediately followed by a **teams distribute** directive.

Restrictions

The restrictions for the **target** and **teams distribute** constructs apply except for the following explicit modifications:

- If an **allocator** clause specifies an *allocator* it can only be a predefined allocator variable.

Cross References

- **target** construct, see Section 2.15.2 on page 132.
- **teams distribute** construct, see Section 2.16.10 on page 182.
- Data attribute clauses, see Section 2.20.3 on page 249.

2.16.13 target teams distribute simd Construct

Summary

The **target teams distribute simd** construct is a shortcut for specifying a **target** construct containing a **teams distribute simd** construct and no other statements.

1 ### Syntax

C / C++

2 The syntax of the **target teams distribute simd** construct is as follows:

```
#pragma omp target teams distribute simd [clause[ [,] clause] ... ] new-line
    for-loops
```

3 where *clause* can be any of the clauses accepted by the **target** or **teams distribute simd**
4 directives with identical meanings and restrictions.

C / C++

Fortran

5 The syntax of the **target teams distribute simd** construct is as follows:

```
!$omp target teams distribute simd [clause[ [,] clause] ... ]
    do-loops
[!$omp end target teams distribute simd]
```

6 where *clause* can be any of the clauses accepted by the **target** or **teams distribute simd**
7 directives with identical meanings and restrictions.

8 If an **end target teams distribute simd** directive is not specified, an
9 **end target teams distribute simd** directive is assumed at the end of the *do-loops*.

Fortran

10 ### Description

11 The semantics are identical to explicitly specifying a **target** directive immediately followed by a
12 **teams distribute simd** directive.

13 ### Restrictions

14 The restrictions for the **target** and **teams distribute simd** constructs apply.

15 ### Cross References

16 • **target** construct, see Section 2.15.2 on page 132.

17 • **teams distribute simd** construct, see Section 2.16.11 on page 183.

18 • Data attribute clauses, see Section 2.20.3 on page 249.

2.16.14 Teams Distribute Parallel Loop Construct

Summary

The teams distribute parallel loop construct is a shortcut for specifying a **teams** construct containing a distribute parallel loop construct and no other statements.

Syntax

C / C++

The syntax of the teams distribute parallel loop construct is as follows:

```
#pragma omp teams distribute parallel for [clause[ [,] clause] ... ] new-line
    for-loops
```

where *clause* can be any of the clauses accepted by the **teams** or **distribute parallel for** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the teams distribute parallel loop construct is as follows:

```
!$omp teams distribute parallel do [clause[ [,] clause] ... ]
    do-loops
[ !$omp end teams distribute parallel do ]
```

where *clause* can be any of the clauses accepted by the **teams** or **distribute parallel do** directives with identical meanings and restrictions.

If an **end teams distribute parallel do** directive is not specified, an **end teams distribute parallel do** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **teams** directive immediately followed by a distribute parallel loop directive. The effect of any clause that applies to both constructs is as if it were applied to both constructs separately.

Restrictions

The restrictions for the **teams** and distribute parallel loop constructs apply.

Cross References

- **teams** construct, see Section 2.15.9 on page 157.
- Distribute parallel loop construct, see Section 2.15.12 on page 165.
- Data attribute clauses, see Section 2.20.3 on page 249.

2.16.15 Target Teams Distribute Parallel Loop Construct

Summary

The target teams distribute parallel loop construct is a shortcut for specifying a **target** construct containing a teams distribute parallel loop construct and no other statements.

Syntax

C / C++

The syntax of the target teams distribute parallel loop construct is as follows:

```
#pragma omp target teams distribute parallel for [clause[ [ , ] clause] ... ] new-line
    for-loops
```

where *clause* can be any of the clauses accepted by the **target** or **teams distribute parallel for** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the target teams distribute parallel loop construct is as follows:

```
!$omp target teams distribute parallel do [clause[ [ , ] clause] ... ]
    do-loops
[!$omp end target teams distribute parallel do]
```

where *clause* can be any of the clauses accepted by the **target** or **teams distribute parallel do** directives with identical meanings and restrictions.

If an **end target teams distribute parallel do** directive is not specified, an **end target teams distribute parallel do** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **target** directive immediately followed by a teams distribute parallel loop directive.

Restrictions

The restrictions for the **target** and teams distribute parallel loop constructs apply except for the following explicit modifications:

- If any **if** clause on the directive includes a *directive-name-modifier* then all **if** clauses on the directive must include a *directive-name-modifier*.
- At most one **if** clause without a *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **parallel** *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **target** *directive-name-modifier* can appear on the directive.

Cross References

- **target** construct, see Section 2.15.5 on page 141.
- Teams distribute parallel loop construct, see Section 2.16.14 on page 187.
- **if** Clause, see Section 2.17 on page 192.
- Data attribute clauses, see Section 2.20.3 on page 249.

2.16.16 Teams Distribute Parallel Loop SIMD Construct

Summary

The teams distribute parallel loop SIMD construct is a shortcut for specifying a **teams** construct containing a distribute parallel loop SIMD construct and no other statements.

Syntax

C / C++

The syntax of the teams distribute parallel loop construct is as follows:

```
#pragma omp teams distribute parallel for simd [clause[ [,] clause] ... ] new-line
    for-loops
```

where *clause* can be any of the clauses accepted by the **teams** or **distribute parallel for simd** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the teams distribute parallel loop construct is as follows:

```
!$omp teams distribute parallel do simd [clause[ [,] clause] ... ]
    do-loops
[!$omp end teams distribute parallel do simd]
```

where *clause* can be any of the clauses accepted by the **teams** or **distribute parallel do simd** directives with identical meanings and restrictions.

If an **end teams distribute parallel do simd** directive is not specified, an **end teams distribute parallel do simd** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **teams** directive immediately followed by a distribute parallel loop SIMD directive. The effect of any clause that applies to both constructs is as if it were applied to both constructs separately.

Restrictions

The restrictions for the **teams** and distribute parallel loop SIMD constructs apply.

Cross References

- **teams** construct, see Section 2.15.9 on page 157.
- Distribute parallel loop SIMD construct, see Section 2.15.13 on page 167.
- Data attribute clauses, see Section 2.20.3 on page 249.

2.16.17 Target Teams Distribute Parallel Loop SIMD Construct

Summary

The target teams distribute parallel loop SIMD construct is a shortcut for specifying a **target** construct containing a teams distribute parallel loop SIMD construct and no other statements.

Syntax

C / C++

The syntax of the target teams distribute parallel loop SIMD construct is as follows:

```
#pragma omp target teams distribute parallel for simd \
            [clause[ [ , ] clause] ... ] new-line
    for-loops
```

where *clause* can be any of the clauses accepted by the **target** or **teams distribute parallel for simd** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the target teams distribute parallel loop SIMD construct is as follows:

```
!$omp target teams distribute parallel do simd [clause[ [ , ] clause] ... ]
    do-loops
[!$omp end target teams distribute parallel do simd]
```

where *clause* can be any of the clauses accepted by the **target** or **teams distribute parallel do simd** directives with identical meanings and restrictions.

If an **end target teams distribute parallel do simd** directive is not specified, an **end target teams distribute parallel do simd** directive is assumed at the end of the *do-loops*.

Fortran

Description

The semantics are identical to explicitly specifying a **target** directive immediately followed by a teams distribute parallel loop SIMD directive.

Restrictions

The restrictions for the **target** and teams distribute parallel loop SIMD constructs apply except for the following explicit modifications:

- If any **if** clause on the directive includes a *directive-name-modifier* then all **if** clauses on the directive must include a *directive-name-modifier*.
- At most one **if** clause without a *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **parallel** *directive-name-modifier* can appear on the directive.
- At most one **if** clause with the **target** *directive-name-modifier* can appear on the directive.

Cross References

- **target** construct, see Section 2.15.5 on page 141.
- Teams distribute parallel loop SIMD construct, see Section 2.16.16 on page 189.
- **if** Clause, see Section 2.17 on page 192.
- Data attribute clauses, see Section 2.20.3 on page 249.

2.17 if Clause

Summary

The semantics of an **if** clause are described in the section on the construct to which it applies. The **if** clause *directive-name-modifier* names the associated construct to which an expression applies, and is particularly useful for composite and combined constructs.

Syntax

C / C++

The syntax of the **if** clause is as follows:

```
if([ directive-name-modifier :] scalar-expression)
```

C / C++

Fortran

1 The syntax of the **if** clause is as follows:

```
if ([ directive-name-modifier :] scalar-logical-expression)
```

Fortran

2 ### Description

3 The effect of the **if** clause depends on the construct to which it is applied. For combined or
4 composite constructs, the **if** clause only applies to the semantics of the construct named in the
5 *directive-name-modifier* if one is specified. If no *directive-name-modifier* is specified for a
6 combined or composite construct then the **if** clause applies to all constructs to which an **if** clause
7 can apply.

8 ## 2.18 Master and Synchronization Constructs
9 and Clauses

10 OpenMP provides the following synchronization constructs:

11 - the **master** construct;
12 - the **critical** construct;
13 - the **barrier** construct;
14 - the **taskwait** construct;
15 - the **taskgroup** construct;
16 - the **atomic** construct;
17 - the **flush** construct;
18 - the **ordered** construct.

2.18.1 `master` Construct

Summary

The **`master`** construct specifies a structured block that is executed by the master thread of the team.

Syntax

C / C++

The syntax of the **`master`** construct is as follows:

```
#pragma omp master new-line
    structured-block
```

C / C++

Fortran

The syntax of the **`master`** construct is as follows:

```
!$omp master
    structured-block
!$omp end master
```

Fortran

Binding

The binding thread set for a **`master`** region is the current team. A **`master`** region binds to the innermost enclosing **`parallel`** region. Only the master thread of the team executing the binding **`parallel`** region participates in the execution of the structured block of the **`master`** region.

Description

Other threads in the team do not execute the associated structured block. There is no implied barrier either on entry to, or exit from, the **`master`** construct.

Events

The *master-begin* event occurs in the thread encountering the **`master`** construct on entry to the master region, if it is the master thread of the team.

The *master-end* event occurs in the thread encountering the **`master`** construct on exit of the master region, if it is the master thread of the team.

Tool Callbacks

A thread dispatches a registered **ompt_callback_master** callback for each occurrence of a *master-begin* and a *master-end* event in that thread.

The callback occurs in the context of the task executed by the master thread. This callback has the type signature **ompt_callback_master_t**. The callback receives **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate.

Restrictions

C++

- A throw executed inside a **master** region must cause execution to resume within the same **master** region, and the same thread that threw the exception must catch it

C++

Cross References

- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.1.3.4.10 on page 397.
- **ompt_callback_master_t**, see Section 4.1.4.2.6 on page 411.

2.18.2 critical Construct

Summary

The **critical** construct restricts execution of the associated structured block to a single thread at a time.

Syntax

C / C++

The syntax of the **critical** construct is as follows:

```
#pragma omp critical [(name) [[,] hint(hint-expression)] ] new-line
   structured-block
```

where *hint-expression* is an integer constant expression that evaluates to a valid synchronization hint (as described in Section 2.18.11 on page 229).

C / C++

Fortran

The syntax of the **critical** construct is as follows:

```
!$omp critical [(name) [[,] hint(hint-expression)] ]
      structured-block
!$omp end critical [(name)]
```

where *hint-expression* is a constant expression that evaluates to a scalar value with kind **omp_sync_hint_kind** and a value that is a valid synchronization hint (as described in Section 2.18.11 on page 229).

Fortran

Binding

The binding thread set for a **critical** region is all threads in the contention group. The region is executed as if only a single thread at a time among all threads in the contention group is entering the region for execution, without regard to the team(s) to which the threads belong.

Description

An optional *name* may be used to identify the **critical** construct. All **critical** constructs without a name are considered to have the same unspecified name.

C / C++

Identifiers used to identify a **critical** construct have external linkage and are in a name space that is separate from the name spaces used by labels, tags, members, and ordinary identifiers.

C / C++

Fortran

The names of **critical** constructs are global entities of the program. If a name conflicts with any other entity, the behavior of the program is unspecified.

Fortran

The threads of a contention group execute the **critical** region as if only one thread of the contention group is executing the **critical** region at a time. The **critical** construct enforces these execution semantics with respect to all **critical** constructs with the same name in all threads in the contention group, not just those threads in the current team.

If present, the **hint** clause gives the implementation additional information about the expected runtime properties of the **critical** region that can optionally be used to optimize the implementation. The presence of a **hint** clause does not affect the isolation guarantees provided by the **critical** construct. If no **hint** clause is specified, the effect is as if **hint(omp_sync_hint_none)** had been specified.

Events

The *critical-acquire* event occurs in the thread encountering the **critical** construct on entry to the critical region before initiating synchronization for the region.

The *critical-acquired* event occurs in the thread encountering the **critical** construct after entering the region, but before executing the structured block of the **critical** region.

The *critical-release* event occurs in the thread encountering the **critical** construct after completing any synchronization on exit from the **critical** region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_mutex_acquire** callback for each occurrence of a *critical-acquire* event in that thread. This callback has the type signature **ompt_callback_mutex_acquire_t**.

A thread dispatches a registered **ompt_callback_mutex_acquired** callback for each occurrence of a *critical-acquired* event in that thread. This callback has the type signature **ompt_callback_mutex_t**.

A thread dispatches a registered **ompt_callback_mutex_released** callback for each occurrence of a *critical-release* event in that thread. This callback has the type signature **ompt_callback_mutex_t**. The callbacks occur in the task encountering the critical construct. The callbacks should receive **ompt_mutex_critical** as their *kind* argument if practical, but a less specific kind is acceptable.

Restrictions

- If the **hint** clause is specified, the **critical** construct must have a *name*.
- If the **hint** clause is specified, each of the **critical** constructs with the same *name* must have a **hint** clause for which the *hint-expression* evaluates to the same value.

C++

- A throw executed inside a **critical** region must cause execution to resume within the same **critical** region, and the same thread that threw the exception must catch it.

C++

Fortran

The following restrictions apply to the critical construct:

- If a *name* is specified on a **critical** directive, the same *name* must also be specified on the **end critical** directive.
- If no *name* appears on the **critical** directive, no *name* can appear on the **end critical** directive.

Fortran

Cross References

- Synchronization Hints, see Section 2.18.11 on page 229.
- **ompt_mutex_critical**, see Section 4.1.3.4.14 on page 399.
- **ompt_callback_mutex_acquire_t**, see Section 4.1.4.2.13 on page 419.
- **ompt_callback_mutex_t**, see Section 4.1.4.2.14 on page 420.

2.18.3 barrier Construct

Summary

The **barrier** construct specifies an explicit barrier at the point at which the construct appears. The **barrier** construct is a stand-alone directive.

Syntax

C / C++

The syntax of the **barrier** construct is as follows:

```
#pragma omp barrier new-line
```

C / C++

Fortran

The syntax of the **barrier** construct is as follows:

```
!$omp barrier
```

Fortran

Binding

The binding thread set for a **barrier** region is the current team. A **barrier** region binds to the innermost enclosing **parallel** region.

Description

All threads of the team executing the binding **parallel** region must execute the **barrier** region and complete execution of all explicit tasks bound to this **parallel** region before any are allowed to continue execution beyond the barrier.

The **barrier** region includes an implicit task scheduling point in the current task region.

Events

The *barrier-begin* event occurs in each thread encountering the **barrier** construct on entry to the **barrier** region.

The *barrier-wait-begin* event occurs when a task begins an interval of active or passive waiting in a **barrier** region.

The *barrier-wait-end* event occurs when a task ends an interval of active or passive waiting and resumes execution in a **barrier** region.

The *barrier-end* event occurs in each thread encountering the **barrier** construct after the barrier synchronization on exit from the **barrier** region.

A *cancellation* event occurs if cancellation is activated at an implicit cancellation point in an barrier region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_sync_region** callback for each occurrence of a *barrier-begin* and *barrier-end* event in that thread. The callback occurs in the task encountering the barrier construct. This callback has the type signature **ompt_callback_sync_region_t**. The callback receives **ompt_sync_region_barrier** as its *kind* argument and **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate.

A thread dispatches a registered **ompt_callback_sync_region_wait** callback for each occurrence of a *barrier-wait-begin* and *barrier-wait-end* event. This callback has type signature **ompt_callback_sync_region_t**. This callback executes in the context of the task that encountered the **barrier** construct. The callback receives **ompt_sync_region_barrier** as its *kind* argument and **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate.

A thread dispatches a registered **ompt_callback_cancel** callback for each occurrence of a *cancellation* event in that thread. The callback occurs in the context of the encountering task. The callback has type signature **ompt_callback_cancel_t**. The callback receives **ompt_cancel_detected** as its *flags* argument.

Restrictions

The following restrictions apply to the **barrier** construct:

- Each **barrier** region must be encountered by all threads in a team or by none at all, unless cancellation has been requested for the innermost enclosing parallel region.
- The sequence of worksharing regions and **barrier** regions encountered must be the same for every thread in a team.

Cross References

- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.1.3.4.10 on page 397.
- **ompt_sync_region_barrier**, see Section 4.1.3.4.11 on page 398.
- **ompt_callback_sync_region_t**, see Section 4.1.4.2.12 on page 418.
- **ompt_callback_cancel_t**, see Section 4.1.4.2.27 on page 437.

2.18.4 Implicit Barriers

Implicit tasks in a parallel region synchronize with one another using implicit barriers at the end of worksharing constructs and at the end of the **parallel** region. This section describes the OMPT events and tool callbacks associated with implicit barriers.

Implicit barriers are task scheduling points. For a description of task sheduling points, associated events, and tool callbacks, see Section 2.13.6 on page 125.

Events

A *cancellation* event occurs if cancellation is activated at an implicit cancellation point in an implicit barrier region.

The *implicit-barrier-begin* event occurs in each implicit task at the beginning of an implicit barrier.

The *implicit-barrier-wait-begin* event occurs when a task begins an interval of active or passive waiting while executing in an implicit barrier region.

The *implicit-barrier-wait-end* event occurs when a task ends an interval of active or waiting and resumes execution of an implicit barrier region.

The *implicit-barrier-end* event occurs in each implicit task at the end of an implicit barrier.

Tool Callbacks

A thread dispatches a registered **ompt_callback_sync_region** callback for each occurrence of a *implicit-barrier-begin* and *implicit-barrier-end* event in that thread. The callback occurs in the implicit task executing in a parallel region. This callback has the type signature **ompt_callback_sync_region_t**. The callback receives **ompt_sync_region_barrier** as its *kind* argument and **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate.

A thread dispatches a registered **ompt_callback_cancel** callback for each occurrence of a *cancellation* event in that thread. The callback occurs in the context of the encountering task. The callback has type signature **ompt_callback_cancel_t**. The callback receives **ompt_cancel_detected** as its *flags* argument.

A thread dispatches a registered **ompt_callback_sync_region_wait** callback for each occurrence of a *implicit-barrier-wait-begin* and *implicit-barrier-wait-end* event. This callback has type signature **ompt_callback_sync_region_t**. The callback occurs in each implicit task participating in an implicit barrier. The callback receives **ompt_sync_region_barrier** as its *kind* argument and **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate.

Restrictions

If a thread is in the state **omp_state_wait_barrier_implicit_parallel**, a call to **ompt_get_parallel_info** may return a pointer to a copy of the current parallel region's *parallel_data* rather than a pointer to the data word for the region itself. This convention enables the master thread for a parallel region to free storage for the region immediately after the region ends, yet avoid having some other thread in the region's team potentially reference the region's *parallel_data* object after it has been freed.

Cross References

- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.1.3.4.10 on page 397.
- **ompt_sync_region_barrier**, see Section 4.1.3.4.11 on page 398
- **ompt_cancel_detected**, see Section 4.1.3.4.22 on page 403.
- **ompt_callback_sync_region_t**, see Section 4.1.4.2.12 on page 418.
- **ompt_callback_cancel_t**, see Section 4.1.4.2.27 on page 437.

2.18.5 `taskwait` Construct

Summary

The **`taskwait`** construct specifies a wait on the completion of child tasks of the current task. The **`taskwait`** construct is a stand-alone directive.

Syntax

C / C++

The syntax of the **`taskwait`** construct is as follows:

```
#pragma omp taskwait [clause[ [,] clause] ... ] new-line
```

where *clause* is one of the following:

`depend(`*dependence-type* **:** *locator-list* **[:** *iterators-definition* **])**

C / C++

Fortran

The syntax of the **`taskwait`** construct is as follows:

```
!$omp taskwait [clause[ [,] clause] ... ]
```

where *clause* is one of the following:

`depend(`*dependence-type* **:** *locator-list* **[:** *iterators-definition* **])**

Fortran

Binding

The **`taskwait`** region binds to the current task region. The binding thread set of the **`taskwait`** region is the current team.

Description

If no **depend** clause is present on the **taskwait** construct, the current task region is suspended at an implicit task scheduling point associated with the construct. The current task region remains suspended until all child tasks that it generated before the **taskwait** region complete execution.

Otherwise, if one or more **depend** clauses are present on the **taskwait** construct, the behavior is as if these clauses were applied to a **task** construct with an empty associated structured block that generates a *mergeable* and *included task*. Thus, the current task region is suspended until the *predecessor tasks* of this task complete execution.

Events

The *taskwait-begin* event occurs in each thread encountering the **taskwait** construct on entry to the **taskwait** region.

The *taskwait-wait-begin* event occurs when a task begins an interval of active or passive waiting in a **taskwait** region.

The *taskwait-wait-end* event occurs when a task ends an interval of active or passive waiting and resumes execution in a **taskwait** region.

The *taskwait-end* event occurs in each thread encountering the **taskwait** construct after the taskwait synchronization on exit from the **taskwait** region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_sync_region** callback for each occurrence of a *taskwait-begin* and *taskwait-end* event in that thread. The callback occurs in the task encountering the taskwait construct. This callback has the type signature **ompt_callback_sync_region_t**. The callback receives **ompt_sync_region_taskwait** as its *kind* argument and **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate.

A thread dispatches a registered **ompt_callback_sync_region_wait** callback for each occurrence of a *taskwait-wait-begin* and *taskwait-wait-end* event. This callback has type signature **ompt_callback_sync_region_t**. This callback executes in the context of the task that encountered the **taskwait** construct. The callback receives **ompt_sync_region_taskwait** as its *kind* argument and **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate.

Restrictions

The **mutexinoutset** *dependence-type* may not appear in a **depend** clause on a **taskwait** construct.

Cross References

- **task** construct, see Section 2.13.1 on page 110.
- Task scheduling, see Section 2.13.6 on page 125.
- **depend** clause, see Section 2.18.10 on page 225.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.1.3.4.10 on page 397.
- **ompt_sync_region_taskwait**, see Section 4.1.3.4.11 on page 398.
- **ompt_callback_sync_region_t**, see Section 4.1.4.2.12 on page 418.

2.18.6 taskgroup Construct

Summary

The **taskgroup** construct specifies a wait on completion of child tasks of the current task and their descendent tasks.

Syntax

C / C++

The syntax of the **taskgroup** construct is as follows:

```
#pragma omp taskgroup [clause[[,] clause] ...] new-line
    structured-block
```

where *clause* is one of the following:

task_reduction(*reduction-identifier* **:** *list***)**

allocate(*[allocator:]list***)**

C / C++

Fortran

The syntax of the **taskgroup** construct is as follows:

```
!$omp taskgroup [clause [ [,] clause] ...]
      structured-block
!$omp end taskgroup
```

where *clause* is one of the following:

task_reduction(*reduction-identifier* **:** *list***)**

allocate(*[allocator:]list***)**

Fortran

Binding

A **taskgroup** region binds to the current task region. A **taskgroup** region binds to the innermost enclosing **parallel** region.

Description

When a thread encounters a **taskgroup** construct, it starts executing the region. All child tasks generated in the **taskgroup** region and all of their descendants that bind to the same **parallel** region as the **taskgroup** region are part of the *taskgroup set* associated with the **taskgroup** region.

There is an implicit task scheduling point at the end of the **taskgroup** region. The current task is suspended at the task scheduling point until all tasks in the *taskgroup set* complete execution.

Events

The *taskgroup-begin* event occurs in each thread encountering the **taskgroup** construct on entry to the **taskgroup** region.

The *taskgroup-wait-begin* event occurs when a task begins an interval of active or passive waiting in a **taskgroup** region.

The *taskgroup-wait-end* event occurs when a task ends an interval of active or passive waiting and resumes execution in a **taskgroup** region.

The *taskgroup-end* event occurs in each thread encountering the **taskgroup** construct after the taskgroup synchronization on exit from the **taskgroup** region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_sync_region** callback for each occurrence of a *taskgroup-begin* and *taskgroup-end* event in that thread. The callback occurs in the task encountering the taskgroup construct. This callback has the type signature **ompt_callback_sync_region_t**. The callback receives **ompt_sync_region_taskgroup** as its *kind* argument and **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate.

A thread dispatches a registered **ompt_callback_sync_region_wait** callback for each occurrence of a *taskgroup-wait-begin* and *taskgroup-wait-end* event. This callback has type signature **ompt_callback_sync_region_t**. This callback executes in the context of the task that encountered the **taskgroup** construct. The callback receives **ompt_sync_region_taskgroup** as its *kind* argument and **ompt_scope_begin** or **ompt_scope_end** as its *endpoint* argument, as appropriate.

Cross References

- Task scheduling, see Section 2.13.6 on page 125.
- **task_reduction** Clause, see Section 2.20.4.5 on page 273.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.1.3.4.10 on page 397.
- **ompt_sync_region_taskgroup**, see Section 4.1.3.4.11 on page 398.
- **ompt_callback_sync_region_t**, see Section 4.1.4.2.12 on page 418.

2.18.7 atomic Construct

Summary

The **atomic** construct ensures that a specific storage location is accessed atomically, rather than exposing it to the possibility of multiple, simultaneous reading and writing threads that may result in indeterminate values.

Syntax

In the following syntax, *atomic-clause* is a clause that indicates the semantics for which atomicity is enforced and *memory-order-clause* is a clause that indicates the memory ordering behavior of the construct. Specifically, *atomic-clause* is one of the following:

1 **read**
2 **write**
3 **update**
4 **capture**

5 and *memory-order-clause* is one of the following:

6 **seq_cst**
7 **acq_rel**
8 **release**
9 **acquire**

C / C++

10 The syntax of the **atomic** construct takes one of the following forms:

```
#pragma omp atomic [memory-order-clause[,]] atomic-clause
                   [[,]hint(hint-expression)]
                   [[,]memory-order-clause] new-line
    expression-stmt
```

11 or

```
#pragma omp atomic [memory-order-clause] [[,]hint(hint-expression)] new-line
    expression-stmt
```

12 or

```
#pragma omp atomic [memory-order-clause[,]] capture
                   [[,][hint(hint-expression)]
                   [[,]memory-order-clause] new-line
    structured-block
```

13 where *expression-stmt* is an expression statement with one of the following forms:

14 • If *atomic-clause* is **read**:
15 *v* = *x*;

16 • If *atomic-clause* is **write**:
17 *x* = *expr*;

- If *atomic-clause* is **update** or not present:

```
x++;
x--;
++x;
--x;
x binop= expr;
x = x binop expr;
x = expr binop x;
```

- If *atomic-clause* is **capture**:

```
v = x++;
v = x--;
v = ++x;
v = --x;
v = x binop= expr;
v = x = x binop expr;
v = x = expr binop x;
```

and where *structured-block* is a structured block with one of the following forms:

```
{v = x; x binop= expr;}
{x binop= expr; v = x;}
{v = x; x = x binop expr;}
{v = x; x = expr binop x;}
{x = x binop expr; v = x;}
{x = expr binop x; v = x;}
{v = x; x = expr;}
{v = x; x++;}
{v = x; ++x;}
{++x; v = x;}
{x++; v = x;}
{v = x; x--;}
{v = x; --x;}
{--x; v = x;}
{x--; v = x;}
```

In the preceding expressions:

- *x* and *v* (as applicable) are both *l-value* expressions with scalar type.
- During the execution of an atomic region, multiple syntactic occurrences of *x* must designate the same storage location.
- Neither of *v* and *expr* (as applicable) may access the storage location designated by *x*.

1 • Neither of *x* and *expr* (as applicable) may access the storage location designated by *v*.

2 • *expr* is an expression with scalar type.

3 • *binop* is one of **+**, *****, **-**, **/**, **&**, **^**, **|**, **<<**, or **>>**.

4 • *binop*, *binop***=**, **++**, and **--** are not overloaded operators.

5 • The expression *x binop expr* must be numerically equivalent to *x binop (expr)*. This requirement
6 is satisfied if the operators in *expr* have precedence greater than *binop*, or by using parentheses
7 around *expr* or subexpressions of *expr*.

8 • The expression *expr binop x* must be numerically equivalent to *(expr) binop x*. This requirement
9 is satisfied if the operators in *expr* have precedence equal to or greater than *binop*, or by using
10 parentheses around *expr* or subexpressions of *expr*.

11 • For forms that allow multiple occurrences of *x*, the number of times that *x* is evaluated is
12 unspecified.

C / C++

Fortran

13 The syntax of the **atomic** construct takes any of the following forms:

```
!$omp atomic [memory-order-clause[,]] read [[,]hint(hint-expression)]
             [[,]memory-order-clause]
    capture-statement
[!$omp end atomic]
```

14 or

```
!$omp atomic [memory-order-clause[,]] write [[,]hint(hint-expression)]
             [[,]memory-order-clause]
    write-statement
[!$omp end atomic]
```

15 or

```
!$omp atomic [memory-order-clause[,]] update [[,]hint(hint-expression)]
             [[,]memory-order-clause]
    update-statement
[!$omp end atomic]
```

16 or

```
!$omp atomic [memory-order-clause] [[,]hint(hint-expression)]
              [[,]memory-order-clause]
      update-statement
[!$omp end atomic]
```

1 or

```
!$omp atomic [memory-order-clause[,]] capture [[,]hint(hint-expression)]
              [[,]memory-order-clause]
      update-statement
      capture-statement
!$omp end atomic
```

2 or

```
!$omp atomic [memory-order-clause[,]] capture [[,]hint(hint-expression)]
              [[,]memory-order-clause]
      capture-statement
      update-statement
!$omp end atomic
```

3 or

```
!$omp atomic [memory-order-clause[,]] capture [[,]hint(hint-expression)]
              [[,]memory-order-clause]
      capture-statement
      write-statement
!$omp end atomic
```

4 where *write-statement* has the following form (if *atomic-clause* is **capture** or **write**):

5 *x* **=** *expr*

6 where *capture-statement* has the following form (if *atomic-clause* is **capture** or **read**):

7 *v* **=** *x*

8 and where *update-statement* has one of the following forms (if *atomic-clause* is **update**,
9 **capture**, or not present):

x **=** x *operator expr*

x **=** *expr operator* x

x **=** *intrinsic_procedure_name* **(**x**,** *expr_list***)**

x **=** *intrinsic_procedure_name* **(***expr_list***,** x**)**

In the preceding statements:

- x and v (as applicable) are both scalar variables of intrinsic type.
- x must not have the **ALLOCATABLE** attribute.
- During the execution of an atomic region, multiple syntactic occurrences of x must designate the same storage location.
- None of v, *expr*, and *expr_list* (as applicable) may access the same storage location as x.
- None of x, *expr*, and *expr_list* (as applicable) may access the same storage location as v.
- *expr* is a scalar expression.
- *expr_list* is a comma-separated, non-empty list of scalar expressions. If *intrinsic_procedure_name* refers to **IAND**, **IOR**, or **IEOR**, exactly one expression must appear in *expr_list*.
- *intrinsic_procedure_name* is one of **MAX**, **MIN**, **IAND**, **IOR**, or **IEOR**.
- *operator* is one of **+**, *****, **-**, **/**, **.AND.**, **.OR.**, **.EQV.**, or **.NEQV.**.
- The expression x *operator expr* must be numerically equivalent to x *operator (expr)*. This requirement is satisfied if the operators in *expr* have precedence greater than *operator*, or by using parentheses around *expr* or subexpressions of *expr*.
- The expression *expr operator* x must be numerically equivalent to *(expr) operator* x. This requirement is satisfied if the operators in *expr* have precedence equal to or greater than *operator*, or by using parentheses around *expr* or subexpressions of *expr*.
- *intrinsic_procedure_name* must refer to the intrinsic procedure name and not to other program entities.
- *operator* must refer to the intrinsic operator and not to a user-defined operator.
- All assignments must be intrinsic assignments.
- For forms that allow multiple occurrences of x, the number of times that x is evaluated is unspecified.

Fortran

1 **Binding**

2 If the size of x is 8, 16, 32, or 64 bits and x is aligned to a multiple of its size, the binding thread set
3 for the **atomic** region is all threads on the device. Otherwise, the binding thread set for the
4 **atomic** region is all threads in the contention group. **atomic** regions enforce exclusive access
5 with respect to other **atomic** regions that access the same storage location x among all threads in
6 the binding thread set without regard to the teams to which the threads belong.

7 **Description**

8 All **atomic** constructs force an atomic operation on the storage location designated by x to be
9 performed by the encountering thread, preceded and/or followed by implicit flush operations as
10 described in this section. The implicit flushes are performed as if they are part of the same atomic
11 operation applied to x. Each non-synchronizable implicit flush is a strong flush. If
12 *memory-order-clause* is present and is **seq_cst**, each synchronizable implicit flush is a strong
13 flush; otherwise, each synchronizable implicit flush is a weak flush.

14 The **atomic** construct with the **read** clause forces an atomic read of the location designated by x
15 regardless of the native machine word size. The atomic read is immediately preceded by a
16 non-synchronizable read flush of x.

17 The **atomic** construct with the **write** clause forces an atomic write of the location designated by
18 x regardless of the native machine word size. The atomic write is immediately followed by a
19 non-synchronizable write flush of x.

20 The **atomic** construct with the **update** clause forces an atomic update of the location designated
21 by x using the designated operator or intrinsic. Note that when no clause is present, the semantics
22 are equivalent to atomic update. Only the read and write of the location designated by x are
23 performed mutually atomically. The evaluation of *expr* or *expr_list* need not be atomic with respect
24 to the read or write of the location designated by x. No task scheduling points are allowed between
25 the read and the write of the location designated by x. The atomic update is immediately preceded
26 by a non-synchronizable read flush of x and immediately followed by a non-synchronizable write
27 flush of x.

28 The **atomic** construct with the **capture** clause forces an atomic captured update — an atomic
29 update of the location designated by x using the designated operator or intrinsic while also
30 capturing the original or final value of the location designated by x with respect to the atomic
31 update. The original or final value of the location designated by x is written in the location
32 designated by v depending on the form of the **atomic** construct structured block or statements
33 following the usual language semantics. Only the read and write of the location designated by x are
34 performed mutually atomically. Neither the evaluation of *expr* or *expr_list*, nor the write to the
35 location designated by v, need be atomic with respect to the read or write of the location designated
36 by x. No task scheduling points are allowed between the read and the write of the location
37 designated by x. The atomic captured update is immediately preceded by a non-synchronizable
38 read flush of x and immediately followed by a non-synchronizable write flush of x.

A release flush operation applied to all variables, with a sync-set containing *x*, is implied at entry to the atomic operation when the **read** clause is not present and the **release**, **acq_rel**, or **seq_cst** clause is present. An acquire flush operation applied to all variables, with a sync-set containing *x*, is implied at exit from the atomic operation when the **read** or **capture** clause is present and the **acquire**, **acq_rel**, or **seq_cst** clause is present.

For all forms of the **atomic** construct, any combination of two or more of these **atomic** constructs enforces mutually exclusive access to the locations designated by *x* among threads in the binding thread set. To avoid race conditions, all accesses of the locations designated by *x* that could potentially occur in parallel must be protected with an **atomic** construct.

atomic regions do not guarantee exclusive access with respect to any accesses outside of **atomic** regions to the same storage location *x* even if those accesses occur during a **critical** or **ordered** region, while an OpenMP lock is owned by the executing task, or during the execution of a **reduction** clause.

However, other OpenMP synchronization can ensure the desired exclusive access. For example, a barrier following a series of atomic updates to *x* guarantees that subsequent accesses do not form a race with the atomic accesses.

A compliant implementation may enforce exclusive access between **atomic** regions that update different storage locations. The circumstances under which this occurs are implementation defined.

If the storage location designated by *x* is not size-aligned (that is, if the byte alignment of *x* is not a multiple of the size of *x*), then the behavior of the **atomic** region is implementation defined.

If present, the **hint** clause gives the implementation additional information about the expected properties of the atomic operation that can optionally be used to optimize the implementation. The presence of a **hint** clause does not affect the semantics of the **atomic** construct, and it is legal to ignore all hints. If no **hint** clause is specified, the effect is as if **hint(omp_sync_hint_none)** had been specified.

Events

The *atomic-acquire* event occurs in the thread encountering the **atomic** construct on entry to the atomic region before initiating synchronization for the region.

The *atomic-acquired* event occurs in the thread encountering the **atomic** construct after entering the region, but before executing the structured block of the **atomic** region.

The *atomic-release* event occurs in the thread encountering the **atomic** construct after completing any synchronization on exit from the **atomic** region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_mutex_acquire** callback for each occurrence of an *atomic-acquire* event in that thread. This callback has the type signature **ompt_callback_mutex_acquire_t**.

A thread dispatches a registered **ompt_callback_mutex_acquired** callback for each occurrence of an *atomic-acquired* event in that thread. This callback has the type signature **ompt_callback_mutex_t**.

A thread dispatches a registered **ompt_callback_mutex_released** callback for each occurrence of an *atomic-release* event in that thread. This callback has the type signature **ompt_callback_mutex_t**. The callbacks occur in the task encountering the atomic construct. The callbacks should receive **ompt_mutex_atomic** as their *kind* argument if practical, but a less specific kind is acceptable.

Restrictions

The following restrictions apply to the **atomic** construct:

- At most one *memory-order-clause* may appear on the construct.
- If *atomic-clause* is **read** then *memory-order-clause* must not be **acq_rel** or **release**.
- If *atomic-clause* is **write** then *memory-order-clause* must not be **acq_rel** or **acquire**.
- If *atomic-clause* is **update** or not present then *memory-order-clause* must not be **acq_rel** or **acquire**.

C / C++

- All atomic accesses to the storage locations designated by *x* throughout the program are required to have a compatible type.

C / C++

Fortran

- All atomic accesses to the storage locations designated by *x* throughout the program are required to have the same type and type parameters.

Fortran

- OpenMP constructs may not be encountered during execution of an **atomic** region.

Cross References

- **critical** construct, see Section 2.18.2 on page 195.
- **barrier** construct, see Section 2.18.3 on page 198.
- **flush** construct, see Section 2.18.8 on page 215.
- **ordered** construct, see Section 2.18.9 on page 221.
- **reduction** clause, see Section 2.20.4.4 on page 272.
- lock routines, see Section 3.3 on page 344.
- Synchronization Hints, see Section 2.18.11 on page 229.
- **ompt_mutex_atomic**, see Section 4.1.3.4.14 on page 399.
- **ompt_callback_mutex_acquire_t**, see Section 4.1.4.2.13 on page 419.
- **ompt_callback_mutex_t**, see Section 4.1.4.2.14 on page 420.

2.18.8 flush Construct

Summary

The **flush** construct executes the OpenMP flush operation. This operation makes a thread's temporary view of memory consistent with memory and enforces an order on the memory operations of the variables explicitly specified or implied. See the memory model description in Section 1.4 on page 20 for more details. The **flush** construct is a stand-alone directive.

Syntax

C / C++

The syntax of the **flush** construct is as follows:

```
#pragma omp flush [memory-order-clause] [(list)] new-line
```

where *memory-order-clause* is one of the following:

acq_rel

release

acquire

C / C++

Fortran

The syntax of the **flush** construct is as follows:

```
!$omp flush [memory-order-clause] [(list)]
```

where *memory-order-clause* is one of the following:

acq_rel

release

acquire

Fortran

Binding

The binding thread set for a **flush** region is the encountering thread. Execution of a **flush** region affects the memory and the temporary view of memory of only the thread that executes the region. It does not affect the temporary view of other threads. Other threads must themselves execute a flush operation in order to be guaranteed to observe the effects of the encountering thread's flush operation

Description

A **flush** construct without a list, executed on a given thread, operates as if the whole thread-visible data state of the program, as defined by the base language, is flushed. A **flush** construct with a list applies the flush operation to the items in the list, and does not return until the operation is complete for all specified list items. An implementation may implement a **flush** with a list by ignoring the list, and treating it the same as a **flush** without a list.

If list items are specified on the **flush** construct, the flush operation is a non-synchronizable flush. Otherwise, the flush operation is a synchronizable flush for which the sync-set contains all variables that are flushed.

The flush operation's flush properties are determined according to *memory-order-clause*, if present:

- If *memory-order-clause* is not specified, the flush operation has the strong, write, and read flush properties and is both a release flush and an acquire flush if the flush does not have a list.
- If *memory-order-clause* is **acq_rel**, the flush operation has the write and read flush properties and is both a release flush and an acquire flush.
- If *memory-order-clause* is **release**, the flush operation has the write flush property and is a release flush.
- If *memory-order-clause* is **acquire**, the flush operation has the read flush property and is an acquire flush.

C / C++

If a pointer is present in the list, the pointer itself is flushed, not the memory block to which the pointer refers.

C / C++

Fortran

If the list item or a subobject of the list item has the **POINTER** attribute, the allocation or association status of the **POINTER** item is flushed, but the pointer target is not. If the list item is a Cray pointer, the pointer is flushed, but the object to which it points is not. If the list item is of type **C_PTR**, the variable is flushed, but the storage that corresponds to that address is not flushed. If the list item or the subobject of the list item has the **ALLOCATABLE** attribute and has an allocation status of allocated, the allocated variable is flushed; otherwise the allocation status is flushed.

Fortran

Note – Use of a **flush** construct with a list is extremely error prone and users are strongly discouraged from attempting it. The following examples illustrate the ordering properties of the flush operation. In the following incorrect pseudocode example, the programmer intends to prevent simultaneous execution of the protected section by the two threads, but the program does not work properly because it does not enforce the proper ordering of the operations on variables **a** and **b**. Any shared data accessed in the protected section is not guaranteed to be current or consistent during or after the protected section. The atomic notation in the pseudocode in the following two examples indicates that the accesses to **a** and **b** are **ATOMIC** writes and captures. Otherwise both examples would contain data races and automatically result in unspecified behavior.

Incorrect example:

```
                 a = b = 0

thread 1                          thread 2

atomic(b = 1)                     atomic(a = 1)
flush(b)                          flush(a)
flush(a)                          flush(b)
atomic(tmp = a)                   atomic(tmp = b)
if (tmp == 0) then                if (tmp == 0) then
   protected section                 protected section
end if                            end if
```

The problem with this example is that 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 1 or the flush of **a** on thread 2 to a position completely after the protected section (assuming that the protected section on thread 1 does not reference **b** and the protected section on thread 2 does not reference **a**). If either re-ordering happens, both threads can simultaneously execute the protected section.

The following pseudocode example correctly ensures that the protected section is executed by not more than one of the two threads at any one 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.

Correct example:

```
                    a = b = 0

        thread 1                          thread 2

atomic(b = 1)                      atomic(a = 1)
flush(a,b)                         flush(a,b)
atomic(tmp = a)                    atomic(tmp = b)
if (tmp == 0) then                 if (tmp == 0) then
   protected section                  protected section
end if                             end if
```

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.

Flush operations implied when executing an **atomic** region are described in Section 2.18.7.

A **flush** region arising from a **flush** directive without a list and without *memory-order-clause* present is implied at the following locations:

- During a barrier region.
- At entry to a **target update** region whose corresponding construct has a **to** clause.
- At exit from a **target update** region whose corresponding construct has a **from** clause.
- At entry to and exit from **parallel**, **critical**, **target** and **target data** regions.
- At entry to and exit from an **ordered** region, if a **threads** clause or a **depend** clause is present, or if no clauses are present.
- At entry to a **target enter data** region.
- At exit from a **target exit data** region.
- At exit from worksharing regions unless a **nowait** is present.
- During **omp_set_lock** and **omp_unset_lock** regions.
- During **omp_test_lock**, **omp_set_nest_lock**, **omp_unset_nest_lock** and **omp_test_nest_lock** regions, if the region causes the lock to be set or unset.
- Immediately before and immediately after every task scheduling point.

1 • During a **cancel** or **cancellation point** region, if the *cancel-var* ICV is *true* and
2 cancellation has been activated.

3 Note – A **flush** region is not implied at the following locations:

4 • At entry to worksharing regions.

5 • At entry to or exit from a **master** region.

6 ## Events

7 The *flush* event occurs in a thread encountering the **flush** construct.

8 ## Tool Callbacks

9 A thread dispatches a registered **ompt_callback_flush** callback for each occurrence of a
10 *flush* event in that thread. This callback has the type signature **ompt_callback_flush_t**.

11 ## Restrictions

12 The following restrictions apply to the **flush** construct:

13 • If *memory-order-clause* is **release**, **acquire**, or **acq_rel**, list items must not be specified
14 on the **flush** directive.

15 ## Cross References

16 • **ompt_callback_flush_t**, see Section 4.1.4.2.17 on page 424.

2.18.9 `ordered` Construct

Summary

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, or it is a stand-alone directive that specifies cross-iteration dependences in a doacross loop nest. The **`ordered`** construct sequentializes and orders the execution of **`ordered`** regions while allowing code outside the region to run in parallel.

Syntax

C / C++

The syntax of the **`ordered`** construct is as follows:

```
#pragma omp ordered [clause[ [,] clause] ]  new-line
    structured-block
```

where *clause* is one of the following:

`threads`

`simd`

or

```
#pragma omp ordered clause [[[,] clause] ... ]  new-line
```

where *clause* is one of the following:

`depend(source)`

`depend(sink :` *vec***`)`**

C / C++

Fortran

The syntax of the **ordered** construct is as follows:

```
!$omp ordered [clause[ [,] clause] ]
    structured-block
!$omp end ordered
```

where *clause* is one of the following:

threads

simd

or

```
!$omp ordered clause [[[,] clause] ... ]
```

where *clause* is one of the following:

depend(source)

depend(sink : *vec***)**

Fortran

If the **depend** clause is specified, the **ordered** construct is a stand-alone directive.

Binding

The binding thread set for an **ordered** region is the current team. An **ordered** region binds to the innermost enclosing **simd** or loop SIMD region if the **simd** clause is present, and otherwise it binds to the innermost enclosing loop region. **ordered** regions that bind to different regions execute independently of each other.

Description

If no clause is specified, the **ordered** construct behaves as if the **threads** clause had been specified. If the **threads** clause is specified, the threads in the team executing the loop region execute **ordered** regions sequentially in the order of the loop iterations. If any **depend** clauses are specified then those clauses specify the order in which the threads in the team execute **ordered** regions. If the **simd** clause is specified, the **ordered** regions encountered by any thread will use only a single SIMD lane to execute the **ordered** regions in the order of the loop iterations.

When the thread executing the first iteration of the loop encounters an **ordered** construct, it can enter the **ordered** region without waiting. When a thread executing any subsequent iteration encounters an **ordered** construct without a **depend** clause, it waits at the beginning of the **ordered** region until execution of all **ordered** regions belonging to all previous iterations has completed. When a thread executing any subsequent iteration encounters an **ordered** construct with one or more **depend(sink:***vec***)** clauses, it waits until its dependences on all valid iterations specified by the **depend** clauses are satisfied before it completes execution of the **ordered** region. A specific dependence is satisfied when a thread executing the corresponding iteration encounters an **ordered** construct with a **depend(source)** clause.

Events

The *ordered-acquire* event occurs in the thread encountering the **ordered** construct on entry to the ordered region before initiating synchronization for the region.

The *ordered-acquired* event occurs in the thread encountering the **ordered** construct after entering the region, but before executing the structured block of the **ordered** region.

The *ordered-release* event occurs in the thread encountering the **ordered** construct after completing any synchronization on exit from the **ordered** region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_mutex_acquire** callback for each occurrence of an *ordered-acquire* event in that thread. This callback has the type signature **ompt_callback_mutex_acquire_t**.

A thread dispatches a registered **ompt_callback_mutex_acquired** callback for each occurrence of an *ordered-acquired* event in that thread. This callback has the type signature **ompt_callback_mutex_t**.

A thread dispatches a registered **ompt_callback_mutex_released** callback for each occurrence of an *ordered-release* event in that thread. This callback has the type signature **ompt_callback_mutex_t**. The callbacks occur in the task encountering the ordered construct. The callbacks should receive **ompt_mutex_ordered** as their *kind* argument if practical, but a less specific kind is acceptable.

Restrictions

Restrictions to the **ordered** construct are as follows:

- At most one **threads** clause can appear on an **ordered** construct.
- At most one **simd** clause can appear on an **ordered** construct.
- At most one **depend(source)** clause can appear on an **ordered** construct.
- Either **depend(sink:***vec***)** clauses or **depend(source)** clauses may appear on an **ordered** construct, but not both.
- The loop or loop SIMD region to which an **ordered** region arising from an **ordered** construct without a **depend** clause binds must have an **ordered** clause without the parameter specified on the corresponding loop or loop SIMD directive.
- The loop region to which an **ordered** region arising from an **ordered** construct with any **depend** clauses binds must have an **ordered** clause with the parameter specified on the corresponding loop directive.
- An **ordered** construct with the **depend** clause specified must be closely nested inside a loop (or parallel loop) construct.
- An **ordered** region arising from an **ordered** construct with the **simd** clause specified must be closely nested inside a **simd** or loop SIMD region.
- An **ordered** region arising from an **ordered** construct with both the **simd** and **threads** clauses must be closely nested inside a loop SIMD region.
- During execution of an iteration of a loop or a loop nest within a loop, **simd**, or loop SIMD region, a thread must not execute more than one **ordered** region arising from an **ordered** construct without a **depend** clause.

C++

- A throw executed inside a **ordered** region must cause execution to resume within the same **ordered** region, and the same thread that threw the exception must catch it.

C++

Cross References

- loop construct, see Section 2.10.1 on page 78.
- **simd** construct, see Section 2.11.1 on page 96.
- parallel loop construct, see Section 2.16.1 on page 169.
- **depend** Clause, see Section 2.18.10 on page 225
- **ompt_mutex_ordered**, see Section 4.1.3.4.14 on page 399.

- **ompt_callback_mutex_acquire_t**, see Section 4.1.4.2.13 on page 419.
- **ompt_callback_mutex_t**, see Section 4.1.4.2.14 on page 420.

2.18.10 depend Clause

Summary

The **depend** clause enforces additional constraints on the scheduling of tasks or loop iterations. These constraints establish dependences only between sibling tasks or between loop iterations.

Syntax

The syntax of the **depend** clause is as follows:

depend(*dependence-type* **:** *locator-list* **[:***iterators-definition***])**

where *dependence-type* is one of the following:

in

out

inout

mutexinoutset

or

depend(*dependence-type***)**

where *dependence-type* is:

source

or

depend(*dependence-type* **:** *vec***)**

where *dependence-type* is:

sink

and where *vec* is the iteration vector, which has the form:

x_1 [$\pm$ d_1], x_2 [$\pm$ d_2], . . . , x_n [$\pm$ d_n]

where n is the value specified by the **ordered** clause in the loop directive, x_i denotes the loop iteration variable of the i-th nested loop associated with the loop directive, and d_i is a constant non-negative integer.

Description

Task dependences are derived from the *dependence-type* of a **depend** clause and its list items when *dependence-type* is **in**, **out**, **inout**, or **mutexinoutset**.

For the **in** *dependence-type*, if the storage location of at least one of the list items is the same as the storage location of a list item appearing in a **depend** clause with an **out**, **inout**, or **mutexinoutset** *dependence-type* on a construct from which a sibling task was previously generated, then the generated task will be a dependent task of that sibling task.

For the **out** and **inout** *dependence-types*, if the storage location of at least one of the list items is the same as the storage location of a list item appearing in a **depend** clause with an **in**, **out**, **inout**, or **mutexinoutset** *dependence-type* on a construct from which a sibling task was previously generated, then the generated task will be a dependent task of that sibling task.

For the **mutexinoutset** *dependence-type*, if the storage location of at least one of the list items is the same as the storage location of a list item appearing in a **depend** clause with an **in**, **out**, or **inout** *dependence-type* on a construct from which a sibling task was previously generated, then the generated task will be a dependent task of that sibling task.

If a list item appearing in a **depend** clause with a **mutexinoutset** *dependence-type* on a task-generating construct has the same storage location as a list item appearing in a **depend** clause with a **mutexinoutset** *dependence-type* on a different task generating construct, and both constructs generate sibling tasks, the sibling tasks will be mutually exclusive tasks.

The list items that appear in the **depend** clause may reference iterators defined by an *iterators-definition* appearing on the same clause.

Fortran

If a list item has the **ALLOCATABLE** attribute and its allocation status is unallocated, the behavior is unspecified. If a list item has the **POINTER** attribute and its association status is disassociated or undefined, the behavior is unspecified.

Fortran

The list items that appear in the **depend** clause may include array sections.

Note – The enforced task dependence establishes a synchronization of memory accesses performed by a dependent task with respect to accesses performed by the predecessor tasks. However, it is the responsibility of the programmer to synchronize properly with respect to other concurrent accesses that occur outside of those tasks.

The **source** *dependence-type* specifies the satisfaction of cross-iteration dependences that arise from the current iteration.

The **sink** *dependence-type* specifies a cross-iteration dependence, where the iteration vector *vec* indicates the iteration that satisfies the dependence.

If the iteration vector *vec* does not occur in the iteration space, the **depend** clause is ignored. If all **depend** clauses on an **ordered** construct are ignored then the construct is ignored.

Note – If the iteration vector *vec* does not indicate a lexicographically earlier iteration, it can cause a deadlock.

Events

The *task-dependences* event occurs in a thread encountering a tasking construct with a **depend** clause immediately after the *task-create* event for the new task.

The *task-dependence* event indicates an unfulfilled dependence for the generated task. This event occurs in a thread that observes the unfulfilled dependence before it is satisfied.

Tool Callbacks

A thread dispatches the **ompt_callback_task_dependences** callback for each occurrence of the *task-dependences* event to announce its dependences with respect to the list items in the **depend** clause. This callback has type signature **ompt_callback_task_dependences_t**.

A thread dispatches the **ompt_callback_task_dependence** callback for a *task-dependence* event to report a dependence between a predecessor task (*src_task_data*) and a dependent task (*sink_task_data*). This callback has type signature **ompt_callback_task_dependence_t**.

Restrictions

Restrictions to the **depend** clause are as follows:

- List items used in **depend** clauses of the same task or sibling tasks must indicate identical storage locations or disjoint storage locations.
- List items used in **depend** clauses cannot be zero-length array sections.

Fortran

- A common block name cannot appear in a **depend** clause.

Fortran

- For a *vec* element of **sink** *dependence-type* of the form $x_i + d_i$ or $x_i - d_i$ if the loop iteration variable x_i has an integral or pointer type, the expression $x_i + d_i$ or $x_i - d_i$ for any value of the loop iteration variable x_i that can encounter the **ordered** construct must be computable in the loop iteration variable's type without overflow.

C++

- For a *vec* element of **sink** *dependence-type* of the form $x_i + d_i$ or $x_i - d_i$ if the loop iteration variable x_i is of a random access iterator type other than pointer type, the expression $(x_i - lb_i) + d_i$ or $(x_i - lb_i) - d_i$ for any value of the loop iteration variable x_i that can encounter the **ordered** construct must be computable in the type that would be used by *std::distance* applied to variables of the type of x_i without overflow.

C++

C / C++

- A bit-field cannot appear in a **depend** clause.

C / C++

Cross References

- Array sections, see Section 2.5 on page 60.
- **task** construct, see Section 2.13.1 on page 110.
- **target enter data** construct, see Section 2.15.3 on page 134.
- **target exit data** construct, see Section 2.15.4 on page 137.
- **target** construct, see Section 2.15.5 on page 141.
- **target update** construct, see Section 2.15.6 on page 146.
- Task scheduling constraints, see Section 2.13.6 on page 125.
- **ordered** construct, see Section 2.18.9 on page 221.
- Iterators, see Section 2.6 on page 62.
- **ompt_callback_task_dependences_t**, see Section 4.1.4.2.8 on page 413.
- **ompt_callback_task_dependence_t**, see Section 4.1.4.2.9 on page 414.

2.18.11 Synchronization Hints

Hints about the expected dynamic behavior or suggested implementation can be provided by the programmer to locks (by using the **omp_init_lock_with_hint** or **omp_init_nest_lock_with_hint** functions to initialize the lock), and to **atomic** and **critical** directives by using the **hint** clause. The effect of a hint is implementation defined. The OpenMP implementation is free to ignore the hint since doing so cannot change program semantics.

The C/C++ header file (**omp.h**) and the Fortran include file (**omp_lib.h**) and/or Fortran 90 module file (**omp_lib**) define the valid hint constants. The valid constants must include the following, which can be extended with implementation-defined values:

C / C++

```
typedef enum omp_sync_hint_t {
  omp_sync_hint_none = 0,
  omp_lock_hint_none = omp_sync_hint_none,
  omp_sync_hint_uncontended = 1,
  omp_lock_hint_uncontended = omp_sync_hint_uncontended,
  omp_sync_hint_contended = 2,
  omp_lock_hint_contended = omp_sync_hint_contended,
  omp_sync_hint_nonspeculative = 4,
```

```
    omp_lock_hint_nonspeculative = omp_sync_hint_nonspeculative,
    omp_sync_hint_speculative = 8
    omp_lock_hint_speculative = omp_sync_hint_speculative
} omp_sync_hint_t;

typedef omp_sync_hint_t omp_lock_hint_t;
```

C / C++

Fortran

```
integer, parameter :: omp_lock_hint_kind = omp_sync_hint_kind

integer (kind=omp_sync_hint_kind), &
  parameter :: omp_sync_hint_none = 0
integer (kind=omp_lock_hint_kind), &
  parameter :: omp_lock_hint_none = omp_sync_hint_none
integer (kind=omp_sync_hint_kind), &
  parameter :: omp_sync_hint_uncontended = 1
integer (kind=omp_lock_hint_kind), &
  parameter :: omp_lock_hint_uncontended = &
                   omp_sync_hint_uncontended
integer (kind=omp_sync_hint_kind), &
  parameter :: omp_sync_hint_contended = 2
integer (kind=omp_lock_hint_kind), &
  parameter :: omp_lock_hint_contended = &
                 omp_sync_hint_contended
integer (kind=omp_sync_hint_kind), &
  parameter :: omp_sync_hint_nonspeculative = 4
integer (kind=omp_lock_hint_kind), &
  parameter :: omp_lock_hint_nonspeculative = &
                 omp_sync_hint_nonspeculative
integer (kind=omp_sync_hint_kind), &
  parameter :: omp_sync_hint_speculative = 8
integer (kind=omp_lock_hint_kind), &
  parameter :: omp_lock_hint_speculative = &
                 omp_sync_hint_speculative
```

Fortran

The hints can be combined by using the **+** or **|** operators in C/C++ or the **+** operator in Fortran. The effect of the combined hint is implementation defined and can be ignored by the implementation. Combining **omp_sync_hint_none** with any other hint is equivalent to specifying the other hint. The following restrictions apply to combined hints; violating these restrictions results in unspecified behavior:

- the hints **omp_sync_hint_uncontended** and **omp_sync_hint_contended** cannot be combined,
- the hints **omp_sync_hint_nonspeculative** and **omp_sync_hint_speculative** cannot be combined.

The rules for combining multiple values of **omp_sync_hint** apply equally to the corresponding values of **omp_lock_hint**, and expressions mixing the two types.

The intended meaning of hints is

- **omp_sync_hint_uncontended**: low contention is expected in this operation, that is, few threads are expected to be performing the operation simultaneously in a manner that requires synchronization.
- **omp_sync_hint_contended**: high contention is expected in this operation, that is, many threads are expected to be performing the operation simultaneously in a manner that requires synchronization.
- **omp_sync_hint_speculative**: the programmer suggests that the operation should be implemented using speculative techniques such as transactional memory.
- **omp_sync_hint_nonspeculative**: the programmer suggests that the operation should not be implemented using speculative techniques such as transactional memory.

Note – Future OpenMP specifications may add additional hints to the **omp_sync_hint_t** type and the **omp_sync_hint_kind** kind. Implementers are advised to add implementation-defined hints starting from the most significant bit of the **omp_sync_hint_t** type and **omp_sync_hint_kind** kind and to include the name of the implementation in the name of the added hint to avoid name conflicts with other OpenMP implementations.

The **omp_sync_hint_t** and **omp_lock_hint_t** enumeration types and the equivalent types in Fortran are synonyms for each other. The type **omp_lock_hint_t** has been deprecated.

Cross References

- **atomic** construct, see Section 2.18.7 on page 206
- **critical** construct, see Section 2.18.2 on page 195.
- **omp_init_lock_with_hint** and **omp_init_nest_lock_with_hint**, see Section 3.3.2 on page 347.

1
2.19 Cancellation Constructs

2
2.19.1 `cancel` Construct

3
Summary

4 The **`cancel`** construct activates cancellation of the innermost enclosing region of the type
5 specified. The **`cancel`** construct is a stand-alone directive.

6
Syntax

C / C++

7 The syntax of the **`cancel`** construct is as follows:

```
#pragma omp cancel construct-type-clause [ [,] if-clause] new-line
```

8 where *construct-type-clause* is one of the following:

9 **`parallel`**

10 **`sections`**

11 **`for`**

12 **`taskgroup`**

13 and *if-clause* is

14 **`if (`**[**`cancel :`**] *scalar-expression***`)`**

C / C++

Fortran

The syntax of the **cancel** construct is as follows:

```
!$omp cancel construct-type-clause [ [ , ] if-clause]
```

where *construct-type-clause* is one of the following:

parallel

sections

do

taskgroup

and *if-clause* is

if ([**cancel :**] *scalar-logical-expression***)**

Fortran

Binding

The binding thread set of the **cancel** region is the current team. The binding region of the **cancel** region is the innermost enclosing region of the type corresponding to the *construct-type-clause* specified in the directive (that is, the innermost **parallel**, **sections**, loop, or **taskgroup** region).

Description

The **cancel** construct activates cancellation of the binding region only if the *cancel-var* ICV is *true*, in which case the **cancel** construct causes the encountering task to continue execution at the end of the binding region if *construct-type-clause* is **parallel**, **for**, **do**, or **sections**. If the *cancel-var* ICV is *true* and *construct-type-clause* is **taskgroup**, the encountering task continues execution at the end of the current task region. If the *cancel-var* ICV is *false*, the **cancel** construct is ignored.

Threads check for active cancellation only at cancellation points that are implied at the following locations:

- **cancel** regions;
- **cancellation point** regions;
- **barrier** regions;
- implicit barriers regions.

When a thread reaches one of the above cancellation points and if the *cancel-var* ICV is *true*, then:

- If the thread is at a **cancel** or **cancellation point** region and *construct-type-clause* is **parallel**, **for**, **do**, or **sections**, the thread continues execution at the end of the canceled region if cancellation has been activated for the innermost enclosing region of the type specified.
- If the thread is at a **cancel** or **cancellation point** region and *construct-type-clause* is **taskgroup**, the encountering task checks for active cancellation of all of the *taskgroup sets* to which the encountering task belongs, and continues execution at the end of the current task region if cancellation has been activated for any of the *taskgroup sets*.
- If the encountering task is at a barrier region, the encountering task checks for active cancellation of the innermost enclosing **parallel** region. If cancellation has been activated, then the encountering task continues execution at the end of the canceled region.

Note – If one thread activates cancellation and another thread encounters a cancellation point, the order of execution between the two threads is non-deterministic. Whether the thread that encounters a cancellation point detects the activated cancellation depends on the underlying hardware and operating system.

When cancellation of tasks is activated through the **cancel taskgroup** construct, the tasks that belong to the *taskgroup set* of the innermost enclosing **taskgroup** region will be canceled. The task that encountered the **cancel taskgroup** construct continues execution at the end of its **task** region, which implies completion of that task. Any task that belongs to the innermost enclosing **taskgroup** and has already begun execution must run to completion or until a cancellation point is reached. Upon reaching a cancellation point and if cancellation is active, the task continues execution at the end of its **task** region, which implies the task's completion. Any task that belongs to the innermost enclosing **taskgroup** and that has not begun execution may be discarded, which implies its completion.

When cancellation is active for a **parallel**, **sections**, or loop region, each thread of the binding thread set resumes execution at the end of the canceled region if a cancellation point is encountered. If the canceled region is a **parallel** region, any tasks that have been created by a **task** construct and their descendent tasks are canceled according to the above **taskgroup** cancellation semantics. If the canceled region is a **sections**, or loop region, no task cancellation occurs.

C++

The usual C++ rules for object destruction are followed when cancellation is performed.

C++

Fortran

All private objects or subobjects with **ALLOCATABLE** attribute that are allocated inside the canceled construct are deallocated.

Fortran

If the canceled construct contains a **reduction** or **lastprivate** clause, the final value of the **reduction** or **lastprivate** variable is undefined.

When an **if** clause is present on a **cancel** construct and the **if** expression evaluates to *false*, the **cancel** construct does not activate cancellation. The cancellation point associated with the **cancel** construct is always encountered regardless of the value of the **if** expression.

Note – The programmer is responsible for releasing locks and other synchronization data structures that might cause a deadlock when a **cancel** construct is encountered and blocked threads cannot be canceled. The programmer is also responsible for ensuring proper synchronizations to avoid deadlocks that might arise from cancellation of OpenMP regions that contain OpenMP synchronization constructs.

Events

If a task encounters a **cancel** construct that will activate cancellation then a *cancel* event occurs.

A *discarded-task* event occurs for any discarded tasks.

Tool Callbacks

A thread dispatches a registered **ompt_callback_cancel** callback for each occurrence of a *cancel* event in that thread. The callback occurs in the context of the encountering task. The callback has type signature **ompt_callback_cancel_t**. The callback receives **ompt_cancel_activated** as its *flags* argument.

A thread dispatches a registered **ompt_callback_cancel** callback for each occurrence of a *discarded-task* event. The callback occurs in the context of the task that discards the task. The callback has type signature **ompt_callback_cancel_t**. The callback receives the *ompt_data_t* associated with the discarded task as its *task_data* argument. The callback receives **ompt_cancel_discarded_task** as its *flags* argument.

Restrictions

The restrictions to the **cancel** construct are as follows:

- The behavior for concurrent cancellation of a region and a region nested within it is unspecified.
- If *construct-type-clause* is **taskgroup**, the **cancel** construct must be closely nested inside a **task** construct and the **cancel** region must be closely nested inside a **taskgroup** region. If *construct-type-clause* is **sections**, the **cancel** construct must be closely nested inside a **sections** or **section** construct. Otherwise, the **cancel** construct must be closely nested inside an OpenMP construct that matches the type specified in *construct-type-clause* of the **cancel** construct.
- A worksharing construct that is canceled must not have a **nowait** clause.
- A loop construct that is canceled must not have an **ordered** clause.
- During execution of a construct that may be subject to cancellation, a thread must not encounter an orphaned cancellation point. That is, a cancellation point must only be encountered within that construct and must not be encountered elsewhere in its region.

Cross References

- *cancel-var* ICV, see Section 2.4.1 on page 49.
- **cancellation point** construct, see Section 2.19.2 on page 237.
- **if** Clause, see Section 2.17 on page 192.
- **omp_get_cancellation** routine, see Section 3.2.9 on page 308.
- **ompt_callback_cancel_t**, see Section 4.1.4.2.27 on page 437.
- **omp_cancel_flag_t** enumeration type, see Section 4.1.3.4.22 on page 403.

2.19.2 cancellation point Construct

Summary

The **cancellation point** construct introduces a user-defined cancellation point at which implicit or explicit tasks check if cancellation of the innermost enclosing region of the type specified has been activated. The **cancellation point** construct is a stand-alone directive.

Syntax

C / C++

The syntax of the **cancellation point** construct is as follows:

```
#pragma omp cancellation point construct-type-clause new-line
```

where *construct-type-clause* is one of the following:

parallel

sections

for

taskgroup

C / C++

Fortran

The syntax of the **cancellation point** construct is as follows:

```
!$omp cancellation point construct-type-clause
```

where *construct-type-clause* is one of the following:

parallel

sections

do

taskgroup

Fortran

Binding

The binding thread set of the **cancellation point** construct is the current team. The binding region of the **cancellation point** region is the innermost enclosing region of the type corresponding to the *construct-type-clause* specified in the directive (that is, the innermost **parallel**, **sections**, loop, or **taskgroup** region).

Description

This directive introduces a user-defined cancellation point at which an implicit or explicit task must check if cancellation of the innermost enclosing region of the type specified in the clause has been requested. This construct does not implement any synchronization between threads or tasks.

When an implicit or explicit task reaches a user-defined cancellation point and if the *cancel-var* ICV is *true*, then:

- If the *construct-type-clause* of the encountered **cancellation point** construct is **parallel**, **for**, **do**, or **sections**, the thread continues execution at the end of the canceled region if cancellation has been activated for the innermost enclosing region of the type specified.
- If the *construct-type-clause* of the encountered **cancellation point** construct is **taskgroup**, the encountering task checks for active cancellation of all *taskgroup sets* to which the encountering task belongs and continues execution at the end of the current task region if cancellation has been activated for any of them.

Events

The *cancellation* event occurs if a task encounters a cancellation point and detected the activation of cancellation.

Tool Callbacks

A thread dispatches a registered **ompt_callback_cancel** callback for each occurrence of a *cancellation* event in that thread. The callback occurs in the context of the encountering task. The callback has type signature **ompt_callback_cancel_t**. The callback receives **ompt_cancel_detected** as its *flags* argument.

Restrictions

- A **cancellation point** construct for which *construct-type-clause* is **taskgroup** must be closely nested inside a **task** construct, and the **cancellation point** region must be closely nested inside a **taskgroup** region. A **cancellation point** construct for which *construct-type-clause* is **sections** must be closely nested inside a **sections** or **section** construct. Otherwise, a **cancellation point** construct must be closely nested inside an OpenMP construct that matches the type specified in *construct-type-clause*.

Cross References

- *cancel-var* ICV, see Section 2.4.1 on page 49.
- **cancel** construct, see Section 2.19.1 on page 232.
- **omp_get_cancellation** routine, see Section 3.2.9 on page 308.
- **ompt_callback_cancel_t**, see Section 4.1.4.2.27 on page 437.

2.20 Data Environment

This section presents a directive and several clauses for controlling data environments.

2.20.1 Data-sharing Attribute Rules

This section describes how the data-sharing attributes of variables referenced in data enviroments are determined. The following two cases are described separately:

- Section 2.20.1.1 on page 240 describes the data-sharing attribute rules for variables referenced in a construct.
- Section 2.20.1.2 on page 243 describes the data-sharing attribute rules for variables referenced in a region, but outside any construct.

2.20.1.1 Data-sharing Attribute Rules for Variables Referenced in a Construct

The data-sharing attributes of variables that are referenced in a construct can be *predetermined*, *explicitly determined*, or *implicitly determined*, according to the rules outlined in this section.

Specifying a variable on a **firstprivate**, **lastprivate**, **linear**, **reduction**, or **copyprivate** clause of an enclosed construct causes an implicit reference to the variable in the enclosing construct. Specifying a variable on a **map** clause of an enclosed construct may cause an implicit reference to the variable in the enclosing construct. Such implicit references are also subject to the data-sharing attribute rules outlined in this section.

Certain variables and objects have *predetermined* data-sharing attributes as follows:

C / C++

- Variables appearing in **threadprivate** directives are threadprivate.
- Variables with automatic storage duration that are declared in a scope inside the construct are private.
- Objects with dynamic storage duration are shared.
- Static data members are shared.
- The loop iteration variable(s) in the associated *for-loop(s)* of a **for**, **parallel for**, **taskloop**, or **distribute** construct is (are) private.
- The loop iteration variable in the associated *for-loop* of a **simd** or **concurrent** construct with just one associated *for-loop* is linear with a *linear-step* that is the increment of the associated *for-loop*.
- The loop iteration variables in the associated *for-loops* of a **simd** construct with multiple associated *for-loops* are lastprivate.
- Variables with static storage duration that are declared in a scope inside the construct are shared.
- If an array section with a named pointer is a list item in a **map** clause on the **target** construct and the named pointer is a scalar variable that does not appear in a **map** clause on the construct, the named pointer is firstprivate.

C / C++

Fortran

1 • Variables and common blocks appearing in **threadprivate** directives are threadprivate.

2 • The loop iteration variable(s) in the associated *do-loop(s)* of a **do**, **parallel do**, **taskloop**,
3 or **distribute** construct is (are) private.

4 • The loop iteration variable in the associated *do-loop* of a **simd** or **concurrent** construct with
5 just one associated *do-loop* is linear with a *linear-step* that is the increment of the associated
6 *do-loop*.

7 • The loop iteration variables in the associated *do-loops* of a **simd** construct with multiple
8 associated *do-loops* are lastprivate.

9 • A loop iteration variable for a sequential loop in a **parallel** or task generating construct is
10 private in the innermost such construct that encloses the loop.

11 • Implied-do indices and **forall** indices are private.

12 • Cray pointees have the same the data-sharing attribute as the storage with which their Cray
13 pointers are associated.

14 • Assumed-size arrays are shared.

15 • An associate name preserves the association with the selector established at the **ASSOCIATE**
16 statement.

Fortran

17 Variables with predetermined data-sharing attributes may not be listed in data-sharing attribute
18 clauses, except for the cases listed below. For these exceptions only, listing a predetermined
19 variable in a data-sharing attribute clause is allowed and overrides the variable's predetermined
20 data-sharing attributes.

C / C++

21 • The loop iteration variable(s) in the associated *for-loop(s)* of a **for**, **parallel for**,
22 **taskloop**, or **distribute** construct may be listed in a **private** or **lastprivate** clause.

23 • The loop iteration variable in the associated *for-loop* of a **simd** construct with just one
24 associated *for-loop* may be listed in a **linear** clause with a *linear-step* that is the increment of
25 the associated *for-loop*.

26 • The loop iteration variables in the associated *for-loops* of a **simd** construct with multiple
27 associated *for-loops* may be listed in a **lastprivate** clause.

28 • Variables with **const**-qualified type having no mutable member may be listed in a
29 **firstprivate** clause, even if they are static data members.

C / C++

Fortran

- The loop iteration variable(s) in the associated *do-loop(s)* of a **do**, **parallel do**, **taskloop**, or **distribute** construct may be listed in a **private** or **lastprivate** clause.
- The loop iteration variable in the associated *d*o-loop of a **simd** construct with just one associated *do-loop* may be listed in a **linear** clause with a *linear-step* that is the increment of the associated loop.
- The loop iteration variables in the associated *do-loops* of a **simd** construct with multiple associated *do-loops* may be listed in a **lastprivate** clause.
- Variables used as loop iteration variables in sequential loops in a **parallel** or task generating construct may be listed in data-sharing clauses on the construct itself, and on enclosed constructs, subject to other restrictions.
- Assumed-size arrays may be listed in a **shared** clause.

Fortran

Additional restrictions on the variables that may appear in individual clauses are described with each clause in Section 2.20.3 on page 249.

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.

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 on the construct.

Rules for variables with *implicitly determined* data-sharing attributes are as follows:

- In a **parallel**, **teams**, or task generating construct, the data-sharing attributes of these variables are determined by the **default** clause, if present (see Section 2.20.3.1 on page 250).
- In a **parallel** construct, if no **default** clause is present, these variables are shared.
- For constructs other than task generating constructs, if no **default** clause is present, these variables reference the variables with the same names that exist in the enclosing context.
- In a **target** construct, variables that are not mapped after applying data-mapping attribute rules (see Section 2.20.6 on page 279) are firstprivate.

C++

- In an orphaned task generating construct, if no **default** clause is present, formal arguments passed by reference are firstprivate.

C++

Fortran

- In an orphaned task generating construct, if no **default** clause is present, dummy arguments are firstprivate.

Fortran

- In a task generating construct, if no **default** clause is present, a variable for which the data-sharing attribute is not determined by the rules above and that in the enclosing context is determined to be shared by all implicit tasks bound to the current team is shared.
- In a task generating construct, if no **default** clause is present, a variable for which the data-sharing attribute is not determined by the rules above is firstprivate.

Additional restrictions on the variables for which data-sharing attributes cannot be implicitly determined in a task generating construct are described in Section 2.20.3.4 on page 256.

2.20.1.2 Data-sharing Attribute Rules for Variables Referenced in a Region but not in a Construct

The data-sharing attributes of variables that are referenced in a region, but not in a construct, are determined as follows:

C / C++

- Variables with static storage duration that are declared in called routines in the region are shared.
- File-scope or namespace-scope variables referenced in called routines in the region are shared unless they appear in a **threadprivate** directive.
- Objects with dynamic storage duration are shared.
- Static data members are shared unless they appear in a **threadprivate** directive.
- In C++, formal arguments of called routines in the region that are passed by reference have the same data-sharing attributes as the associated actual arguments.
- Other variables declared in called routines in the region are private.

C / C++

Fortran

- Local variables declared in called routines in the region and that have the **save** attribute, or that are data initialized, are shared unless they appear in a **threadprivate** directive.
- Variables belonging to common blocks, or accessed by host or use association, and referenced in called routines in the region are shared unless they appear in a **threadprivate** directive.
- Dummy arguments of called routines in the region that have the **VALUE** attribute are private.
- Dummy arguments of called routines in the region that do not have the **VALUE** attribute are private if the associated actual argument is not shared.
- Dummy arguments of called routines in the region that do not have the **VALUE** attribute are shared if the actual argument is shared and it is a scalar variable, structure, an array that is not a pointer or assumed-shape array, or a simply contiguous array section. Otherwise, the data-sharing attribute of the dummy argument is implementation-defined if the associated actual argument is shared.
- Cray pointees have the same data-sharing attribute as the storage with which their Cray pointers are associated.
- Implied-do indices, **forall** indices, and other local variables declared in called routines in the region are private.

Fortran

2.20.2 threadprivate Directive

Summary

The **threadprivate** directive specifies that variables are replicated, with each thread having its own copy. The **threadprivate** directive is a declarative directive.

Syntax

C / C++

The syntax of the **threadprivate** directive is as follows:

```
#pragma omp threadprivate(list) new-line
```

where *list* is a comma-separated list of file-scope, namespace-scope, or static block-scope variables that do not have incomplete types.

C / C++

Fortran

The syntax of the **threadprivate** directive is as follows:

```
!$omp threadprivate(list)
```

where *list* is a comma-separated list of named variables and named common blocks. Common block names must appear between slashes.

Fortran

Description

Each copy of a threadprivate variable is initialized once, in the manner specified by the program, but at an unspecified point in the program prior to the first reference to that copy. The storage of all copies of a threadprivate variable is freed according to how static variables are handled in the base language, but at an unspecified point in the program.

A program in which a thread references another thread's copy of a threadprivate variable is non-conforming.

The content of a threadprivate variable can change across a task scheduling point if the executing thread switches to another task that modifies the variable. For more details on task scheduling, see Section 1.3 on page 18 and Section 2.13 on page 110.

In **parallel** regions, references by the master thread will be to the copy of the variable in the thread that encountered the **parallel** region.

During a sequential part references will be to the initial thread's copy of the variable. The values of data in the initial thread's copy of a threadprivate variable are guaranteed to persist between any two consecutive references to the variable in the program.

The values of data in the threadprivate variables of non-initial threads are guaranteed to persist between two consecutive active **parallel** regions only if all of the following conditions hold:

- Neither **parallel** region is nested inside another explicit **parallel** region.
- The number of threads used to execute both **parallel** regions is the same.
- The thread affinity policies used to execute both **parallel** regions are the same.
- The value of the *dyn-var* internal control variable in the enclosing task region is *false* at entry to both **parallel** regions.

If these conditions all hold, and if a threadprivate variable is referenced in both regions, then threads with the same thread number in their respective regions will reference the same copy of that variable.

C / C++

If the above conditions hold, the storage duration, lifetime, and value of a thread's copy of a threadprivate variable that does not appear in any **copyin** clause on the second region will be retained. Otherwise, the storage duration, lifetime, and value of a thread's copy of the variable in the second region is unspecified.

If the value of a variable referenced in an explicit initializer of a threadprivate variable is modified prior to the first reference to any instance of the threadprivate variable, then the behavior is unspecified.

C / C++

C++

The order in which any constructors for different threadprivate variables of class type are called is unspecified. The order in which any destructors for different threadprivate variables of class type are called is unspecified.

C++

Fortran

A variable is affected by a **copyin** clause if the variable appears in the **copyin** clause or it is in a common block that appears in the **copyin** clause.

If the above conditions hold, the definition, association, or allocation status of a thread's copy of a threadprivate variable or a variable in a threadprivate common block, that is not affected by any **copyin** clause that appears on the second region, will be retained. Otherwise, the definition and association status of a thread's copy of the variable in the second region are undefined, and the allocation status of an allocatable variable will be implementation defined.

If a threadprivate variable or a variable in a threadprivate common block is not affected by any **copyin** clause that appears on the first **parallel** region in which it is referenced, the variable or any subobject of the variable is initially defined or undefined according to the following rules:

- If it has the **ALLOCATABLE** attribute, each copy created will have an initial allocation status of unallocated.
- If it has the **POINTER** attribute:
  - if it has an initial association status of disassociated, either through explicit initialization or default initialization, each copy created will have an association status of disassociated;
  - otherwise, each copy created will have an association status of undefined.

- If it does not have either the **POINTER** or the **ALLOCATABLE** attribute:
  - if it is initially defined, either through explicit initialization or default initialization, each copy created is so defined;
  - otherwise, each copy created is undefined.

Fortran

Restrictions

The restrictions to the **threadprivate** directive are as follows:

- A threadprivate variable must not appear in any clause except the **copyin**, **copyprivate**, **schedule**, **num_threads**, **thread_limit**, and **if** clauses.
- A program in which an untied task accesses threadprivate storage is non-conforming.

C / C++

- A variable that is part of another variable (as an array or structure element) cannot appear in a **threadprivate** clause unless it is a static data member of a C++ class.
- A **threadprivate** directive for file-scope variables must appear outside any definition or declaration, and must lexically precede all references to any of the variables in its list.
- A **threadprivate** directive for namespace-scope variables must appear outside any definition or declaration other than the namespace definition itself, and must lexically precede all references to any of the variables in its list.
- Each variable in the list of a **threadprivate** directive at file, namespace, or class scope must refer to a variable declaration at file, namespace, or class scope that lexically precedes the directive.
- A **threadprivate** directive for static block-scope variables must appear in the scope of the variable and not in a nested scope. The directive must lexically precede all references to any of the variables in its list.
- Each variable in the list of a **threadprivate** directive in block scope must refer to a variable declaration in the same scope that lexically precedes the directive. The variable declaration must use the static storage-class specifier.
- If a variable is specified in a **threadprivate** directive in one translation unit, it must be specified in a **threadprivate** directive in every translation unit in which it is declared.
- The address of a threadprivate variable is not an address constant.

C / C++

C++

- A **threadprivate** directive for static class member variables must appear in the class definition, in the same scope in which the member variables are declared, and must lexically precede all references to any of the variables in its list.
- A threadprivate variable must not have an incomplete type or a reference type.
- A threadprivate variable with class type must have:
  - an accessible, unambiguous default constructor in case of default initialization without a given initializer;
  - an accessible, unambiguous constructor accepting the given argument in case of direct initialization;
  - an accessible, unambiguous copy constructor in case of copy initialization with an explicit initializer

C++

Fortran

- A variable that is part of another variable (as an array or structure element) cannot appear in a **threadprivate** clause.
- The **threadprivate** directive must appear in the declaration section of a scoping unit in which the common block or variable is declared. Although variables in common blocks can be accessed by use association or host association, common block names cannot. This means that a common block name specified in a **threadprivate** directive must be declared to be a common block in the same scoping unit in which the **threadprivate** directive appears.
- If a **threadprivate** directive specifying a common block name appears in one program unit, then such a directive must also appear in every other program unit that contains a **COMMON** statement specifying the same name. It must appear after the last such **COMMON** statement in the program unit.
- If a threadprivate variable or a threadprivate common block is declared with the **BIND** attribute, the corresponding C entities must also be specified in a **threadprivate** directive in the C program.
- A blank common block cannot appear in a **threadprivate** directive.
- A variable can only appear in a **threadprivate** directive in the scope in which it is declared. It must not be an element of a common block or appear in an **EQUIVALENCE** statement.
- A variable that appears in a **threadprivate** directive must be declared in the scope of a module or have the **SAVE** attribute, either explicitly or implicitly.

Fortran

Cross References

- *dyn-var* ICV, see Section 2.4 on page 49.
- Number of threads used to execute a **parallel** region, see Section 2.8.1 on page 71.
- **copyin** clause, see Section 2.20.5.1 on page 275.

2.20.3 Data-Sharing Attribute Clauses

Several constructs accept clauses that allow a user to control the data-sharing attributes of variables referenced in the construct. Data-sharing attribute clauses apply only to variables for which the names are visible in the construct on which the clause appears.

Not all of the clauses listed in this section are valid on all directives. The set of clauses that is valid on a particular directive is described with the directive.

Most of the clauses accept a comma-separated list of list items (see Section 2.1 on page 36). All list items appearing in a clause must be visible, according to the scoping rules of the base language. With the exception of the **default** clause, clauses may be repeated as needed. A list item that specifies a given variable may not appear in more than one clause on the same directive, except that a variable may be specified in both **firstprivate** and **lastprivate** clauses.

The reduction data-sharing clauses are explained in Section 2.20.4.

C++

If a variable referenced in a data-sharing attribute clause has a type derived from a template, and there are no other references to that variable in the program, then any behavior related to that variable is unspecified.

C++

Fortran

When a named common block appears in a **private**, **firstprivate**, **lastprivate**, or **shared** clause of a directive, none of its members may be declared in another data-sharing attribute clause in that directive. When individual members of a common block appear in a **private**, **firstprivate**, **lastprivate**, **reduction**, or **linear** clause of a directive, the storage of the specified variables is no longer Fortran associated with the storage of the common block itself.

Fortran

2.20.3.1 `default` Clause

Summary

The **`default`** clause explicitly determines the data-sharing attributes of variables that are referenced in a **`parallel`**, **`teams`**, or task generating construct and would otherwise be implicitly determined (see Section 2.20.1.1 on page 240).

Syntax

C / C++

The syntax of the **`default`** clause is as follows:

```
default(shared | none)
```

C / C++

Fortran

The syntax of the **`default`** clause is as follows:

```
default(private | firstprivate | shared | none)
```

Fortran

Description

The **`default(shared)`** clause causes all variables referenced in the construct that have implicitly determined data-sharing attributes to be shared.

Fortran

The **`default(firstprivate)`** clause causes all variables in the construct that have implicitly determined data-sharing attributes to be firstprivate.

The **`default(private)`** clause causes all variables referenced in the construct that have implicitly determined data-sharing attributes to be private.

Fortran

The **`default(none)`** clause requires that each variable that is referenced in the construct, and that does not have a predetermined data-sharing attribute, must have its data-sharing attribute explicitly determined by being listed in a data-sharing attribute clause.

1 **Restrictions**

2 The restrictions to the **default** clause are as follows:

3 • Only a single **default** clause may be specified on a **parallel**, **task**, **taskloop** or
4 **teams** directive.

5 ## 2.20.3.2 shared Clause

6 **Summary**

7 The **shared** clause declares one or more list items to be shared by tasks generated by a
8 **parallel**, **teams**, or task generating construct.

9 **Syntax**

10 The syntax of the **shared** clause is as follows:

```
shared(list)
```

11 **Description**

12 All references to a list item within a task refer to the storage area of the original variable at the point
13 the directive was encountered.

14 The programmer must ensure, by adding proper synchronization, that storage shared by an explicit
15 task region does not reach the end of its lifetime before the explicit task region completes its
16 execution.

Fortran

17 The association status of a shared pointer becomes undefined upon entry to and on exit from the
18 **parallel**, **teams**, or task generating construct if it is associated with a target or a subobject of a
19 target that is in a **private**, **firstprivate**, **lastprivate**, or **reduction** clause in the
20 construct.

Note – Passing a shared variable to a procedure may result in the use of temporary storage in place of the actual argument when the corresponding dummy argument does not have the **VALUE** attribute and its data-sharing attribute is implementation-defined as per the rules in Section 2.20.1.2 on page 243. These conditions effectively result in references to, and definitions of, the temporary storage during the procedure reference. Furthermore, the value of the shared variable is copied into the intervening temporary storage before the procedure reference when the dummy argument does not have the **INTENT(OUT)** attribute, and back out of the temporary storage into the shared variable when the dummy argument does not have the **INTENT(IN)** attribute. Any references to (or definitions of) the shared storage that is associated with the dummy argument by any other task must be synchronized with the procedure reference to avoid possible race conditions.

Fortran

Restrictions

The restrictions for the **shared** clause are as follows:

C

- A variable that is part of another variable (as an array or structure element) cannot appear in a shared clause.

C

C++

- A variable that is part of another variable (as an array or structure element) cannot appear in a **shared** clause except if the **shared** clause is associated with a construct within a class non-static member function and the variable is an accessible data member of the object for which the non-static member function is invoked.

C++

Fortran

- A variable that is part of another variable (as an array or structure element) cannot appear in a shared clause.

Fortran

2.20.3.3 private Clause

Summary

The **private** clause declares one or more list items to be private to a task or to a SIMD lane.

1 ## Syntax

2 The syntax of the private clause is as follows:

```
private(list)
```

3 ## Description

4 Each task that references a list item that appears in a **private** clause in any statement in the
5 construct receives a new list item. Each SIMD lane used in a **simd** construct that references a list
6 item that appears in a private clause in any statement in the construct receives a new list item. For
7 each reference to a list item that appears in a **private** clause on a **concurrent** construct, the
8 behavior will be as if a private copy of the list item is created for each logical loop iteration.
9 Language-specific attributes for new list items are derived from the corresponding original list item.
10 Inside the construct, all references to the original list item are replaced by references to the new list
11 item. In the rest of the region, it is unspecified whether references are to the new list item or the
12 original list item.

13 Therefore, if an attempt is made to reference the original item, its value after the region is also
14 unspecified. If a SIMD construct or a task does not reference a list item that appears in a **private**
15 clause, it is unspecified whether SIMD lanes or the task receive a new list item.

16 The value and/or allocation status of the original list item will change only:

17 • if accessed and modified via pointer,

18 • if possibly accessed in the region but outside of the construct,

19 • as a side effect of directives or clauses, or

Fortran

20 • if accessed and modified via construct association.

Fortran

C++

21 If the construct is contained in a member function, it is unspecified anywhere in the region if
22 accesses through the implicit **this** pointer refer to the new list item or the original list item.

C++

List items that appear in a **private**, **firstprivate**, or **reduction** clause in a **parallel** construct may also appear in a **private** clause in an enclosed **parallel**, worksharing, **task**, **taskloop**, **simd**, or **target** construct.

List items that appear in a **private** or **firstprivate** clause in a **task** or **taskloop** construct may also appear in a **private** clause in an enclosed **parallel**, **task**, **taskloop**, **simd**, or **target** construct.

List items that appear in a **private**, **firstprivate**, **lastprivate**, or **reduction** clause in a worksharing construct may also appear in a **private** clause in an enclosed **parallel**, **task**, **simd**, or **target** construct.

List items that appear in a **private** or **firstprivate** clause on a **concurrent** construct may also appear in a **private** or **firstprivate** clause in an enclosed **parallel** construct.

C / C++

A new list item of the same type, with automatic storage duration, is allocated for the construct. The storage and thus lifetime of these list items lasts until the block in which they are created exits. The size and alignment of the new list item are determined by the type of the variable. This allocation occurs once for each task generated by the construct and once for each SIMD lane used by the construct.

The new list item is initialized, or has an undefined initial value, as if it had been locally declared without an initializer.

C / C++

C++

If the type of a list item is a reference to a type *T* then the type will be considered to be *T* for all purposes of this clause.

The order in which any default constructors for different private variables of class type are called is unspecified. The order in which any destructors for different private variables of class type are called is unspecified.

C++

Fortran

If any statement of the construct references a list item, a new list item of the same type and type parameters is allocated. This allocation occurs once for each task generated by the construct and once for each SIMD lane used by the construct. The initial value of the new list item is undefined. The initial status of a private pointer is undefined.

For a list item or the subobject of a list item with the **ALLOCATABLE** attribute:

- if the allocation status is unallocated, the new list item or the subobject of the new list item will have an initial allocation status of unallocated.

- if the allocation status is allocated, the new list item or the subobject of the new list item will have an initial allocation status of allocated.
- If the new list item or the subobject of the new list item is an array, its bounds will be the same as those of the original list item or the subobject of the original list item.

A list item that appears in a **private** clause may be storage-associated with other variables when the **private** clause is encountered. Storage association may exist because of constructs such as **EQUIVALENCE** or **COMMON**. If *A* is a variable appearing in a **private** clause on a construct and *B* is a variable that is storage-associated with *A*, then:

- The contents, allocation, and association status of *B* are undefined on entry to the region.
- Any definition of *A*, or of its allocation or association status, causes the contents, allocation, and association status of *B* to become undefined.
- Any definition of *B*, or of its allocation or association status, causes the contents, allocation, and association status of *A* to become undefined.

A list item that appears in a **private** clause may be a selector of an **ASSOCIATE** construct. If the construct association is established prior to a **parallel** region, the association between the associate name and the original list item will be retained in the region.

Finalization of a list item of a finalizable type or subojects of a list item of a finalizable type occurs at the end of the region. The order in which any final subroutines for different variables of a finalizable type are called is unspecified.

Fortran

Restrictions

The restrictions to the **private** clause are as follows:

C

- A variable that is part of another variable (as an array or structure element) cannot appear in a **private** clause.

C

C++

- A variable that is part of another variable (as an array or structure element) cannot appear in a **private** clause except if the **private** clause is associated with a construct within a class non-static member function and the variable is an accessible data member of the object for which the non-static member function is invoked.

- A variable of class type (or array thereof) that appears in a **private** clause requires an accessible, unambiguous default constructor for the class type.

C++

C / C++

- A variable that appears in a **private** clause must not have a **const**-qualified type unless it is of class type with a **mutable** member. This restriction does not apply to the **firstprivate** clause.
- A variable that appears in a **private** clause must not have an incomplete type or be a reference to an incomplete type.

C / C++

Fortran

- A variable that is part of another variable (as an array or structure element) cannot appear in a **private** clause.
- A variable that appears in a **private** clause must either be definable, or an allocatable variable. This restriction does not apply to the **firstprivate** clause.
- Variables that appear in namelist statements, in variable format expressions, and in expressions for statement function definitions, may not appear in a **private** clause.
- Pointers with the **INTENT(IN)** attribute may not appear in a **private** clause. This restriction does not apply to the **firstprivate** clause.
- Assumed-size arrays may not appear in the **private** clause in a **target**, **teams**, or **distribute** construct.

Fortran

2.20.3.4 firstprivate Clause

Summary

The **firstprivate** clause declares one or more list items to be private to a task, and initializes each of them with the value that the corresponding original item has when the construct is encountered.

Syntax

The syntax of the **firstprivate** clause is as follows:

```
firstprivate(list)
```

1
Description

2 The **firstprivate** clause provides a superset of the functionality provided by the **private**
3 clause.

4 A list item that appears in a **firstprivate** clause is subject to the **private** clause semantics
5 described in Section 2.20.3.3 on page 252, except as noted. In addition, the new list item is
6 initialized from the original list item existing before the construct. The initialization of the new list
7 item is done once for each task that references the list item in any statement in the construct. The
8 initialization is done prior to the execution of the construct.

9 For a **firstprivate** clause on a **parallel**, **task**, **taskloop**, **target**, or **teams**
10 construct, the initial value of the new list item is the value of the original list item that exists
11 immediately prior to the construct in the task region where the construct is encountered unless
12 otherwise specified. For a **firstprivate** clause on a worksharing construct, the initial value of
13 the new list item for each implicit task of the threads that execute the worksharing construct is the
14 value of the original list item that exists in the implicit task immediately prior to the point in time
15 that the worksharing construct is encountered unless otherwise specified.

16 To avoid race conditions, concurrent updates of the original list item must be synchronized with the
17 read of the original list item that occurs as a result of the **firstprivate** clause.

18 If a list item appears in both **firstprivate** and **lastprivate** clauses, the update required
19 for **lastprivate** occurs after all the initializations for **firstprivate**.

C / C++

20 For variables of non-array type, the initialization occurs by copy assignment. For an array of
21 elements of non-array type, each element is initialized as if by assignment from an element of the
22 original array to the corresponding element of the new array.

C / C++

C++

23 For each variable of class type:

24 • If the **firstprivate** clause is not on a **target** construct then a copy constructor is invoked
25 to perform the initialization;

26 • If the **firstprivate** clause is on a **target** construct then it is unspecified how many copy
27 constructors, if any, are invoked.

28 If copy constructors are called, the order in which copy constructors for different variables of class
29 type are called is unspecified.

C++

Fortran

If the original list item does not have the **POINTER** attribute, initialization of the new list items occurs as if by intrinsic assignment, unless the original list item has the allocation status of unallocated, in which case the new list items will have the same status.

If the original list item has the **POINTER** attribute, the new list items receive the same association status of the original list item as if by pointer assignment.

Fortran

Restrictions

The restrictions to the **firstprivate** clause are as follows:

- A list item that is private within a **parallel** region must not appear in a **firstprivate** clause on a worksharing construct if any of the worksharing regions arising from the worksharing construct ever bind to any of the **parallel** regions arising from the **parallel** construct.
- A list item that is private within a **teams** region must not appear in a **firstprivate** clause on a **distribute** construct if any of the **distribute** regions arising from the **distribute** construct ever bind to any of the **teams** regions arising from the **teams** construct.
- A list item that appears in a **reduction** clause of a **parallel** construct must not appear in a **firstprivate** clause on a worksharing, **task**, or **taskloop** construct if any of the worksharing or task regions arising from the worksharing, **task**, or **taskloop** construct ever bind to any of the **parallel** regions arising from the **parallel** construct.
- A list item that appears in a **reduction** clause of a **teams** construct must not appear in a **firstprivate** clause on a **distribute** construct if any of the **distribute** regions arising from the **distribute** construct ever bind to any of the **teams** regions arising from the **teams** construct.
- A list item that appears in a **reduction** clause of a worksharing construct must not appear in a **firstprivate** clause in a **task** construct encountered during execution of any of the worksharing regions arising from the worksharing construct.

C++

- A variable of class type (or array thereof) that appears in a **firstprivate** clause requires an accessible, unambiguous copy constructor for the class type.

C++

C / C++

- A variable that appears in a **firstprivate** clause must not have an incomplete C/C++ type or be a reference to an incomplete type.
- If a list item in a **firstprivate** clause on a worksharing construct has a reference type then it must bind to the same object for all threads of the team.

C / C++

Fortran

- Variables that appear in namelist statements, in variable format expressions, or in expressions for statement function definitions, may not appear in a **firstprivate** clause.
- Assumed-size arrays may not appear in the **firstprivate** clause in a **target**, **teams**, or **distribute** construct.
- If the list item is a polymorphic variable with the **ALLOCATABLE** attribute, the behavior is unspecified.

Fortran

2.20.3.5 lastprivate Clause

Summary

The **lastprivate** clause declares one or more list items to be private to an implicit task or to a SIMD lane, and causes the corresponding original list item to be updated after the end of the region.

Syntax

The syntax of the **lastprivate** clause is as follows:

lastprivate([*lastprivate-modifier* **:**] *list***)**

where *lastprivate-modifier* is:

conditional

Description

The **lastprivate** clause provides a superset of the functionality provided by the **private** clause.

A list item that appears in a **lastprivate** clause is subject to the **private** clause semantics described in Section 2.20.3.3 on page 252. In addition, when a **lastprivate** clause without the **conditional** modifier appears on a directive, the value of each new list item from the sequentially last iteration of the associated loops, or the lexically last **section** construct, is assigned to the original list item. When the **conditional** modifier appears on the clause, if an assignment to a list item is encountered in the construct then the original list item is assigned the value that is assigned to the new list item in the sequentially last iteration or lexically last section in which such an assignment is encountered.

C / C++

For an array of elements of non-array type, each element is assigned to the corresponding element of the original array.

C / C++

Fortran

If the original list item does not have the **POINTER** attribute, its update occurs as if by intrinsic assignment.

If the original list item has the **POINTER** attribute, its update occurs as if by pointer assignment.

Fortran

When the **conditional** modifier does not appear on the **lastprivate** clause, list items that are not assigned a value by the sequentially last iteration of the loops, or by the lexically last **section** construct, have unspecified values after the construct. Unassigned subcomponents also have unspecified values after the construct.

If the **lastprivate** clause is used on a construct to which neither the **nowait** nor the **nogroup** clauses are applied, the original list item becomes defined at the end of the construct. To avoid race conditions, concurrent reads or updates of the original list item must be synchronized with the update of the original list item that occurs as a result of the **lastprivate** clause.

Otherwise, If the **lastprivate** clause is used on a construct to which the **nowait** or the **nogroup** clauses are applied, accesses to the original list item may create a data race. To avoid this, if an assignment to the original list item occurs then synchronization must be inserted to ensure that the assignment completes and the original list item is flushed to memory.

If a list item appears in both **firstprivate** and **lastprivate** clauses, the update required for **lastprivate** occurs after all initializations for **firstprivate**.

Restrictions

The restrictions to the **lastprivate** clause are as follows:

- A list item that is private within a **parallel** region, or that appears in the **reduction** clause of a **parallel** construct, must not appear in a **lastprivate** clause on a worksharing construct if any of the corresponding worksharing regions ever binds to any of the corresponding **parallel** regions.
- If a list item that appears in a **lastprivate** clause with the **conditional** modifier is modified in the region by an assignment outside the construct or not to the list item then the value assigned to the original list item is unspecified.
- A list item that appears in a **lastprivate** clause with the **conditional** modifier must be a scalar variable.

C++

- A variable of class type (or array thereof) that appears in a **lastprivate** clause requires an accessible, unambiguous default constructor for the class type, unless the list item is also specified in a **firstprivate** clause.
- A variable of class type (or array thereof) that appears in a **lastprivate** clause requires an accessible, unambiguous copy assignment operator for the class type. The order in which copy assignment operators for different variables of class type are called is unspecified.

C++

C / C++

- A variable that appears in a **lastprivate** clause must not have a **const**-qualified type unless it is of class type with a **mutable** member.
- A variable that appears in a **lastprivate** clause must not have an incomplete C/C++ type or be a reference to an incomplete type.
- If a list item in a **lastprivate** clause on a worksharing construct has a reference type then it must bind to the same object for all threads of the team.

C / C++

Fortran

- A variable that appears in a **lastprivate** clause must be definable.
- If the original list item has the **ALLOCATABLE** attribute, the corresponding list item whose value is assigned to the original list item must have an allocation status of allocated upon exit from the sequentially last iteration or lexically last **section** construct.
- Variables that appear in namelist statements, in variable format expressions, or in expressions for statement function definitions, may not appear in a **lastprivate** clause.

- If the list item is a polymorphic variable with the **ALLOCATABLE** attribute, the behavior is unspecified.

Fortran

2.20.3.6 `linear` Clause

Summary

The **linear** clause declares one or more list items to be private to a SIMD lane and to have a linear relationship with respect to the iteration space of a loop.

Syntax

C

The syntax of the **linear** clause is as follows:

linear (*linear-list[* **:** *linear-step]***)**

where *linear-list* is one of the following

list

modifier **(***list***)**

where *modifier* is one of the following:

val

C

C++

The syntax of the **linear** clause is as follows:

```
linear(linear-list[ : linear-step])
```

where *linear-list* is one of the following

list

modifier **(***list***)**

where *modifier* is one of the following:

ref

val

uval

C++

Fortran

The syntax of the **linear** clause is as follows:

```
linear(linear-list[ : linear-step])
```

where *linear-list* is one of the following

list

modifier **(***list***)**

where *modifier* is one of the following:

ref

val

uval

Fortran

Description

The **linear** clause provides a superset of the functionality provided by the **private** clause. A list item that appears in a **linear** clause is subject to the **private** clause semantics described in Section 2.20.3.3 on page 252 except as noted. If *linear-step* is not specified, it is assumed to be 1.

When a **linear** clause is specified on a construct, the value of the new list item on each iteration of the associated loop(s) corresponds to the value of the original list item before entering the construct plus the logical number of the iteration times *linear-step*. The value corresponding to the sequentially last iteration of the associated loop(s) is assigned to the original list item.

When a **linear** clause is specified on a declarative directive, all list items must be formal parameters (or, in Fortran, dummy arguments) of a function that will be invoked concurrently on each SIMD lane. If no *modifier* is specified or the **val** or **uval** modifier is specified, the value of each list item on each lane corresponds to the value of the list item upon entry to the function plus the logical number of the lane times *linear-step*. If the **uval** modifier is specified, each invocation uses the same storage location for each SIMD lane; this storage location is updated with the final value of the logically last lane. If the **ref** modifier is specified, the storage location of each list item on each lane corresponds to an array at the storage location upon entry to the function indexed by the logical number of the lane times *linear-step*.

Restrictions

- The *linear-step* expression must be invariant during the execution of the region associated with the construct. Otherwise, the execution results in unspecified behavior.
- A *list-item* cannot appear in more than one **linear** clause.
- A *list-item* that appears in a **linear** clause cannot appear in any other data-sharing attribute clause.

C

- A *list-item* that appears in a **linear** clause must be of integral or pointer type.

C

C++

- A *list-item* that appears in a **linear** clause without the **ref** modifier must be of integral or pointer type, or must be a reference to an integral or pointer type.
- The **ref** or **uval** modifier can only be used if the *list-item* is of a reference type.
- If a list item in a **linear** clause on a worksharing construct has a reference type then it must bind to the same object for all threads of the team.

- If the list item is of a reference type and the **`ref`** modifier is not specified and if any write to the list item occurs before any read of the list item then the result is unspecified.

C++

Fortran

- A *list-item* that appears in a **`linear`** clause without the **`ref`** modifier must be of type **`integer`**.
- The **`ref`** or **`uval`** modifier can only be used if the *list-item* is a dummy argument without the **`VALUE`** attribute.
- Variables that have the **`POINTER`** attribute and Cray pointers may not appear in a linear clause.
- The list item with the **`ALLOCATABLE`** attribute in the sequentially last iteration must have an allocation status of allocated upon exit from that iteration.
- If the list item is a dummy argument without the **`VALUE`** attribute and the **`ref`** modifier is not specified and if any write to the list item occurs before any read of the list item then the result is unspecified.
- A common block name cannot appear in a **`linear`** clause.

Fortran

2.20.4 Reduction Clauses

The reduction clauses can be used to perform some forms of recurrence calculations (involving mathematically associative and commutative operators) in parallel.

Reduction clauses include reduction scoping clauses and reduction participating clauses. Reduction scoping clauses define the region in which a reduction is computed. Reduction participating clauses define the participants in the reduction.

Reduction clauses specify a *reduction-identifier* and one or more list items.

2.20.4.1 Properties Common To All Reduction Clauses

Syntax

The syntax of a *reduction-identifier* is defined as follows:

C

A *reduction-identifier* is either an *identifier* or one of the following operators: **+**, **-**, *****, **&**, **|**, **^**, **&&** and **||**

C

C++

A *reduction-identifier* is either an *id-expression* or one of the following operators: **+**, **-**, *****, **&**, **|**, **^**, **&&** and **||**

C++

Fortran

A *reduction-identifier* is either a base language identifier, or a user-defined operator, or one of the following operators: **+**, **-**, *****, **.and.**, **.or.**, **.eqv.**, **.neqv.**, or one of the following intrinsic procedure names: **max**, **min**, **iand**, **ior**, **ieor**.

Fortran

C / C++

Table 2.8 lists each *reduction-identifier* that is implicitly declared at every scope for arithmetic types and its semantic initializer value. The actual initializer value is that value as expressed in the data type of the reduction list item.

TABLE 2.8: Implicitly Declared C/C++ *reduction-identifiers*

<table>
<tr><th>Identifier</th><th>Initializer</th><th>Combiner</th></tr>
<tr><td><code>+</code></td><td><code>omp_priv = 0</code></td><td><code>omp_out += omp_in</code></td></tr>
<tr><td><code>*</code></td><td><code>omp_priv = 1</code></td><td><code>omp_out *= omp_in</code></td></tr>
<tr><td><code>-</code></td><td><code>omp_priv = 0</code></td><td><code>omp_out += omp_in</code></td></tr>
<tr><td><code>&</code></td><td><code>omp_priv = ~0</code></td><td><code>omp_out &= omp_in</code></td></tr>
<tr><td><code>|</code></td><td><code>omp_priv = 0</code></td><td><code>omp_out |= omp_in</code></td></tr>
<tr><td><code>^</code></td><td><code>omp_priv = 0</code></td><td><code>omp_out ^= omp_in</code></td></tr>
<tr><td><code>&&</code></td><td><code>omp_priv = 1</code></td><td><code>omp_out = omp_in && omp_out</code></td></tr>
<tr><td><code>||</code></td><td><code>omp_priv = 0</code></td><td><code>omp_out = omp_in || omp_out</code></td></tr>
<tr><td><code>max</code></td><td><code>omp_priv =</code> Least representable number in the reduction list item type</td><td><code>omp_out = omp_in > omp_out ? omp_in : omp_out</code></td></tr>
<tr><td><code>min</code></td><td><code>omp_priv =</code> Largest representable number in the reduction list item type</td><td><code>omp_out = omp_in < omp_out ? omp_in : omp_out</code></td></tr>
</table>

C / C++

Fortran

Table 2.9 lists each *reduction-identifier* that is implicitly declared for numeric and logical types and its semantic initializer value. The actual initializer value is that value as expressed in the data type of the reduction list item.

TABLE 2.9: Implicitly Declared Fortran *reduction-identifiers*

Identifier	Initializer	Combiner
`+`	`omp_priv = 0`	`omp_out = omp_in + omp_out`
`*`	`omp_priv = 1`	`omp_out = omp_in * omp_out`
`-`	`omp_priv = 0`	`omp_out = omp_in + omp_out`
`.and.`	`omp_priv = .true.`	`omp_out = omp_in .and. omp_out`

table continued on next page

table continued from previous page

Identifier	Initializer	Combiner
`.or.`	`omp_priv = .false.`	`omp_out = omp_in .or. omp_out`
`.eqv.`	`omp_priv = .true.`	`omp_out = omp_in .eqv. omp_out`
`.neqv.`	`omp_priv = .false.`	`omp_out = omp_in .neqv. omp_out`
`max`	`omp_priv =` *Least representable number in the reduction list item type*	`omp_out = max(omp_in, omp_out)`
`min`	`omp_priv =` *Largest representable number in the reduction list item type*	`omp_out = min(omp_in, omp_out)`
`iand`	`omp_priv =` *All bits on*	`omp_out = iand(omp_in, omp_out)`
`ior`	`omp_priv = 0`	`omp_out = ior(omp_in, omp_out)`
`ieor`	`omp_priv = 0`	`omp_out = ieor(omp_in, omp_out)`

Fortran

In the above tables, **omp_in** and **omp_out** correspond to two identifiers that refer to storage of the type of the list item. **omp_out** holds the final value of the combiner operation.

Any *reduction-identifier* that is defined with the **declare reduction** directive is also valid. In that case, the initializer and combiner of the *reduction-identifier* are specified by the *initializer-clause* and the *combiner* in the **declare reduction** directive.

Description

A reduction clause specifies a *reduction-identifier* and one or more list items.

The *reduction-identifier* specified in a reduction clause must match a previously declared *reduction-identifier* of the same name and type for each of the list items. This match is done by means of a name lookup in the base language.

The list items that appear in a reduction clause may include array sections.

C++

If the type is a derived class, then any *reduction-identifier* that matches its base classes is also a match, if there is no specific match for the type.

If the *reduction-identifier* is not an *id-expression*, then it is implicitly converted to one by prepending the keyword operator (for example, **+** becomes *operator***+**).

If the *reduction-identifier* is qualified then a qualified name lookup is used to find the declaration.

If the *reduction-identifier* is unqualified then an *argument-dependent name lookup* must be performed using the type of each list item.

C++

If the list item is an array or array section, it will be treated as if a reduction clause would be applied to each separate element of the array section.

Any copies associated with the reduction are initialized with the intializer value of the *reduction-identifier*.

Any copies are combined using the combiner associated with the *reduction-identifier*.

Restrictions

The restrictions common to reduction clauses are as follows:

- Any number of reduction clauses can be specified on the directive, but a list item (or any array element in an array section) can appear only once in reduction clauses for that directive.
- For a *reduction-identifier* declared with the **declare reduction** construct, the directive must appear before its use in a reduction clause.
- If a list item is an array section, its base expression must be a base language identifier.
- If a list item is an array section, it must specify contiguous storage and it cannot be a zero-length array section.
- If a list item is an array section, accesses to the elements of the array outside the specified array section result in unspecified behavior.

C / C++

- The type of a list item that appears in a reduction clause must be valid for the *reduction-identifier*. For a **max** or **min** reduction in C, the type of the list item must be an allowed arithmetic data type: **char**, **int**, **float**, **double**, or **_Bool**, possibly modified with **long**, **short**, **signed**, or **unsigned**. For a **max** or **min** reduction in C++, the type of the list item must be an allowed arithmetic data type: **char**, **wchar_t**, **int**, **float**, **double**, or **bool**, possibly modified with **long**, **short**, **signed**, or **unsigned**.

- A list item that appears in a reduction clause must not be **const**-qualified.
- The *reduction-identifier* for any list item must be unambiguous and accessible.

C / C++

Fortran

- The type and the rank of a list item that appears in a reduction clause must be valid for the *combiner* and *initializer*.
- A list item that appears in a reduction clause must be definable.
- A procedure pointer may not appear in a reduction clause.
- A pointer with the **INTENT(IN)** attribute may not appear in the reduction clause.
- An original list item with the **POINTER** attribute or any pointer component of an original list item that is referenced in the *combiner* must be associated at entry to the construct that contains the reduction clause. Additionally, the list item or the pointer component of the list item must not be deallocated, allocated, or pointer assigned within the region.
- An original list item with the **ALLOCATABLE** attribute or any allocatable component of an original list item that is referenced in the *combiner* must be in the allocated state at entry to the construct that contains the reduction clause. Additionally, the list item or the allocatable component of the list item must be neither deallocated nor allocated within the region.
- If the *reduction-identifier* is defined in a **declare reduction** directive, the **declare reduction** directive must be in the same subprogram, or accessible by host or use association.
- If the *reduction-identifier* is a user-defined operator, the same explicit interface for that operator must be accessible as at the **declare reduction** directive.
- If the *reduction-identifier* is defined in a **declare reduction** directive, any subroutine or function referenced in the initializer clause or combiner expression must be an intrinsic function, or must have an explicit interface where the same explicit interface is accessible as at the **declare reduction** directive.

Fortran

2.20.4.2 Reduction Scoping Clauses

Reduction scoping clauses define the region in which a reduction is computed by tasks or SIMD lanes. All properties common to all reduction clauses, which are defined in Section 2.20.4.1, apply to reduction scoping clauses.

The number of copies created for each list item and the time at which those copies are initialized are determined by the particular reduction scoping clause that appears on the construct.

The time at which the original list item contains the result of the reduction is determined by the particular reduction scoping clause.

Fortran

If the original list item has the **POINTER** attribute, copies of the list item are associated with private targets.

Fortran

If the list item is an array section, the elements of any copy of the array section will be allocated contiguously.

The location in the OpenMP program at which values are combined and the order in which values are combined are unspecified. Therefore, when comparing sequential and parallel runs, or when comparing one parallel run to another (even if the number of threads used is the same), there is no guarantee that bit-identical results will be obtained or that side effects (such as floating-point exceptions) will be identical or take place at the same location in the OpenMP program.

To avoid race conditions, concurrent reads or updates of the original list item must be synchronized with the update of the original list item that occurs as a result of the reduction computation.

2.20.4.3 Reduction Participating Clauses

A reduction participating clause specifies a task or a SIMD lane as a participant in a reduction defined by a reduction scoping clause. All properties common to all reduction clauses, which are defined in Section 2.20.4.1, apply to reduction participating clauses.

Accesses to the original list item may be replaced by accesses to copies of the original list item created by a region associated with a construct with a reduction scoping clause.

In any case, the final value of the reduction must be determined as if all tasks or SIMD lanes that participate in the reduction are executed sequentially in some arbitrary order.

2.20.4.4 `reduction` Clause

Summary

The **`reduction`** clause specifies a *reduction-identifier* and one or more list items. For each list item, a private copy is created in each implicit task or SIMD lane and is initialized with the initializer value of the *reduction-identifier*. After the end of the region, the original list item is updated with the values of the private copies using the combiner associated with the *reduction-identifier*.

Syntax

```
reduction(reduction-identifier : list)
```

Where *reduction-identifier* is defined in Section 2.20.4.1.

Description

The **`reduction`** clause is a reduction scoping clause and a reduction participating clause, as described in Sections 2.20.4.2 and 2.20.4.3.

For **`parallel`** and worksharing constructs, a private copy of each list item is created, one for each implicit task, as if the **`private`** clause had been used. For the **`simd`** construct, a private copy of each list item is created, one for each SIMD lane as if the **`private`** clause had been used. For the **`taskloop`** construct, private copies are created according to the rules of the reduction scoping clauses. For the **`target`** construct, a private copy of each list item is created and initialized for the initial task, and the list item present in the data environment of its associated target task is the original list item. For the **`teams`** construct, a private copy of each list item is created and initialized, one for each team in the league as if the **`private`** clause had been used. At the end of the region for which the **`reduction`** clause was specified, the original list item is updated by combining its original value with the final value of each of the private copies, using the combiner of the specified *reduction-identifier*. For the **`concurrent`** construct, the behavior will be as if a private copy of each list item is created for each loop iteration. At the end of the region for which the **`reduction`** clause was specified, the original list item is updated by combining its original value with the final value of each of the private copies, using the combiner of the specified *reduction-identifier*.

If **`nowait`** is not used, the reduction computation will be complete at the end of the construct; however, if the reduction clause is used on a construct to which **`nowait`** is also applied, accesses to the original list item will create a race and, thus, have unspecified effect unless synchronization ensures that they occur after all threads have executed all of their iterations or **`section`** constructs, and the reduction computation has completed and stored the computed value of that list item. This can most simply be ensured through a barrier synchronization.

Restrictions

The restrictions to the **reduction** clause are as follows:

- All the common restrictions to all reduction clauses, which are listed in Section 2.20.4.1, apply to this clause.
- A list item that appears in a **reduction** clause of a worksharing construct must be shared in the **parallel** regions to which any of the worksharing regions arising from the worksharing construct bind.
- A list item that appears in a **reduction** clause of the innermost enclosing worksharing or **parallel** construct may not be accessed in an explicit task generated by a construct for which an **in_reduction** clause over the same list item does not appear.

C / C++

- If a list item in a **reduction** clause on a worksharing construct has a reference type then it must bind to the same object for all threads of the team.

C / C++

2.20.4.5 task_reduction Clause

Summary

The **task_reduction** clause specifies a reduction among tasks.

Syntax

```
task_reduction(reduction-identifier : list)
```

Where *reduction-identifier* is defined in Section 2.20.4.1.

Description

The **task_reduction** clause is a reduction scoping clause, as described in 2.20.4.2.

For each list item, the number of copies is unspecified. Any copies associated with the reduction are initialized before they are accessed by the tasks participating in the reduction. After the end of the region, the original list item contains the result of the reduction.

Restrictions

The restrictions to the **task_reduction** clause are as follows:

- All the common restrictions to all reduction clauses, which are listed in Section 2.20.4.1, apply to this clause.

2.20.4.6 `in_reduction` Clause

Summary

The **`in_reduction`** clause specifies that a task participates in a reduction.

Syntax

```
in_reduction(reduction-identifier : list)
```

Where *reduction-identifier* is defined in Section 2.20.4.1

Description

The **`in_reduction`** clause is a reduction participating clause, as described in Section 2.20.4.3, that defines a task to be a participant in the task reduction defined by an enclosing **`taskgroup`** region for the list item. For the **`task`** construct, the generated task becomes the participating task. For the **`target`** construct, the target task becomes the participating task.

For each list item, a private copy may be created as if the **`private`** clause had been used.

At the end of the task region, if a private copy was created its value is combined with a copy created by a reduction scoping clause or with the original list item.

Restrictions

The restrictions to the **`in_reduction`** clause are as follows:

- All the common restrictions to all reduction clauses, which are listed in Section 2.20.4.1, apply to this clause.
- A list item that appears in an **`in_reduction`** clause of a task-generating construct must appear in a **`task_reduction`** clause of a construct associated with a taskgroup region that includes the participating task in its *taskgroup set*. The construct associated with the innermost region that meets this condition must specify the same *reduction-identifier* as the **`in_reduction`** clause.

2.20.5 Data Copying Clauses

This section describes the **copyin** clause (allowed on the **parallel** directive and combined parallel worksharing directives) and the **copyprivate** clause (allowed on the **single** directive).

These clauses support the copying of data values from private or threadprivate variables on one implicit task or thread to the corresponding variables on other implicit tasks or threads in the team.

The clauses accept a comma-separated list of list items (see Section 2.1 on page 36). All list items appearing in a clause must be visible, according to the scoping rules of the base language. Clauses may be repeated as needed, but a list item that specifies a given variable may not appear in more than one clause on the same directive.

Fortran

An associate name preserves the association with the selector established at the **ASSOCIATE** statement. A list item that appears in a data copying clause may be a selector of an **ASSOCIATE** construct. If the construct association is established prior to a parallel region, the association between the associate name and the original list item will be retained in the region.

Fortran

2.20.5.1 copyin Clause

Summary

The **copyin** clause provides a mechanism to copy the value of the master thread's threadprivate variable to the threadprivate variable of each other member of the team executing the **parallel** region.

Syntax

The syntax of the **copyin** clause is as follows:

```
copyin(list)
```

Description

C / C++

The copy is done after the team is formed and prior to the start of execution of the associated structured block. For variables of non-array type, the copy occurs by copy assignment. For an array of elements of non-array type, each element is copied as if by assignment from an element of the master thread's array to the corresponding element of the other thread's array.

C / C++

C++

For class types, the copy assignment operator is invoked. The order in which copy assignment operators for different variables of class type are called is unspecified.

C++

Fortran

The copy is done, as if by assignment, after the team is formed and prior to the start of execution of the associated structured block.

On entry to any **parallel** region, each thread's copy of a variable that is affected by a **copyin** clause for the **parallel** region will acquire the allocation, association, and definition status of the master thread's copy, according to the following rules:

- If the original list item has the **POINTER** attribute, each copy receives the same association status of the master thread's copy as if by pointer assignment.
- If the original list item does not have the **POINTER** attribute, each copy becomes defined with the value of the master thread's copy as if by intrinsic assignment, unless it has the allocation status of unallocated, in which case each copy will have the same status.

Fortran

Restrictions

The restrictions to the **copyin** clause are as follows:

C / C++

- A list item that appears in a **copyin** clause must be threadprivate.
- A variable of class type (or array thereof) that appears in a **copyin** clause requires an accessible, unambiguous copy assignment operator for the class type.

C / C++

Fortran

- A list item that appears in a **copyin** clause must be threadprivate. Named variables appearing in a threadprivate common block may be specified: it is not necessary to specify the whole common block.
- A common block name that appears in a **copyin** clause must be declared to be a common block in the same scoping unit in which the **copyin** clause appears.
- If the list item is a polymorphic variable with the **ALLOCATABLE** attribute, the behavior is unspecified.

Fortran

2.20.5.2 copyprivate Clause

Summary

The **copyprivate** clause provides a mechanism to use a private variable to broadcast a value from the data environment of one implicit task to the data environments of the other implicit tasks belonging to the **parallel** region.

To avoid race conditions, concurrent reads or updates of the list item must be synchronized with the update of the list item that occurs as a result of the **copyprivate** clause.

Syntax

The syntax of the **copyprivate** clause is as follows:

```
copyprivate(list)
```

1 ### Description

2 The effect of the **copyprivate** clause on the specified list items occurs after the execution of the
3 structured block associated with the **single** construct (see Section 2.10.3 on page 90), and before
4 any of the threads in the team have left the barrier at the end of the construct.

C / C++

5 In all other implicit tasks belonging to the **parallel** region, each specified list item becomes
6 defined with the value of the corresponding list item in the implicit task associated with the thread
7 that executed the structured block. For variables of non-array type, the definition occurs by copy
8 assignment. For an array of elements of non-array type, each element is copied by copy assignment
9 from an element of the array in the data environment of the implicit task associated with the thread
10 that executed the structured block to the corresponding element of the array in the data environment
11 of the other implicit tasks

C / C++

C++

12 For class types, a copy assignment operator is invoked. The order in which copy assignment
13 operators for different variables of class type are called is unspecified.

C++

Fortran

14 If a list item does not have the **POINTER** attribute, then in all other implicit tasks belonging to the
15 **parallel** region, the list item becomes defined as if by intrinsic assignment with the value of the
16 corresponding list item in the implicit task associated with the thread that executed the structured
17 block.

18 If the list item has the **POINTER** attribute, then, in all other implicit tasks belonging to the
19 **parallel** region, the list item receives, as if by pointer assignment, the same association status of
20 the corresponding list item in the implicit task associated with the thread that executed the
21 structured block.

22 The order in which any final subroutines for different variables of a finalizable type are called is
23 unspecified.

Fortran

24 Note – The **copyprivate** clause is an alternative to using a shared variable for the value when
25 providing such a shared variable would be difficult (for example, in a recursion requiring a different
26 variable at each level).

1 **Restrictions**

2 The restrictions to the **copyprivate** clause are as follows:

3 • All list items that appear in the **copyprivate** clause must be either threadprivate or private in
4 the enclosing context.

5 • A list item that appears in a **copyprivate** clause may not appear in a **private** or
6 **firstprivate** clause on the **single** construct.

C++

7 • A variable of class type (or array thereof) that appears in a **copyprivate** clause requires an
8 accessible unambiguous copy assignment operator for the class type.

C++

Fortran

9 • A common block that appears in a **copyprivate** clause must be threadprivate.

10 • Pointers with the **INTENT(IN)** attribute may not appear in the **copyprivate** clause.

11 • The list item with the **ALLOCATABLE** attribute must have the allocation status of allocated when
12 the intrinsic assignment is performed.

13 • If the list item is a polymorphic variable with the **ALLOCATABLE** attribute, the behavior is
14 unspecified.

Fortran

15
2.20.6 Data-mapping Attribute Rules and Clauses

16 This section describes how the data-mapping attributes of any variable referenced in a **target**
17 region are determined. When specified, explicit **map** clauses on **target** directives determine
18 these attributes. Otherwise, the following data-mapping rules apply for variables referenced in a
19 **target** construct that are not declared in the construct and do not appear in data-sharing attribute
20 or **map** clauses:

21 Certain variables and objects have predetermined data-mapping attributes as follows:

22 • If a variable appears in a **to** or **link** clause on a **declare target** directive then it is treated
23 as if it had appeared in a **map** clause with a *map-type* of **tofrom**.

C / C++

24 • A variable that is of type pointer is treated as if it had appeared in a **map** clause as a zero-length
25 array section.

C / C++

C++

- A variable that is of type reference to pointer is treated as if it had appeared in a **map** clause as a zero-length array section.
- A class member variable is treated as if the *this[:1]* expression had appeared in a **map** clause.

C++

Otherwise, the following implicit data-mapping attribute rules apply:

- If a **defaultmap(tofrom:scalar)** clause is not present then a scalar variable (or a nonallocatable nonpointer scalar variable, or a variable without the **TARGET** attribute, for Fortran) is not mapped, but instead has an implicit data-sharing attribute of firstprivate (see Section 2.20.1.1 on page 240).
- If a **defaultmap(tofrom:scalar)** clause is present then a scalar variable is treated as if it had appeared in a **map** clause with a *map-type* of **tofrom**.
- If a variable is not a scalar then it is treated as if it had appeared in a **map** clause with a *map-type* of **tofrom**.

Fortran

- If a scalar variable that has the **TARGET** attribute then it is treated as if it has appeared in a **map** clause with a *map-type* of **tofrom**.

Fortran

2.20.6.1 map Clause

Summary

The **map** clause specifies how an original list item is mapped from the current task's data environment to a corresponding list item in the device data environment of the device identified by the construct.

Syntax

The syntax of the map clause is as follows:

```
map ([ [map-type-modifier[,]] map-type : ] list)
```

where *map-type* is one of the following:

to

from

tofrom

alloc

release

delete

and *map-type-modifier* is one of the following:

always

mapper(*mapper-identifier***)**

Description

The list items that appear in a **map** clause may include array sections and structure elements.

The *map-type* and *map-type-modifier* specify the effect of the **map** clause, as described below.

For a given construct, the effect of a **map** clause with the **to**, **from**, or **tofrom** *map-type* is ordered before the effect of a **map** clause with the **alloc**, **release**, or **delete** *map-type*. If a mapper is specified for the type being mapped, or explicitly specified with the **mapper** *map-type-modifier*, then the effective **map-type** of a list item will be determined according to the rules of map-type decay.

If a mapper is specified for the type being mapped, or explicitly specified with the **mapper** *map-type-modifier*, then all map clauses that appear on the **declare mapper** directive are treated as though they appeared on the construct with the **map** clause. Array sections of a mapper type are mapped as normal, then each element in the array section is mapped according to the rules of the mapper.

C / C++

If a list item in a **map** clause is a variable of structure type then it is treated as if each structure element contained in the variable is a list item in the clause.

C / C++

Fortran

If a list item in a **map** clause is a derived type variable then it is treated as if each nonpointer component is a list item in the clause.

Fortran

If a list item in a **map** clause is a structure element then all other structure elements (except pointer component, for Fortran) of the containing structure variable form a *structure sibling list*. The **map** clause and the structure sibling list are associated with the same construct. If a corresponding list item of the structure sibling list item is present in the device data environment when the construct is encountered then:

- If the structure sibling list item does not appear in a **map** clause on the construct then:
  - If the construct is a **target**, **target data**, or **target enter data** construct then the structure sibling list item is treated as if it is a list item in a **map** clause on the construct with a *map-type* of **alloc**.
  - If the construct is **target exit data** construct, then the structure sibling list item is treated as if it is a list item in a **map** clause on the construct with a *map-type* of **release**.
- If the **map** clause in which the structure element appears as a list item has a *map-type* of **delete** and the structure sibling list item does not appear as a list item in a **map** clause on the construct with a *map-type* of **delete** then the structure sibling list item is treated as if it is a list item in a **map** clause on the construct with a *map-type* of **delete**.

Fortran

If a list item in a **map** clause has the POINTER attribute and if the association status of the list item is associated, then it is treated as if the pointer target is a list item in the clause.

Fortran

C / C++

If a list item in a **map** clause is a variable of pointer type, and the variable is the named pointer of an array section that is a list item in a **map** clause on the same construct, then the effect of a **map** clause on the pointer variable and the effect of a **map** clause on the array section are ordered as follows:

- If the **map** clauses appear on a **target**, **target data**, or **target enter data** construct then on entry to the region the effect of the **map** clause on the pointer variable is ordered to occur before the effect of the **map** clause on the array section.
- If the **map** clauses appears on a **target**, **target data**, or **target exit data** construct then on exit from the region the effect of the **map** clause on the array section is ordered to occur before the effect of the **map** clause on the pointer variable.

If an array section with a named pointer is a list item in a **map** clause and a pointer variable is present in the device data environment that corresponds to the named pointer when the effect of the **map** clause occurs, then if the corresponding array section is created in the device data environment:

1. The corresponding pointer variable is assigned the address of the corresponding array section.
2. The corresponding pointer variable becomes an attached pointer for the corresponding array section.

C / C++

C++

If a *lambda* is mapped explicitly or implicitly, variables that are captured by the *lambda* behave as follows:

- the variables that are of pointer type are treated as if they had appeared in a **map** clause as zero-length array sections
- the variables that are of reference type are treated as if they had appeared in a **map** clause.

If a member variable is captured by a *lambda* in class scope, and the *lambda* is later mapped explicitly or implicitly with its full static type, the *this* pointer is treated as if it had appeared on a **map** clause.

C++

The original and corresponding list items may share storage such that writes to either item by one task followed by a read or write of the other item by another task without intervening synchronization can result in data races.

If the **map** clause appears on a **target**, **target data**, or **target enter data** construct then on entry to the region the following sequence of steps occurs as if performed as a single atomic operation:

1. If a corresponding list item of the original list item is not present in the device data environment, then:
   a) A new list item with language-specific attributes is derived from the original list item and created in the device data environment.
   b) The new list item becomes the corresponding list item to the original list item in the device data environment.
   c) The corresponding list item has a reference count that is initialized to zero.
   d) The value of the corresponding list item is undefined.
2. If the corresponding list item's reference count was not already incremented because of the effect of a **map** clause on the construct then:

a) The corresponding list item's reference count is incremented by one

3. If the corresponding list item's reference count is one or the **always** *map-type-modifier* is present, then:

   a) If the *map-type* is **to** or **tofrom**, then:

      - For each part of the list item that is an attached pointer:
        – That part of the corresponding list item will have the value it had immediately prior to the effect of the **map** clause;
      - For each part of the list item that is not an attached pointer:
        – The value of that part of the original list item is assigned to that part of the corresponding list item.
      - Concurrent reads or updates of any part of the corresponding list item must be synchronized with the update of the corresponding list item that occurs as a result of the **map** clause.

Note – If the effect of the **map** clauses on a construct would assign the value of an original list item to a corresponding list item more than once, then an implementation is allowed to ignore additional assignments of the same value to the corresponding list item.

If the **map** clause appears on a **target**, **target data**, or **target exit data** construct then on exit from the region the following sequence of steps occurs as if performed as a single atomic operation:

1. If a corresponding list item of the original list item is not present in the device data environment, then the list item is ignored.
2. If a corresponding list item of the original list item is present in the device data environment, then:

   a) If the corresponding list item's reference count is finite, then:

      i. If the corresponding list item's reference count was not already decremented because of the effect of a **map** clause on the construct then:

         A. If the *map-type* is not **delete**, then the corresponding list item's reference count is decremented by one.

      ii. If the *map-type* is **delete**, then the corresponding list item's reference count is set to zero.

   b) If the corresponding list item's reference count is zero or the **always** *map-type-modifier* is present, then:

i. If the *map-type* is **from** or **tofrom** then:

- For each part of the list item that is an attached pointer:
  - That part of the original list item will have the value it had immediately prior to the effect of the **map** clause;
- For each part of the list item that is not an attached pointer:
  - The value of that part of the corresponding list item is assigned to that part of the original list item;
- To avoid race conditions:
  - Concurrent reads or updates of any part of the original list item must be synchronized with the update of the original list item that occurs as a result of the **map** clause;

c) If the corresponding list item's reference count is zero, then the corresponding list item is removed from the device data environment

Note – If the effect of the **map** clauses on a construct would assign the value of a corresponding list item to an original list item more than once, then an implementation is allowed to ignore additional assignments of the same value to the original list item.

If a single contiguous part of the original storage of a list item with an implicit data-mapping attribute has corresponding storage in the device data environment prior to a task encountering the construct associated with the **map** clause, only that part of the original storage will have corresponding storage in the device data environment as a result of the **map** clause.

C / C++

If a new list item is created then a new list item of the same type, with automatic storage duration, is allocated for the construct. The size and alignment of the new list item are determined by the static type of the variable. This allocation occurs if the region references the list item in any statement.

C / C++

Fortran

If a new list item is created then a new list item of the same type, type parameter, and rank is allocated.

Fortran

The *map-type* determines how the new list item is initialized.

If a *map-type* is not specified, the *map-type* defaults to **tofrom**.

Events

The *target-map* event occurs when a thread maps data to or from a target device.

The *target-data-op* event occurs when a thread initiates a data operation on a target device.

Tool Callbacks

A thread dispatches a registered **ompt_callback_target_map** callback for each occurrence of a *target-map* event in that thread. The callback occurs in the context of the target task. The callback has type signature **ompt_callback_target_map_t**.

A thread dispatches a registered **ompt_callback_target_data_op** callback for each occurrence of a *target-data-op* event in that thread. The callback occurs in the context of the target task. The callback has type signature **ompt_callback_target_data_op_t**.

Restrictions

- A list item cannot appear in both a **map** clause and a data-sharing attribute clause on the same construct.
- If a list item is an array section, it must specify contiguous storage.
- If more than one list item of the **map** clauses on the same construct are array items that have the same named array, they must indicate identical original storage.
- List items of the **map** clauses on the same construct must not share original storage unless they are the same variable or array section.
- If any part of the original storage of a list item with a predetermined or explicit data-mapping attribute has corresponding storage in the device data environment prior to a task encountering the construct associated with the map clause, all of the original storage must have corresponding storage in the device data environment prior to the task encountering the construct.
- If a list item is an element of a structure, and a different element of the structure has a corresponding list item in the device data environment prior to a task encountering the construct associated with the **map** clause, then the list item must also have a corresponding list item in the device data environment prior to the task encountering the construct.
- If a list item is an element of a structure, only the rightmost symbol of the variable reference can be an array section.
- A list item must have a mappable type.
- **threadprivate** variables cannot appear in a **map** clause.
- If a **mapper** map-type-modifier is specified, its type must match the type of the list-items passed to that map clause.

C++

1 • If the type of a list item is a reference to a type *T* then the type will be considered to be *T* for all
2 purposes of this clause.

3 • If the list item is a *lambda*, any pointers and references captured by the *lambda* must have the
4 corresponding list item in the device data environment prior to the task encountering the
5 construct.

6 • In the class scope, if the *lambda* is passed as a parameter to a function in which it is specified in
7 the **map** clause, the behavior is unspecified.

C++

C / C++

8 • Initialization and assignment are through bitwise copy.

9 • A list item cannot be a variable that is a member of a structure with a union type.

10 • A bit-field cannot appear in a **map** clause.

11 • A pointer that has a corresponding attached pointer may not be modified for the duration of the
12 lifetime of the array section to which the corresponding pointer is attached in the device data
13 environment.

C / C++

Fortran

14 • The value of the new list item becomes that of the original list item in the map initialization and
15 assignment.

16 • A list item must not contain any components that have the **ALLOCATABLE** attribute.

17 • If the allocation status of a list item with the **ALLOCATABLE** attribute is unallocated upon entry
18 to a **target** region, the list item must be unallocated upon exit from the region.

19 • If the allocation status of a list item with the **ALLOCATABLE** attribute is allocated upon entry to
20 a **target** region, the allocation status of the corresponding list item must not be changed and
21 must not be reshaped in the region.

22 • If an array section is mapped and the size of the section is smaller than that of the whole array,
23 the behavior of referencing the whole array in the **target** region is unspecified.

24 • A list item must not be a whole array of an assumed-size array.

25 • If the association status of a list item with the **POINTER** attribute is associated upon entry to a
26 **target** region, the list item remains associated with the same pointer target upon exit from the
27 region.

28 • If the association status of a list item with the **POINTER** attribute is disassociated upon entry to a
29 **target** region, the list item must be disassociated upon exit from the region.

- If the association status of a list item with the **POINTER** attribute is undefined upon entry to a **target** region, the list item must be undefined upon exit from the region.
- If the association status of a list item with the **POINTER** attribute is disassociated or undefined on entry and if the list item is associated with a pointer target inside a **target** region, then the pointer association status must become disassociated before the end of the region; otherwise the behavior is unspecified.
- If the allocation status of the original list item with the **ALLOCATABLE** attribute is changed on the host device data environment and the corresponding list item is already present on the device data environment, the behavior is unspecified.

Fortran

Cross References

- **ompt_callback_target_map_t**, see Section 4.1.4.2.22 on page 431.
- **ompt_callback_target_data_op_t**, see Section 4.1.4.2.21 on page 428.

2.20.6.2 defaultmap Clause

Summary

The **defaultmap** clause explicitly determines the data-mapping attributes of variables that are referenced in a **target** construct and would otherwise be implicitly determined.

Syntax

The syntax of the **defaultmap** clause is as follows:

```
defaultmap(tofrom:scalar | none)
```

Description

The **defaultmap(tofrom:scalar)** clause causes all scalar variables referenced in the construct that have implicitly determined data-mapping attributes to have the **tofrom** *map-type*.

The **defaultmap(none)** clause requires all variables referenced in the construct to have explicit data-mapping or data sharing attributes, or to be listed in an **is_device_ptr** clause.

2.21 **declare reduction** Directive

Summary

The following section describes the directive for declaring user-defined reductions. The **declare reduction** directive declares a *reduction-identifier* that can be used in a **reduction** clause. The **declare reduction** directive is a declarative directive.

Syntax

C

#pragma omp declare reduction(*reduction-identifier* **:** *typename-list* **:** *combiner* **)** *[initializer-clause] new-line*

where:

- *reduction-identifier* is either a base language identifier or one of the following operators: **+**, **−**, *****, **&**, **|**, **^**, **&&** and **||**
- *typename-list* is a list of type names
- *combiner* is an expression
- *initializer-clause* is **initializer(***initializer-expr***)** where *initializer-expr* is **omp_priv =** *initializer* or *function-name***(***argument-list***)**

C

C++

#pragma omp declare reduction(*reduction-identifier* **:** *typename-list* **:** *combiner***)** *[initializer-clause] new-line*

where:

- *reduction-identifier* is either an *id-expression* or one of the following operators: **+**, **−**, *****, **&**, **|**, **^**, **&&** and **||**
- *typename-list* is a list of type names
- *combiner* is an expression
- *initializer-clause* is **initializer(***initializer-expr***)** where *initializer-expr* is **omp_priv** *initializer* or *function-name***(***argument-list***)**

C++

Fortran

```
!$omp declare reduction(reduction-identifier : type-list : combiner)
[initializer-clause]
```

where:

- *reduction-identifier* is either a base language identifier, or a user-defined operator, or one of the following operators: **+**, **-**, *****, **.and.**, **.or.**, **.eqv.**, **.neqv.**, or one of the following intrinsic procedure names: **max**, **min**, **iand**, **ior**, **ieor**.
- *type-list* is a list of type specifiers that must not be **CLASS(*)**
- *combiner* is either an assignment statement or a subroutine name followed by an argument list
- *initializer-clause* is **initializer**(*initializer-expr*), where *initializer-expr* is **omp_priv =** *expression* or *subroutine-name*(*argument-list*)

Fortran

Description

Custom reductions can be defined using the **declare reduction** directive; the *reduction-identifier* and the type identify the **declare reduction** directive. The *reduction-identifier* can later be used in a **reduction** clause using variables of the type or types specified in the **declare reduction** directive. If the directive applies to several types then it is considered as if there were multiple **declare reduction** directives, one for each type.

Fortran

If a type with deferred or assumed length type parameter is specified in a **declare reduction** directive, the *reduction-identifier* of that directive can be used in a **reduction** clause with any variable of the same type and the same kind parameter, regardless of the length type Fortran parameters with which the variable is declared.

Fortran

The visibility and accessibility of this declaration are the same as those of a variable declared at the same point in the program. The enclosing context of the *combiner* and of the *initializer-expr* will be that of the **declare reduction** directive. The *combiner* and the *initializer-expr* must be correct in the base language as if they were the body of a function defined at the same point in the program.

Fortran

If the *reduction-identifier* is the same as the name of a user-defined operator or an extended operator, or the same as a generic name that is one of the allowed intrinsic procedures, and if the operator or procedure name appears in an accessibility statement in the same module, the accessibility of the corresponding **declare reduction** directive is determined by the accessibility attribute of the statement.

If the *reduction-identifier* is the same as a generic name that is one of the allowed intrinsic procedures and is accessible, and if it has the same name as a derived type in the same module, the accessibility of the corresponding **declare reduction** directive is determined by the accessibility of the generic name according to the base language.

Fortran

C++

The **declare reduction** directive can also appear at points in the program at which a static data member could be declared. In this case, the visibility and accessibility of the declaration are the same as those of a static data member declared at the same point in the program.

C++

The *combiner* specifies how partial results can be combined into a single value. The *combiner* can use the special variable identifiers **omp_in** and **omp_out** that are of the type of the variables being reduced with this *reduction-identifier*. Each of them will denote one of the values to be combined before executing the *combiner*. It is assumed that the special **omp_out** identifier will refer to the storage that holds the resulting combined value after executing the *combiner*.

The number of times the *combiner* is executed, and the order of these executions, for any **reduction** clause is unspecified.

Fortran

If the *combiner* is a subroutine name with an argument list, the *combiner* is evaluated by calling the subroutine with the specified argument list.

If the *combiner* is an assignment statement, the *combiner* is evaluated by executing the assignment statement.

Fortran

As the *initializer-expr* value of a user-defined reduction is not known *a priori* the *initializer-clause* can be used to specify one. Then the contents of the *initializer-clause* will be used as the initializer for private copies of reduction list items where the **omp_priv** identifier will refer to the storage to be initialized. The special identifier **omp_orig** can also appear in the *initializer-clause* and it will refer to the storage of the original variable to be reduced.

The number of times that the *initializer-expr* is evaluated, and the order of these evaluations, is unspecified.

C / C++

If the *initializer-expr* is a function name with an argument list, the *initializer-expr* is evaluated by calling the function with the specified argument list. Otherwise, the *initializer-expr* specifies how **omp_priv** is declared and initialized.

C / C++

C

If no *initializer-clause* is specified, the private variables will be initialized following the rules for initialization of objects with static storage duration.

C

C++

If no *initializer-expr* is specified, the private variables will be initialized following the rules for *default-initialization*.

C++

Fortran

If the *initializer-expr* is a subroutine name with an argument list, the *initializer-expr* is evaluated by calling the subroutine with the specified argument list.

If the *initializer-expr* is an assignment statement, the *initializer-expr* is evaluated by executing the assignment statement.

If no *initializer-clause* is specified, the private variables will be initialized as follows:

- For **complex**, **real**, or **integer** types, the value 0 will be used.
- For **logical** types, the value **.false.** will be used.
- For derived types for which default initialization is specified, default initialization will be used.
- Otherwise, not specifying an *initializer-clause* results in unspecified behavior.

Fortran

C / C++

If *reduction-identifier* is used in a **target** region then a **declare target** construct must be specified for any function that can be accessed through the *combiner* and *initializer-expr*.

C / C++

Fortran

If *reduction-identifier* is used in a **target** region then a **declare target** construct must be specified for any function or subroutine that can be accessed through the *combiner* and *initializer-expr*.

Fortran

Restrictions

- The only variables allowed in the *combiner* are **omp_in** and **omp_out**.
- The only variables allowed in the *initializer-clause* are **omp_priv** and **omp_orig**.
- If the variable **omp_orig** is modified in the *initializer-clause*, the behavior is unspecified.
- If execution of the *combiner* or the *initializer-expr* results in the execution of an OpenMP construct or an OpenMP API call, then the behavior is unspecified.
- A *reduction-identifier* may not be re-declared in the current scope for the same type or for a type that is compatible according to the base language rules.
- At most one *initializer-clause* can be specified.

C / C++

- A type name in a **declare reduction** directive cannot be a function type, an array type, a reference type, or a type qualified with **const**, **volatile** or **restrict**.

C / C++

C

- If the *initializer-expr* is a function name with an argument list, then one of the arguments must be the address of **omp_priv**.

C

C++

- If the *initializer-expr* is a function name with an argument list, then one of the arguments must be **omp_priv** or the address of **omp_priv**.

C++

Fortran

1 • If the *initializer-expr* is a subroutine name with an argument list, then one of the arguments must
2 be **omp_priv**.

3 • If the **declare reduction** directive appears in the specification part of a module and the
4 corresponding reduction clause does not appear in the same module, the *reduction-identifier* must
5 be the same as the name of a user-defined operator, one of the allowed operators that is extended
6 or a generic name that is the same as the name of one of the allowed intrinsic procedures.

7 • If the **declare reduction** directive appears in the specification of a module, if the
8 corresponding **reduction** clause does not appear in the same module, and if the
9 *reduction-identifier* is the same as the name of a user-defined operator or an extended operator, or
10 the same as a generic name that is the same as one of the allowed intrinsic procedures then the
11 interface for that operator or the generic name must be defined in the specification of the same
12 module, or must be accessible by use association.

13 • Any subroutine or function used in the **initializer** clause or *combiner* expression must be
14 an intrinsic function, or must have an accessible interface.

15 • Any user-defined operator or extended operator used in the **initializer** clause or *combiner*
16 expression must have an accessible interface.

17 • If any subroutine, function, user-defined operator, or extended operator is used in the
18 **initializer** clause or *combiner* expression, it must be accessible to the subprogram in
19 which the corresponding **reduction** clause is specified.

20 • If the length type parameter is specified for a character type, it must be a constant, a colon or an *****.

21 • If a character type with deferred or assumed length parameter is specified in a
22 **declare reduction** directive, no other **declare reduction** directive with Fortran
23 character type of the same kind parameter and the same *reduction-identifier* is allowed in the
24 same scope.

25 • Any subroutine used in the **initializer** clause or *combiner* expression must not have any
26 alternate returns appear in the argument list.

Fortran

27 ## Cross References

28 • **reduction** clause, Section 2.20.4.4 on page 272.

2.22 Nesting of Regions

This section describes a set of restrictions on the nesting of regions. The restrictions on nesting are as follows:

- A worksharing region may not be closely nested inside a worksharing, **task**, **taskloop**, **critical**, **ordered**, **atomic**, or **master** region.
- A **barrier** region may not be closely nested inside a worksharing, **task**, **taskloop**, **critical**, **ordered**, **atomic**, or **master** region.
- A **master** region may not be closely nested inside a worksharing, **atomic**, **task**, or **taskloop** region.
- An **ordered** region arising from an **ordered** construct without any clause or with the **threads** or **depend** clause may not be closely nested inside a **critical**, **ordered**, **atomic**, **task**, or **taskloop** region.
- An **ordered** region arising from an **ordered** construct without any clause or with the **threads** or **depend** clause must be closely nested inside a loop region (or parallel loop region) with an **ordered** clause.
- An **ordered** region arising from an **ordered** construct with the **simd** clause must be closely nested inside a **simd** (or loop SIMD) region.
- An **ordered** region arising from an **ordered** construct with both the **simd** and **threads** clauses must be closely nested inside a loop SIMD region.
- A **critical** region may not be nested (closely or otherwise) inside a **critical** region with the same name. This restriction is not sufficient to prevent deadlock.
- OpenMP constructs may not be encountered during execution of an **atomic** region.
- The only OpenMP constructs that can be encountered during execution of a **simd** (or loop SIMD) region are the **atomic** construct, **concurrent** construct, and an **ordered** construct with the **simd** clause.
- If a **target**, **target update**, **target data**, **target enter data**, or **target exit data** construct is encountered during execution of a **target** region, the behavior is unspecified.
- If specified, a **teams** construct must be contained within a **target** construct. That **target** construct must not contain any statements or directives outside of the **teams** construct.
- **distribute**, **distribute simd**, distribute parallel loop, distribute parallel loop SIMD, **concurrent**, and **parallel** regions, including any **parallel** regions arising from combined constructs, are the only OpenMP regions that may be strictly nested inside the **teams** region.

1 • The region associated with the **distribute** construct must be strictly nested inside a **teams**
2 region.

3 • If *construct-type-clause* is **taskgroup**, the **cancel** construct must be closely nested inside a
4 **task** construct and the **cancel** region must be closely nested inside a **taskgroup** region. If
5 *construct-type-clause* is **sections**, the **cancel** construct must be closely nested inside a
6 **sections** or **section** construct. Otherwise, the **cancel** construct must be closely nested
7 inside an OpenMP construct that matches the type specified in *construct-type-clause* of the
8 **cancel** construct.

9 • A **cancellation point** construct for which *construct-type-clause* is **taskgroup** must be
10 closely nested inside a **task** construct, and the **cancellation point** region must be closely
11 nested inside a **taskgroup** region. A **cancellation point** construct for which
12 *construct-type-clause* is **sections** must be closely nested inside a **sections** or **section**
13 construct. Otherwise, a **cancellation point** construct must be closely nested inside an
14 OpenMP construct that matches the type specified in *construct-type-clause*.

15 • A **concurrent** region must be closely nested within a **teams** construct, **distribute**
16 construct, **parallel** construct, Loop construct, SIMD construct, **concurrent** construct, or
17 regions resulting from the combined or composite constructs composed of these constructs.

18 • Only the following directives may be nested within a **concurrent** construct: **parallel**
19 construct, Loop construct, SIMD construct, **concurrent** construct, **single** construct, and
20 any combined or compound construct composed of these constructs.

CHAPTER 3

Runtime Library Routines

This chapter describes the OpenMP API runtime library routines and queryable runtime states, and is divided into the following sections:

- Runtime library definitions (Section 3.1 on page 298).
- Execution environment routines that can be used to control and to query the parallel execution environment (Section 3.2 on page 300).
- Lock routines that can be used to synchronize access to data (Section 3.3 on page 344).
- Portable timer routines (Section 3.4 on page 356).
- Device memory routines that can be used to allocate memory and to manage pointers on target devices (Section 3.5 on page 358).
- Execution routines to control the application monitoring (Section 3.7 on page 372)

Throughout this chapter, *true* and *false* are used as generic terms to simplify the description of the routines.

C / C++

true means a nonzero integer value and *false* means an integer value of zero.

C / C++

Fortran

true means a logical value of **.TRUE.** and *false* means a logical value of **.FALSE.**.

Fortran

Fortran

Restrictions

The following restriction applies to all OpenMP runtime library routines:

- OpenMP runtime library routines may not be called from **PURE** or **ELEMENTAL** procedures.

Fortran

3.1 Runtime Library Definitions

For each base language, a compliant implementation must supply a set of definitions for the OpenMP API runtime library routines and the special data types of their parameters. The set of definitions must contain a declaration for each OpenMP API runtime library routine and a declaration for the *simple lock*, *nestable lock*, *schedule*, and *thread affinity policy* data types. In addition, each set of definitions may specify other implementation specific values.

C / C++

The library routines are external functions with "C" linkage.

Prototypes for the C/C++ runtime library routines described in this chapter shall be provided in a header file named **omp.h**. This file defines the following:

- The prototypes of all the routines in the chapter.
- The type **omp_lock_t**.
- The type **omp_nest_lock_t**.
- The type **omp_lock_hint_t**.
- The type **omp_sched_t**.
- The type **omp_proc_bind_t**.
- The type **omp_control_tool_t**.
- The type **omp_control_tool_result_t**.
- The type **omp_allocator_t**.
- A global variable of type **const omp_allocator_t *** for each predefined memory allocator in Table 2.5 on page 64.

  A program that declares a new variable with the same identifier as one of the predefined allocators listed in Table 2.5 on page 64 results in unspecified behavior.

1 See Section B.1 on page 606 for an example of this file.

C / C++

Fortran

2 The OpenMP Fortran API runtime library routines are external procedures. The return values of
3 these routines are of default kind, unless otherwise specified.

4 Interface declarations for the OpenMP Fortran runtime library routines described in this chapter
5 shall be provided in the form of a Fortran **include** file named **omp_lib.h** or a Fortran 90
6 **module** named **omp_lib**. It is implementation defined whether the **include** file or the
7 **module** file (or both) is provided.

8 These files define the following:

9 • The interfaces of all of the routines in this chapter.

10 • The **integer parameter omp_lock_kind**.

11 • The **integer parameter omp_nest_lock_kind**.

12 • The **integer parameter omp_lock_hint_kind**.

13 • The **integer parameter omp_sched_kind**.

14 • The **integer parameter omp_proc_bind_kind**.

15 • The **integer parameter omp_allocator_kind**.

16 • An **integer parameter** variable of kind **omp_allocator_kind** for each predefined
17 memory allocator in Table 2.5 on page 64.

18 • The **integer parameter openmp_version** with a value *yyyymm* where *yyyy* and *mm* are
19 the year and month designations of the version of the OpenMP Fortran API that the
20 implementation supports. This value matches that of the C preprocessor macro **_OPENMP**, when
21 a macro preprocessor is supported (see Section 2.2 on page 44).

22 See Section B.1 on page 611 and Section B.3 on page 617 for examples of these files.

23 It is implementation defined whether any of the OpenMP runtime library routines that take an
24 argument are extended with a generic interface so arguments of different **KIND** type can be
25 accommodated. See Appendix B.4 for an example of such an extension.

Fortran

3.2 Execution Environment Routines

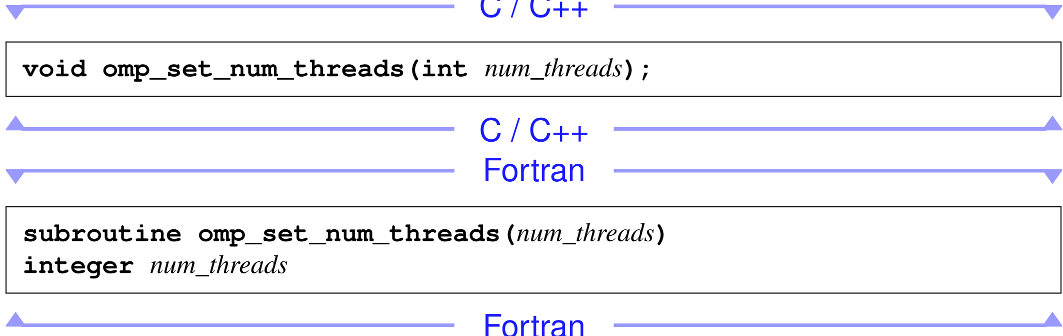
This section describes routines that affect and monitor threads, processors, and the parallel environment.

3.2.1 omp_set_num_threads

Summary

The **omp_set_num_threads** routine affects the number of threads to be used for subsequent parallel regions that do not specify a **num_threads** clause, by setting the value of the first element of the *nthreads-var* ICV of the current task.

Format



Constraints on Arguments

The value of the argument passed to this routine must evaluate to a positive integer, or else the behavior of this routine is implementation defined.

Binding

The binding task set for an **omp_set_num_threads** region is the generating task.

Effect

The effect of this routine is to set the value of the first element of the *nthreads-var* ICV of the current task to the value specified in the argument.

Cross References

- *nthreads-var* ICV, see Section 2.4 on page 49.
- **parallel** construct and **num_threads** clause, see Section 2.8 on page 66.
- Determining the number of threads for a **parallel** region, see Section 2.8.1 on page 71.
- **omp_get_max_threads** routine, see Section 3.2.3 on page 302.
- **OMP_NUM_THREADS** environment variable, see Section 5.2 on page 565.

3.2.2 omp_get_num_threads

Summary

The **omp_get_num_threads** routine returns the number of threads in the current team.

Format

C / C++

```
int omp_get_num_threads(void);
```

C / C++

Fortran

```
integer function omp_get_num_threads()
```

Fortran

Binding

The binding region for an **omp_get_num_threads** region is the innermost enclosing **parallel** region.

Effect

The **omp_get_num_threads** routine returns the number of threads in the team executing the **parallel** region to which the routine region binds. If called from the sequential part of a program, this routine returns 1.

Cross References

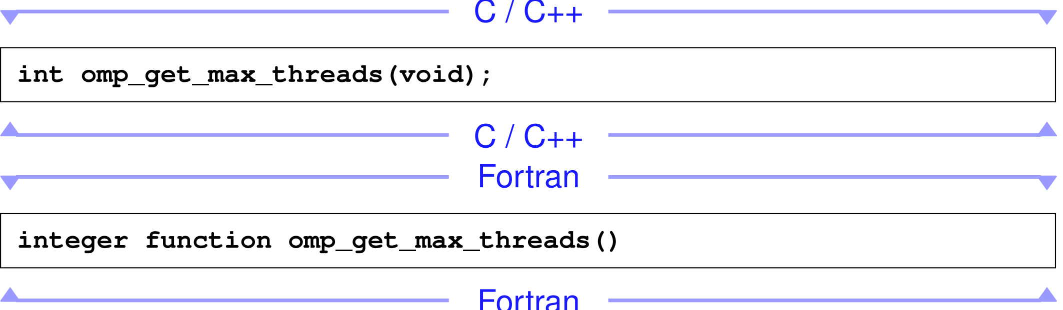
- **`parallel`** construct, see Section 2.8 on page 66.
- Determining the number of threads for a **`parallel`** region, see Section 2.8.1 on page 71.
- **`omp_set_num_threads`** routine, see Section 3.2.1 on page 300.
- **`OMP_NUM_THREADS`** environment variable, see Section 5.2 on page 565.

3.2.3 `omp_get_max_threads`

Summary

The **`omp_get_max_threads`** routine returns an upper bound on the number of threads that could be used to form a new team if a **`parallel`** construct without a **`num_threads`** clause were encountered after execution returns from this routine.

Format



Binding

The binding task set for an **`omp_get_max_threads`** region is the generating task.

Effect

The value returned by **omp_get_max_threads** is the value of the first element of the *nthreads-var* ICV of the current task. This value is also an upper bound on the number of threads that could be used to form a new team if a parallel region without a **num_threads** clause were encountered after execution returns from this routine.

Note – The return value of the **omp_get_max_threads** routine can be used to dynamically allocate sufficient storage for all threads in the team formed at the subsequent active **parallel** region.

Cross References

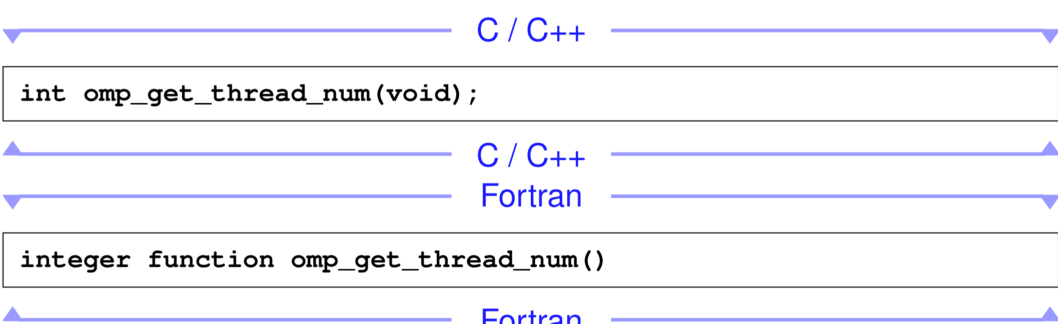
- *nthreads-var* ICV, see Section 2.4 on page 49.
- **parallel** construct, see Section 2.8 on page 66.
- **num_threads** clause, see Section 2.8 on page 66.
- Determining the number of threads for a **parallel** region, see Section 2.8.1 on page 71.
- **omp_set_num_threads** routine, see Section 3.2.1 on page 300.
- **OMP_NUM_THREADS** environment variable, see Section 5.2 on page 565.

3.2.4 omp_get_thread_num

Summary

The **omp_get_thread_num** routine returns the thread number, within the current team, of the calling thread.

Format



C / C++

```
int omp_get_thread_num(void);
```

C / C++

Fortran

```
integer function omp_get_thread_num()
```

Fortran

Binding

The binding thread set for an **omp_get_thread_num** region is the current team. The binding region for an **omp_get_thread_num** region is the innermost enclosing **parallel** region.

Effect

The **omp_get_thread_num** routine returns the thread number of the calling thread, within the team executing the **parallel** region to which the routine region binds. The thread number is an integer between 0 and one less than the value returned by **omp_get_num_threads**, inclusive. The thread number of the master thread of the team is 0. The routine returns 0 if it is called from the sequential part of a program.

Note – The thread number may change during the execution of an untied task. The value returned by **omp_get_thread_num** is not generally useful during the execution of such a task region.

Cross References

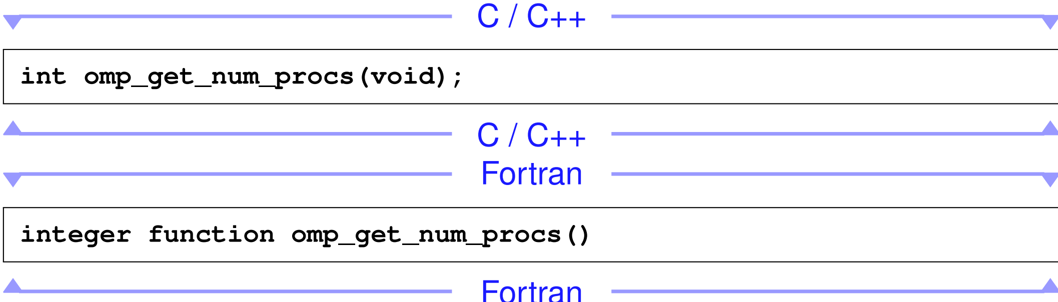
- **omp_get_num_threads** routine, see Section 3.2.2 on page 301.

3.2.5 omp_get_num_procs

Summary

The **omp_get_num_procs** routine returns the number of processors available to the device.

Format


C / C++

```
int omp_get_num_procs(void);
```

C / C++

Fortran

```
integer function omp_get_num_procs()
```

Fortran

Binding

The binding thread set for an **omp_get_num_procs** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The **omp_get_num_procs** routine returns the number of processors that are available to the device at the time the routine is called. This value may change between the time that it is determined by the **omp_get_num_procs** routine and the time that it is read in the calling context due to system actions outside the control of the OpenMP implementation.

Cross References

None.

3.2.6 omp_in_parallel

Summary

The **omp_in_parallel** routine returns *true* if the *active-levels-var* ICV is greater than zero; otherwise, it returns *false*.

Format

C / C++

```
int omp_in_parallel(void);
```

C / C++

Fortran

```
logical function omp_in_parallel()
```

Fortran

Binding

The binding task set for an **omp_in_parallel** region is the generating task.

Effect

The effect of the **omp_in_parallel** routine is to return *true* if the current task is enclosed by an active **parallel** region, and the **parallel** region is enclosed by the outermost initial task region on the device; otherwise it returns *false*.

Cross References

- *active-levels-var*, see Section 2.4 on page 49.
- **parallel** construct, see Section 2.8 on page 66.
- **omp_get_active_level** routine, see Section 3.2.20 on page 321.

3.2.7 omp_set_dynamic

Summary

The **omp_set_dynamic** routine enables or disables dynamic adjustment of the number of threads available for the execution of subsequent **parallel** regions by setting the value of the *dyn-var* ICV.

Format

C / C++

```
void omp_set_dynamic(int dynamic_threads);
```

C / C++

Fortran

```
subroutine omp_set_dynamic(dynamic_threads)
logical dynamic_threads
```

Fortran

Binding

The binding task set for an **omp_set_dynamic** region is the generating task.

Effect

For implementations that support dynamic adjustment of the number of threads, if the argument to **omp_set_dynamic** evaluates to *true*, dynamic adjustment is enabled for the current task; otherwise, dynamic adjustment is disabled for the current task. For implementations that do not support dynamic adjustment of the number of threads this routine has no effect: the value of *dyn-var* remains *false*.

Cross References

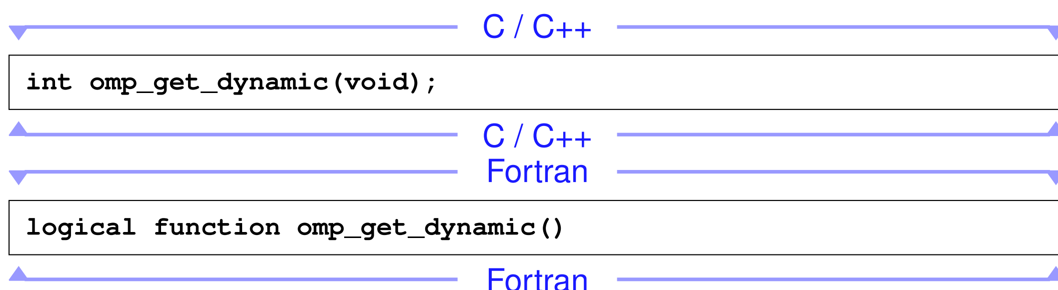
- *dyn-var* ICV, see Section 2.4 on page 49.
- Determining the number of threads for a **parallel** region, see Section 2.8.1 on page 71.
- **omp_get_num_threads** routine, see Section 3.2.2 on page 301.
- **omp_get_dynamic** routine, see Section 3.2.8 on page 308.
- **OMP_DYNAMIC** environment variable, see Section 5.3 on page 566.

3.2.8 `omp_get_dynamic`

Summary

The **`omp_get_dynamic`** routine returns the value of the *dyn-var* ICV, which determines whether dynamic adjustment of the number of threads is enabled or disabled.

Format




```c
int omp_get_dynamic(void);
```

```fortran
logical function omp_get_dynamic()
```

Binding

The binding task set for an **`omp_get_dynamic`** region is the generating task.

Effect

This routine returns *true* if dynamic adjustment of the number of threads is enabled for the current task; it returns *false*, otherwise. If an implementation does not support dynamic adjustment of the number of threads, then this routine always returns *false*.

Cross References

- *dyn-var* ICV, see Section 2.4 on page 49.
- Determining the number of threads for a **`parallel`** region, see Section 2.8.1 on page 71.
- **`omp_set_dynamic`** routine, see Section 3.2.7 on page 306.
- **`OMP_DYNAMIC`** environment variable, see Section 5.3 on page 566.

3.2.9 `omp_get_cancellation`

Summary

The **`omp_get_cancellation`** routine returns the value of the *cancel-var* ICV, which determines if cancellation is enabled or disabled.

Format

C / C++

```
int omp_get_cancellation(void);
```

C / C++

Fortran

```
logical function omp_get_cancellation()
```

Fortran

Binding

The binding task set for an **omp_get_cancellation** region is the whole program.

Effect

This routine returns *true* if cancellation is enabled. It returns *false* otherwise.

Cross References

- *cancel-var* ICV, see Section 2.4.1 on page 49.
- **cancel** construct, see Section 2.19.1 on page 232
- **OMP_CANCELLATION** environment variable, see Section 5.11 on page 572

3.2.10 omp_set_nested

Summary

The deprecated **omp_set_nested** routine enables or disables nested parallelism, by setting the *nest-var* ICV.

1
Format

C / C++

```
void omp_set_nested(int nested);
```

C / C++

Fortran

```
subroutine omp_set_nested(nested)
logical nested
```

Fortran

2
Binding

3 The binding task set for an **omp_set_nested** region is the generating task.

4
Effect

5 For implementations that support nested parallelism, if the argument to **omp_set_nested**
6 evaluates to *true*, nested parallelism is enabled for the current task; otherwise, nested parallelism is
7 disabled for the current task. For implementations that do not support nested parallelism, this
8 routine has no effect: the value of *nest-var* remains *false*. This routine has been deprecated.

9
Cross References

10 • *nest-var* ICV, see Section 2.4 on page 49.

11 • Determining the number of threads for a **parallel** region, see Section 2.8.1 on page 71.

12 • **omp_set_max_active_levels** routine, see Section 3.2.15 on page 315.

13 • **omp_get_max_active_levels** routine, see Section 3.2.16 on page 317.

14 • **omp_get_nested** routine, see Section 3.2.11 on page 311.

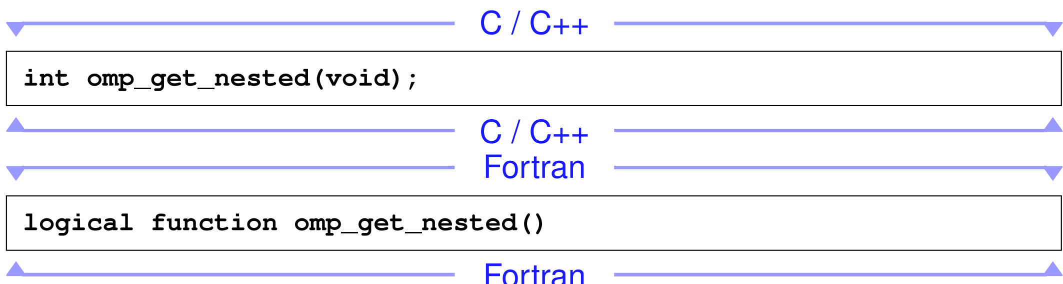
15 • **OMP_NESTED** environment variable, see Section 5.6 on page 569.

3.2.11 omp_get_nested

Summary

The deprecated **omp_get_nested** routine returns the value of the *nest-var* ICV, which determines if nested parallelism is enabled or disabled.

Format



Binding

The binding task set for an **omp_get_nested** region is the generating task.

Effect

This routine returns *true* if nested parallelism is enabled for the current task; it returns *false*, otherwise. If an implementation does not support nested parallelism, this routine always returns *false*. This routine has been deprecated.

Cross References

- *nest-var* ICV, see Section 2.4 on page 49.
- Determining the number of threads for a **parallel** region, see Section 2.8.1 on page 71.
- **omp_set_nested** routine, see Section 3.2.10 on page 309.
- **OMP_NESTED** environment variable, see Section 5.6 on page 569.

3.2.12 omp_set_schedule

Summary

The **omp_set_schedule** routine affects the schedule that is applied when **runtime** is used as schedule kind, by setting the value of the *run-sched-var* ICV.

1 **Format**

C / C++

```
void omp_set_schedule(omp_sched_t kind, int chunk_size);
```

C / C++

Fortran

```
subroutine omp_set_schedule(kind, chunk_size)
integer (kind=omp_sched_kind) kind
integer chunk_size
```

Fortran

2 **Constraints on Arguments**

3 The first argument passed to this routine can be one of the valid OpenMP schedule kinds (except for
4 **runtime**) or any implementation specific schedule. The C/C++ header file (**omp.h**) and the
5 Fortran include file (**omp_lib.h**) and/or Fortran 90 module file (**omp_lib**) define the valid
6 constants. The valid constants must include the following, which can be extended with
7 implementation specific values:

C / C++

```
typedef enum omp_sched_t {
    omp_sched_static = 1,
    omp_sched_dynamic = 2,
    omp_sched_guided = 3,
    omp_sched_auto = 4
} omp_sched_t;
```

C / C++

Fortran

```
integer(kind=omp_sched_kind), parameter :: omp_sched_static = 1
integer(kind=omp_sched_kind), parameter :: omp_sched_dynamic = 2
integer(kind=omp_sched_kind), parameter :: omp_sched_guided = 3
integer(kind=omp_sched_kind), parameter :: omp_sched_auto = 4
```

Fortran

Binding

The binding task set for an **omp_set_schedule** region is the generating task.

Effect

The effect of this routine is to set the value of the *run-sched-var* ICV of the current task to the values specified in the two arguments. The schedule is set to the schedule type specified by the first argument *kind*. It can be any of the standard schedule types or any other implementation specific one. For the schedule types **static**, **dynamic**, and **guided** the *chunk_size* is set to the value of the second argument, or to the default *chunk_size* if the value of the second argument is less than 1; for the schedule type **auto** the second argument has no meaning; for implementation specific schedule types, the values and associated meanings of the second argument are implementation defined.

Cross References

- *run-sched-var* ICV, see Section 2.4 on page 49.
- Determining the schedule of a worksharing loop, see Section 2.10.1.1 on page 86.
- **omp_get_schedule** routine, see Section 3.2.13 on page 313.
- **OMP_SCHEDULE** environment variable, see Section 5.1 on page 564.

3.2.13 omp_get_schedule

Summary

The **omp_get_schedule** routine returns the schedule that is applied when the runtime schedule is used.

Format

C / C++

```
void omp_get_schedule(omp_sched_t * kind, int * chunk_size);
```

C / C++

Fortran

```
subroutine omp_get_schedule(kind, chunk_size)
integer (kind=omp_sched_kind) kind
integer chunk_size
```

Fortran

Binding

The binding task set for an **omp_get_schedule** region is the generating task.

Effect

This routine returns the *run-sched-var* ICV in the task to which the routine binds. The first argument *kind* returns the schedule to be used. It can be any of the standard schedule types as defined in Section 3.2.12 on page 311, or any implementation specific schedule type. The second argument is interpreted as in the **omp_set_schedule** call, defined in Section 3.2.12 on page 311.

Cross References

- *run-sched-var* ICV, see Section 2.4 on page 49.
- Determining the schedule of a worksharing loop, see Section 2.10.1.1 on page 86.
- **omp_set_schedule** routine, see Section 3.2.12 on page 311.
- **OMP_SCHEDULE** environment variable, see Section 5.1 on page 564.

3.2.14 omp_get_thread_limit

Summary

The **omp_get_thread_limit** routine returns the maximum number of OpenMP threads available to participate in the current contention group.

Format

C / C++

```
int omp_get_thread_limit(void);
```

C / C++

Fortran

```
integer function omp_get_thread_limit()
```

Fortran

Binding

The binding thread set for an **omp_get_thread_limit** region is all threads on the device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The **omp_get_thread_limit** routine returns the value of the *thread-limit-var* ICV.

Cross References

- *thread-limit-var* ICV, see Section 2.4 on page 49.
- **OMP_THREAD_LIMIT** environment variable, see Section 5.10 on page 572.

3.2.15 omp_set_max_active_levels

Summary

The **omp_set_max_active_levels** routine limits the number of nested active parallel regions on the device, by setting the *max-active-levels-var* ICV

Format

C / C++

```
void omp_set_max_active_levels(int max_levels);
```

C / C++

Fortran

```
subroutine omp_set_max_active_levels(max_levels)
integer max_levels
```

Fortran

Constraints on Arguments

The value of the argument passed to this routine must evaluate to a non-negative integer, otherwise the behavior of this routine is implementation defined.

Binding

When called from a sequential part of the program, the binding thread set for an **omp_set_max_active_levels** region is the encountering thread. When called from within any explicit parallel region, the binding thread set (and binding region, if required) for the **omp_set_max_active_levels** region is implementation defined.

Effect

The effect of this routine is to set the value of the *max-active-levels-var* ICV to the value specified in the argument.

If the number of parallel levels requested exceeds the number of levels of parallelism supported by the implementation, the value of the *max-active-levels-var* ICV will be set to the number of parallel levels supported by the implementation.

This routine has the described effect only when called from a sequential part of the program. When called from within an explicit **parallel** region, the effect of this routine is implementation defined.

Cross References

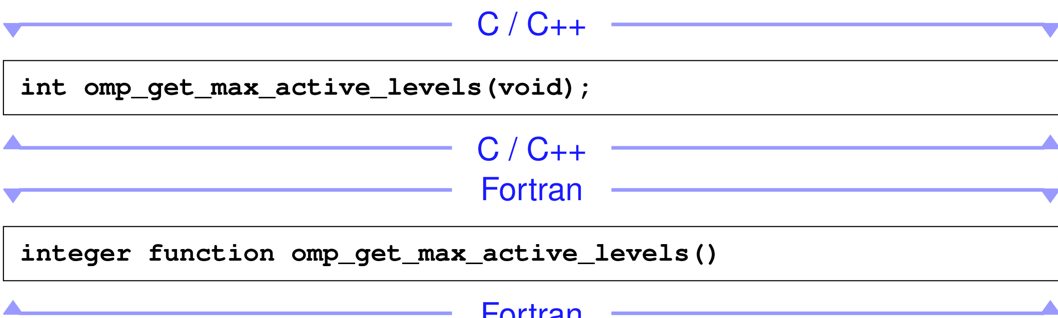
- *max-active-levels-var* ICV, see Section 2.4 on page 49.
- **omp_get_max_active_levels** routine, see Section 3.2.16 on page 317.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 5.9 on page 572.

3.2.16 `omp_get_max_active_levels`

Summary

The **`omp_get_max_active_levels`** routine returns the value of the *max-active-levels-var* ICV, which determines the maximum number of nested active parallel regions on the device.

Format



Binding

When called from a sequential part of the program, the binding thread set for an **`omp_get_max_active_levels`** region is the encountering thread. When called from within any explicit parallel region, the binding thread set (and binding region, if required) for the **`omp_get_max_active_levels`** region is implementation defined.

Effect

The **`omp_get_max_active_levels`** routine returns the value of the *max-active-levels-var* ICV, which determines the maximum number of nested active parallel regions on the device.

Cross References

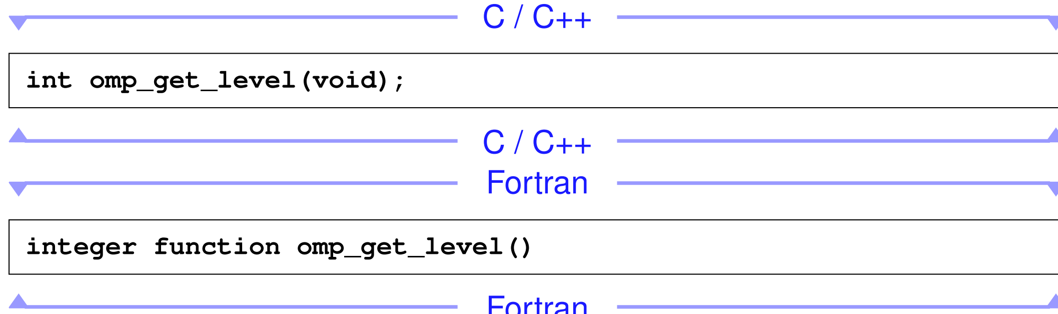
- *max-active-levels-var* ICV, see Section 2.4 on page 49.
- **`omp_set_max_active_levels`** routine, see Section 3.2.15 on page 315.
- **`OMP_MAX_ACTIVE_LEVELS`** environment variable, see Section 5.9 on page 572.

3.2.17 omp_get_level

Summary

The **omp_get_level** routine returns the value of the *levels-var* ICV.

Format



```
int omp_get_level(void);
```

```
integer function omp_get_level()
```

Binding

The binding task set for an **omp_get_level** region is the generating task.

Effect

The effect of the **omp_get_level** routine is to return the number of nested **parallel** regions (whether active or inactive) enclosing the current task such that all of the **parallel** regions are enclosed by the outermost initial task region on the current device.

Cross References

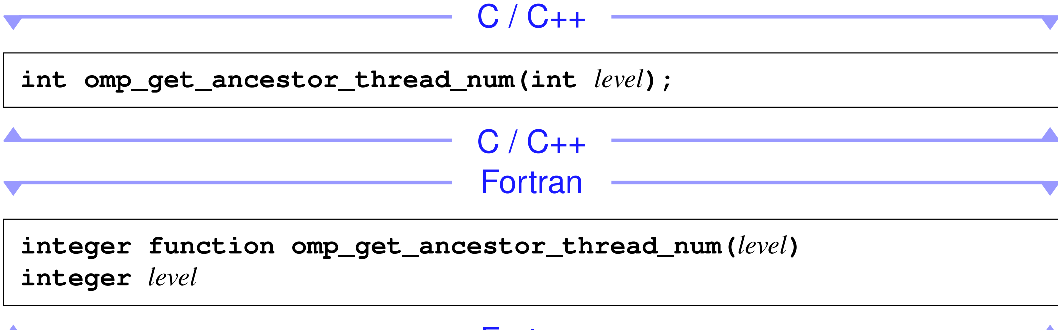
- *levels-var* ICV, see Section 2.4 on page 49.
- **omp_get_active_level** routine, see Section 3.2.20 on page 321.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 5.9 on page 572.

3.2.18 omp_get_ancestor_thread_num

Summary

The **omp_get_ancestor_thread_num** routine returns, for a given nested level of the current thread, the thread number of the ancestor of the current thread.

Format



C / C++

```
int omp_get_ancestor_thread_num(int level);
```

C / C++

Fortran

```
integer function omp_get_ancestor_thread_num(level)
integer level
```

Fortran

Binding

The binding thread set for an **omp_get_ancestor_thread_num** region is the encountering thread. The binding region for an **omp_get_ancestor_thread_num** region is the innermost enclosing **parallel** region.

Effect

The **omp_get_ancestor_thread_num** routine returns the thread number of the ancestor at a given nest level of the current thread or the thread number of the current thread. If the requested nest level is outside the range of 0 and the nest level of the current thread, as returned by the **omp_get_level** routine, the routine returns -1.

Note – When the **omp_get_ancestor_thread_num** routine is called with a value of **level**=0, the routine always returns 0. If **level=omp_get_level()**, the routine has the same effect as the **omp_get_thread_num** routine.

Cross References

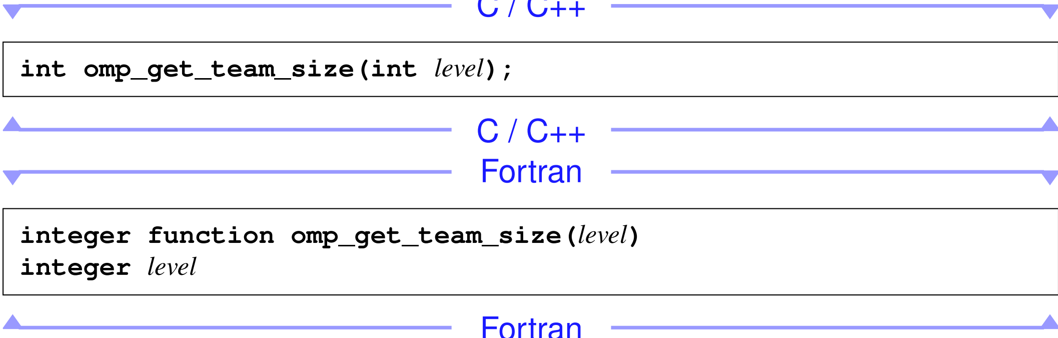
- **omp_get_thread_num** routine, see Section 3.2.4 on page 304.
- **omp_get_level** routine, see Section 3.2.17 on page 318.
- **omp_get_team_size** routine, see Section 3.2.19 on page 320.

3.2.19 omp_get_team_size

Summary

The **omp_get_team_size** routine returns, for a given nested level of the current thread, the size of the thread team to which the ancestor or the current thread belongs.

Format



Binding

The binding thread set for an **omp_get_team_size** region is the encountering thread. The binding region for an **omp_get_team_size** region is the innermost enclosing **parallel** region.

Effect

The **omp_get_team_size** routine returns the size of the thread team to which the ancestor or the current thread belongs. If the requested nested level is outside the range of 0 and the nested level of the current thread, as returned by the **omp_get_level** routine, the routine returns -1. Inactive parallel regions are regarded like active parallel regions executed with one thread.

Note – When the **omp_get_team_size** routine is called with a value of **level**=0, the routine always returns 1. If **level=omp_get_level()**, the routine has the same effect as the **omp_get_num_threads** routine.

Cross References

- **omp_get_num_threads** routine, see Section 3.2.2 on page 301.
- **omp_get_level** routine, see Section 3.2.17 on page 318.
- **omp_get_ancestor_thread_num** routine, see Section 3.2.18 on page 319.

3.2.20 omp_get_active_level

Summary

The **omp_get_active_level** routine returns the value of the *active-level-vars* ICV..

Format

C / C++

```
int omp_get_active_level(void);
```

C / C++

Fortran

```
integer function omp_get_active_level()
```

Fortran

Binding

The binding task set for the an **omp_get_active_level** region is the generating task.

Effect

The effect of the **omp_get_active_level** routine is to return the number of nested, active **parallel** regions enclosing the current task such that all of the **parallel** regions are enclosed by the outermost initial task region on the current device.

Cross References

- *active-levels-var* ICV, see Section 2.4 on page 49.
- **omp_get_level** routine, see Section 3.2.17 on page 318.

3.2.21 omp_in_final

Summary

The **omp_in_final** routine returns *true* if the routine is executed in a final task region; otherwise, it returns *false*.

Format

C / C++

```
int omp_in_final(void);
```

C / C++

Fortran

```
logical function omp_in_final()
```

Fortran

Binding

The binding task set for an **omp_in_final** region is the generating task.

Effect

omp_in_final returns *true* if the enclosing task region is final. Otherwise, it returns *false*.

Cross References

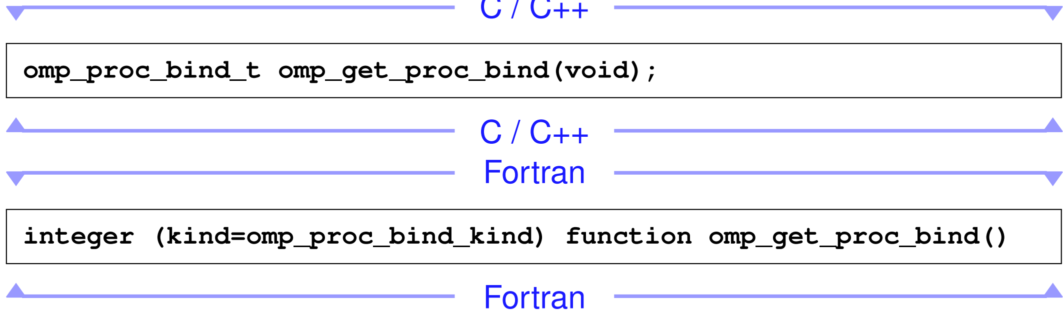
- **task** construct, see Section 2.13.1 on page 110.

3.2.22 omp_get_proc_bind

Summary

The **omp_get_proc_bind** routine returns the thread affinity policy to be used for the subsequent nested **parallel** regions that do not specify a **proc_bind** clause.

Format



C / C++

```
omp_proc_bind_t omp_get_proc_bind(void);
```

C / C++

Fortran

```
integer (kind=omp_proc_bind_kind) function omp_get_proc_bind()
```

Fortran

Constraints on Arguments

The value returned by this routine must be one of the valid affinity policy kinds. The C/ C++ header file (**omp.h**) and the Fortran include file (**omp_lib.h**) and/or Fortran 90 module file (**omp_lib**) define the valid constants. The valid constants must include the following:

C / C++

```
typedef enum omp_proc_bind_t {
  omp_proc_bind_false = 0,
  omp_proc_bind_true = 1,
  omp_proc_bind_master = 2,
  omp_proc_bind_close = 3,
  omp_proc_bind_spread = 4
} omp_proc_bind_t;
```

C / C++

Fortran

```
integer (kind=omp_proc_bind_kind), &
                parameter :: omp_proc_bind_false = 0
integer (kind=omp_proc_bind_kind), &
                parameter :: omp_proc_bind_true = 1
integer (kind=omp_proc_bind_kind), &
                parameter :: omp_proc_bind_master = 2
integer (kind=omp_proc_bind_kind), &
                parameter :: omp_proc_bind_close = 3
integer (kind=omp_proc_bind_kind), &
                parameter :: omp_proc_bind_spread = 4
```

Fortran

Binding

The binding task set for an **omp_get_proc_bind** region is the generating task

Effect

The effect of this routine is to return the value of the first element of the *bind-var* ICV of the current task. See Section 2.8.2 on page 73 for the rules governing the thread affinity policy.

Cross References

- *bind-var* ICV, see Section 2.4 on page 49.
- Controlling OpenMP thread affinity, see Section 2.8.2 on page 73.
- **OMP_PROC_BIND** environment variable, see Section 5.4 on page 566.

3.2.23 omp_get_num_places

Summary

The **omp_get_num_places** routine returns the number of places available to the execution environment in the place list.

Format

C / C++

```
int omp_get_num_places(void);
```

C / C++

Fortran

```
integer function omp_get_num_places()
```

Fortran

Binding

The binding thread set for an **omp_get_num_places** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The **omp_get_num_places** routine returns the number of places in the place list. This value is equivalent to the number of places in the *place-partition-var* ICV in the execution environment of the initial task.

Cross References

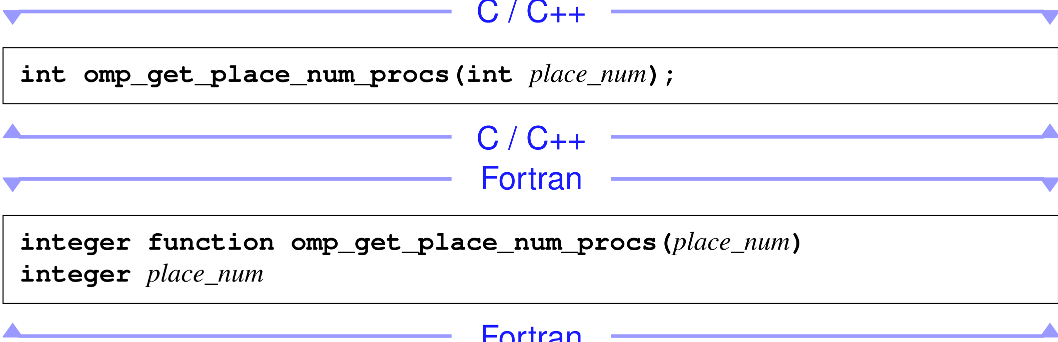
- *place-partition-var* ICV, see Section 2.4 on page 49.
- **OMP_PLACES** environment variable, see Section 5.5 on page 567.

3.2.24 omp_get_place_num_procs

Summary

The **omp_get_place_num_procs** routine returns the number of processors available to the execution environment in the specified place.

Format




```c
int omp_get_place_num_procs(int place_num);
```

```fortran
integer function omp_get_place_num_procs(place_num)
integer place_num
```

Binding

The binding thread set for an **omp_get_place_num_procs** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The **omp_get_place_num_procs** routine returns the number of processors associated with the place numbered *place_num*. The routine returns zero when *place_num* is negative, or is equal to or larger than the value returned by **omp_get_num_places()**.

Cross References

- **OMP_PLACES** environment variable, see Section 5.5 on page 567.

3.2.25 omp_get_place_proc_ids

Summary

The **omp_get_place_proc_ids** routine returns the numerical identifiers of the processors available to the execution environment in the specified place.

Format

C / C++

```
void omp_get_place_proc_ids(int place_num, int *ids);
```

C / C++

Fortran

```
subroutine omp_get_place_proc_ids(place_num, ids)
integer place_num
integer ids(*)
```

Fortran

Binding

The binding thread set for an **omp_get_place_proc_ids** region is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Effect

The **omp_get_place_proc_ids** routine returns the numerical identifiers of each processor associated with the place numbered *place_num*. The numerical identifiers are non-negative, and their meaning is implementation defined. The numerical identifiers are returned in the array *ids* and their order in the array is implementation defined. The array must be sufficiently large to contain **omp_get_place_num_procs(***place_num***)** integers; otherwise, the behavior is unspecified. The routine has no effect when *place_num* has a negative value, or a value equal or larger than **omp_get_num_places()**.

Cross References

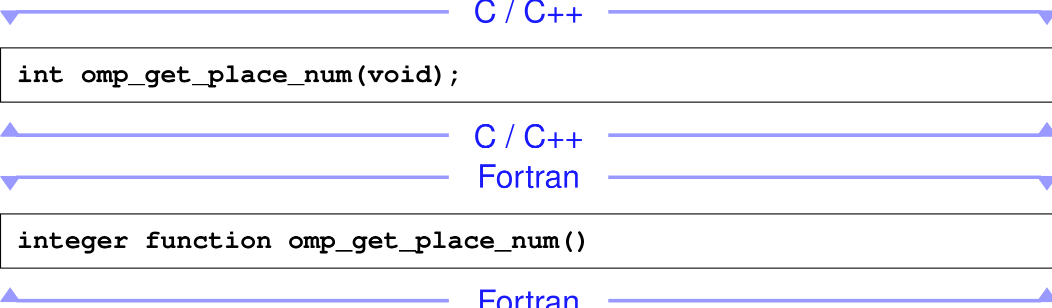
- **omp_get_place_num_procs** routine, see Section 3.2.24 on page 326.
- **omp_get_num_places** routine, see Section 3.2.23 on page 325.
- **OMP_PLACES** environment variable, see Section 5.5 on page 567.

3.2.26 omp_get_place_num

Summary

The **omp_get_place_num** routine returns the place number of the place to which the encountering thread is bound.

Format



C / C++

```
int omp_get_place_num(void);
```

C / C++

Fortran

```
integer function omp_get_place_num()
```

Fortran

Binding

The binding thread set for an **omp_get_place_num** region is the encountering thread.

Effect

When the encountering thread is bound to a place, the **omp_get_place_num** routine returns the place number associated with the thread. The returned value is between 0 and one less than the value returned by **omp_get_num_places()**, inclusive. When the encountering thread is not bound to a place, the routine returns -1.

Cross References

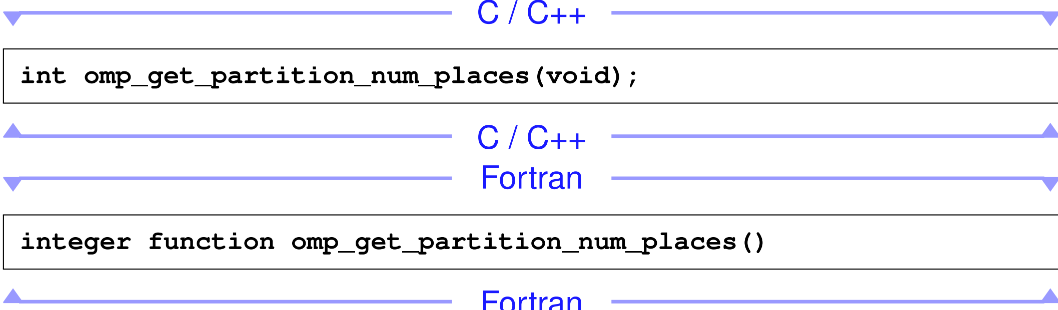
- Controlling OpenMP thread affinity, see Section 2.8.2 on page 73.
- **`omp_get_num_places`** routine, see Section 3.2.23 on page 325.
- **`OMP_PLACES`** environment variable, see Section 5.5 on page 567.

3.2.27 `omp_get_partition_num_places`

Summary

The **`omp_get_partition_num_places`** routine returns the number of places in the place partition of the innermost implicit task.

Format



C / C++

```c
int omp_get_partition_num_places(void);
```

C / C++

Fortran

```fortran
integer function omp_get_partition_num_places()
```

Fortran

Binding

The binding task set for an **`omp_get_partition_num_places`** region is the encountering implicit task.

Effect

The **`omp_get_partition_num_places`** routine returns the number of places in the *place-partition-var* ICV.

Cross References

- *place-partition-var* ICV, see Section 2.4 on page 49.
- Controlling OpenMP thread affinity, see Section 2.8.2 on page 73.
- **OMP_PLACES** environment variable, see Section 5.5 on page 567.

3.2.28 omp_get_partition_place_nums

Summary

The **omp_get_partition_place_nums** routine returns the list of place numbers corresponding to the places in the *place-partition-var* ICV of the innermost implicit task.

Format

C / C++

```
void omp_get_partition_place_nums(int *place_nums);
```

C / C++

Fortran

```
subroutine omp_get_partition_place_nums(place_nums)
integer place_nums(*)
```

Fortran

Binding

The binding task set for an **omp_get_partition_place_nums** region is the encountering implicit task.

Effect

The **omp_get_partition_place_nums** routine returns the list of place numbers corresponding to the places in the *place-partition-var* ICV of the innermost implicit task. The array must be sufficiently large to contain **omp_get_partition_num_places()** integers; otherwise, the behavior is unspecified.

Cross References

- *place-partition-var* ICV, see Section 2.4 on page 49.
- Controlling OpenMP thread affinity, see Section 2.8.2 on page 73.
- **omp_get_partition_num_places** routine, see Section 3.2.27 on page 329.
- **OMP_PLACES** environment variable, see Section 5.5 on page 567.

3.2.29 omp_set_affinity_format

Summary

The **omp_set_affinity_format** routine sets the affinity format to be used on the device by setting the value of the *affinity-format-var* ICV.

Format

C / C++

```
void omp_set_affinity_format(char const *format);
```

C / C++

Fortran

```
subroutine omp_set_affinity_format(format)
character(len=*),intent(in)::format
```

Fortran

Binding

When called from a sequential part of the program, the binding thread set for an **omp_set_affinity_format** region is the encountering thread. When called from within any explicit parallel region, the binding thread set (and binding region, if required) for the **omp_set_affinity_format** region is implementation defined.

Effect

The effect of **omp_set_affinity_format** routine is to set the value of the *affinity-format-var* ICV on the current device to the *format* specified in the argument.

This routine has the described effect only when called from a sequential part of the program. When called from within an explicit **parallel** region, the effect of this routine is implementation defined.

Cross References

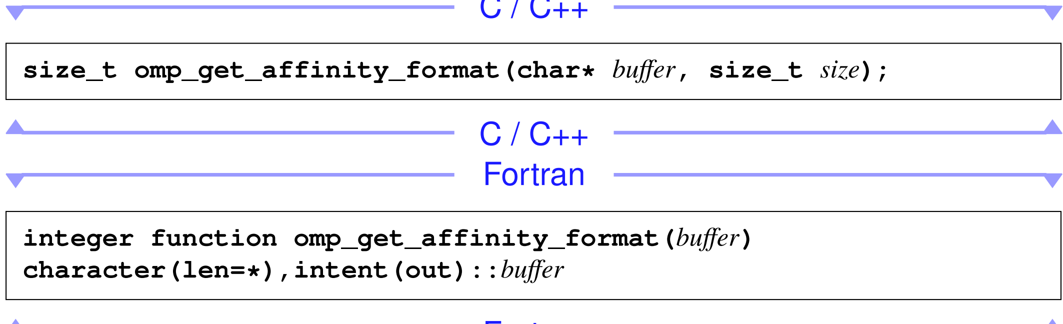
- Controlling OpenMP thread affinity, see Section 2.8.2 on page 73.
- **OMP_DISPLAY_AFFINITY** environment variable, see Section 5.13 on page 574.
- **OMP_AFFINITY_FORMAT** environment variable, see Section 5.14 on page 575.
- **omp_get_affinity_format** routine, see Section 3.2.30 on page 332.
- **omp_display_affinity** routine, see Section 3.2.31 on page 333.
- **omp_capture_affinity** routine, see Section 3.2.32 on page 334.

3.2.30 omp_get_affinity_format

Summary

The **omp_get_affinity_format** routine returns the value of the *affinity-format-var* ICV on the device.

Format



Binding

When called from a sequential part of the program, the binding thread set for an **omp_get_affinity_format** region is the encountering thread. When called from within any explicit **parallel** region, the binding thread set (and binding region, if required) for the **omp_get_affinity_format** region is implementation defined.

Effect

The **omp_get_affinity_format** routine returns the number of characters required to hold the entire affinity format specification and writes the value of the *affinity-format-var* ICV on the current device to *buffer*. If the return value is larger than *size* (for C/C++) or **len(***buffer***)** (for Fortran), the affinity format specification is truncated.

Cross References

- Controlling OpenMP thread affinity, see Section 2.8.2 on page 73.
- **OMP_DISPLAY_AFFINITY** environment variable, see Section 5.13 on page 574.
- **OMP_AFFINITY_FORMAT** environment variable, see Section 5.14 on page 575.
- **omp_set_affinity_format** routine, see Section 3.2.29 on page 331.
- **omp_display_affinity** routine, see Section 3.2.31 on page 333.
- **omp_capture_affinity** routine, see Section 3.2.32 on page 334.

3.2.31 omp_display_affinity

Summary

The **omp_display_affinity** routine prints the OpenMP thread affinity information using the format specification provided.

Format

C / C++

```
void omp_display_affinity(char const *format);
```

C / C++

Fortran

```
subroutine omp_display_affinity(format)
character(len=*),intent(in)::format
```

Fortran

1 ### Binding

2 The binding thread set for an **omp_display_affinity** region is the encountering thread.

3 ### Effect

4 The **omp_display_affinity** routine prints the thread affinity information of the current
5 thread in the format specified by the *format* argument. If the *format* is NULL (for C/C++) or a
6 zero-length string (for Fortran and C/C++), the value of the *affinity-format-var* ICV is used.

7 ### Cross References

8 • Controlling OpenMP thread affinity, see Section 2.8.2 on page 73.

9 • **OMP_DISPLAY_AFFINITY** environment variable, see Section 5.13 on page 574.

10 • **OMP_AFFINITY_FORMAT** environment variable, see Section 5.14 on page 575.

11 • **omp_set_affinity_format** routine, see Section 3.2.29 on page 331.

12 • **omp_get_affinity_format** routine, see Section 3.2.30 on page 332.

13 • **omp_capture_affinity** routine, see Section 3.2.32 on page 334.

14 ## 3.2.32 omp_capture_affinity

15 ### Summary

16 The **omp_capture_affinity** routine prints the OpenMP thread affinity information into a
17 buffer using the format specification provided.

1 **Format**

C / C++

```
size_t omp_capture_affinity(char *buffer, size_t size,
                            char const *format);
```

C / C++

Fortran

```
integer function omp_capture_affinity(buffer,format)
character(len=*),intent(out)::buffer
character(len=*),intent(in)::format
```

Fortran

2 **Binding**

3 The binding thread set for an **omp_capture_affinity** region is the encountering thread.

4 **Effect**

5 The **omp_capture_affinity** routine returns the number of characters required to hold the
6 entire affinity format specification and prints the thread affinity information of the current thread
7 into the character string **buffer** with the size of *size* (for C/C++) or **len(***buffer***)** (for Fortran) in
8 the format specified by the *format* argument. If the *format* is NULL (for C/C++) or a zero-length
9 string (for Fortran and C/C++), the value of the *affinity-format-var* ICV is used. The *buffer* must be
10 allocated prior to calling the routine. If the return value is larger than *size* (for C/C++) or
11 **len(***buffer***)** (for Fortran), the affinity format specification is truncated.

12 **Cross References**

13 • Controlling OpenMP thread affinity, see Section 2.8.2 on page 73.

14 • **OMP_DISPLAY_AFFINITY** environment variable, see Section 5.13 on page 574.

15 • **OMP_AFFINITY_FORMAT** environment variable, see Section 5.14 on page 575.

16 • **omp_set_affinity_format** routine, see Section 3.2.29 on page 331.

17 • **omp_get_affinity_format** routine, see Section 3.2.30 on page 332.

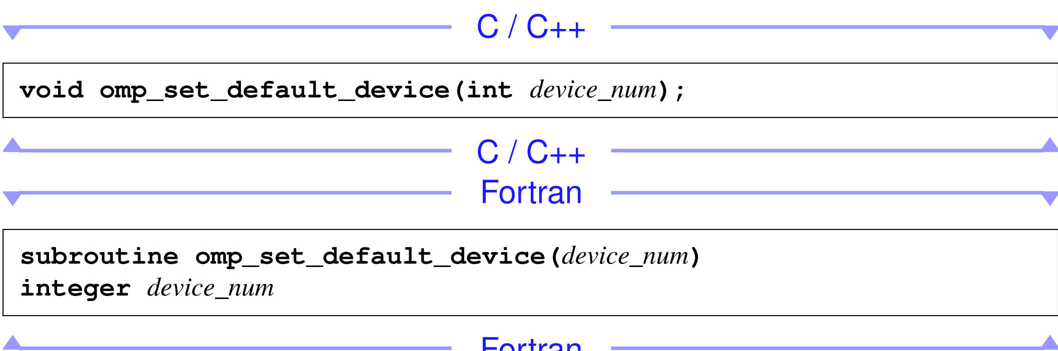
18 • **omp_display_affinity** routine, see Section 3.2.31 on page 333.

3.2.33 omp_set_default_device

Summary

The **omp_set_default_device** routine controls the default target device by assigning the value of the *default-device-var* ICV.

Format



C / C++

```
void omp_set_default_device(int device_num);
```

C / C++

Fortran

```
subroutine omp_set_default_device(device_num)
integer device_num
```

Fortran

Binding

The binding task set for an **omp_set_default_device** region is the generating task.

Effect

The effect of this routine is to set the value of the *default-device-var* ICV of the current task to the value specified in the argument. When called from within a **target** region the effect of this routine is unspecified.

Cross References

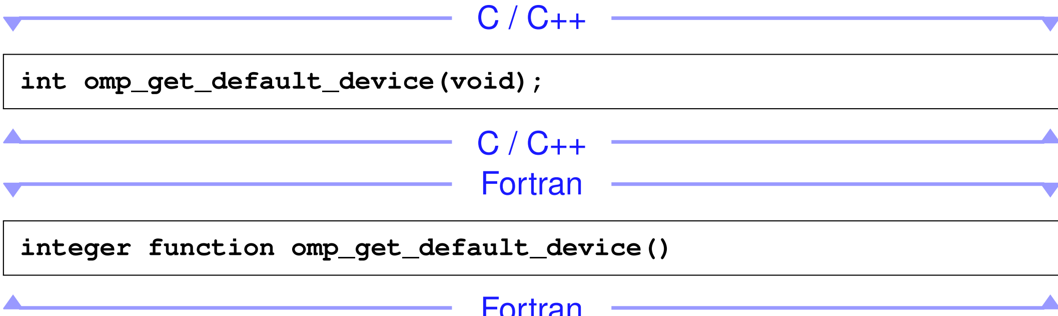
- *default-device-var*, see Section 2.4 on page 49.
- **omp_get_default_device**, see Section 3.2.34 on page 337.
- **OMP_DEFAULT_DEVICE** environment variable, see Section 5.15 on page 577

3.2.34 omp_get_default_device

Summary

The **omp_get_default_device** routine returns the default target device.

Format



C / C++

```
int omp_get_default_device(void);
```

C / C++

Fortran

```
integer function omp_get_default_device()
```

Fortran

Binding

The binding task set for an **omp_get_default_device** region is the generating task.

Effect

The **omp_get_default_device** routine returns the value of the *default-device-var* ICV of the current task. When called from within a **target** region the effect of this routine is unspecified.

Cross References

- *default-device-var*, see Section 2.4 on page 49.
- **omp_set_default_device**, see Section 3.2.33 on page 336.
- **OMP_DEFAULT_DEVICE** environment variable, see Section 5.15 on page 577.

3.2.35 omp_get_num_devices

Summary

The **omp_get_num_devices** routine returns the number of target devices.

Format

C / C++

```
int omp_get_num_devices(void);
```

C / C++

Fortran

```
integer function omp_get_num_devices()
```

Fortran

Binding

The binding task set for an **omp_get_num_devices** region is the generating task.

Effect

The **omp_get_num_devices** routine returns the number of available target devices. When called from within a **target** region the effect of this routine is unspecified.

Cross References

None.

3.2.36 omp_get_device_num

Summary

The **omp_get_device_num** routine returns the device number of the device on which the calling thread is executing.

Format

C / C++

```
int omp_get_device_num(void);
```

C / C++

Fortran

```
integer function omp_get_device_num()
```

Fortran

1 **Binding**

2 The binding task set for an **omp_get_devices_num** region is the generating task.

3 **Effect**

4 The **omp_get_device_num** routine returns the device number of the device on which the
5 calling thread is executing. When called on the host device, it will return the same value as the
6 **omp_get_initial_device** routine.

7 **Cross References**

8 • **omp_get_initial_device** routine, see Section 3.2.40 on page 342.

9 3.2.37 omp_get_num_teams

10 **Summary**

11 The **omp_get_num_teams** routine returns the number of initial teams in the current **teams**
12 region.

13 **Format**

C / C++

```
int omp_get_num_teams(void);
```

C / C++

Fortran

```
integer function omp_get_num_teams()
```

Fortran

Binding

The binding task set for an **omp_get_num_teams** region is the generating task

Effect

The effect of this routine is to return the number of initial teams in the current **teams** region. The routine returns 1 if it is called from outside of a **teams** region.

Cross References

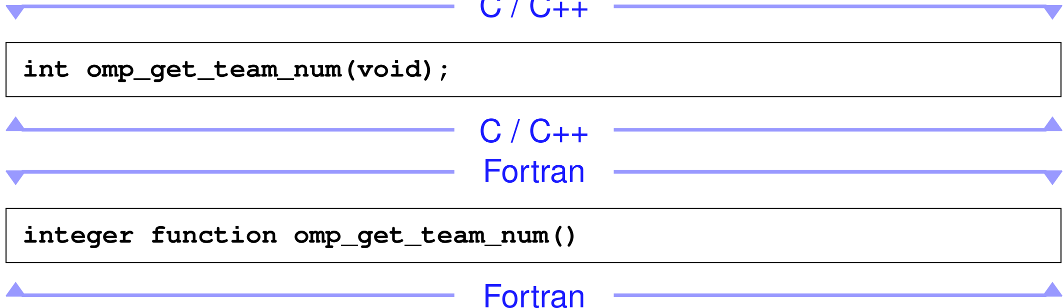
- **teams** construct, see Section 2.15.9 on page 157.

3.2.38 omp_get_team_num

Summary

The **omp_get_team_num** routine returns the initial team number of the calling thread.

Format



C / C++

```
int omp_get_team_num(void);
```

C / C++

Fortran

```
integer function omp_get_team_num()
```

Fortran

Binding

The binding task set for an **omp_get_team_num** region is the generating task.

Effect

The **omp_get_team_num** routine returns the initial team number of the calling thread. The initial team number is an integer between 0 and one less than the value returned by **omp_get_num_teams()**, inclusive. The routine returns 0 if it is called outside of a **teams** region.

Cross References

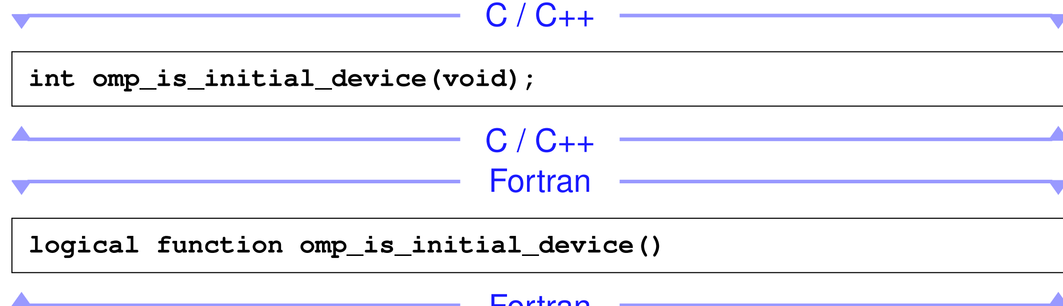
- **teams** construct, see Section 2.15.9 on page 157.
- **omp_get_num_teams** routine, see Section 3.2.37 on page 339.

3.2.39 omp_is_initial_device

Summary

The **omp_is_initial_device** routine returns *true* if the current task is executing on the host device; otherwise, it returns *false*.

Format



C / C++

```
int omp_is_initial_device(void);
```

C / C++

Fortran

```
logical function omp_is_initial_device()
```

Fortran

Binding

The binding task set for an **omp_is_initial_device** region is the generating task.

Effect

The effect of this routine is to return *true* if the current task is executing on the host device; otherwise, it returns *false*.

Cross References

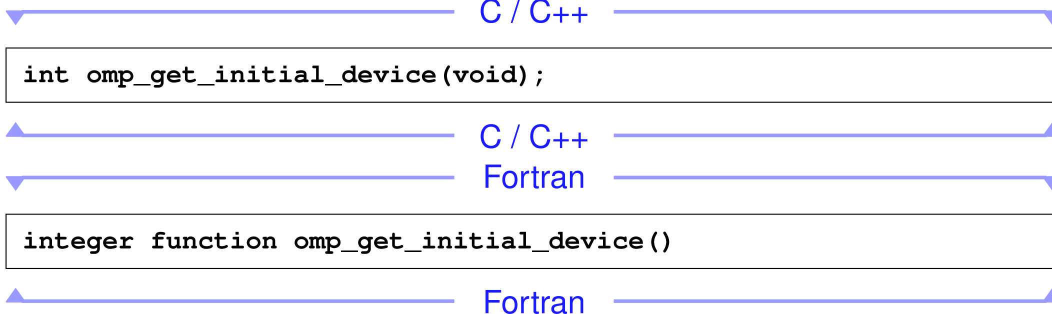
- **target** construct, see Section 2.15.5 on page 141

3.2.40 omp_get_initial_device

Summary

The **omp_get_initial_device** routine returns a device number representing the host device.

Format



Binding

The binding task set for an **omp_get_initial_device** region is the generating task.

Effect

The effect of this routine is to return the device number of the host device. The value of the device number is implementation defined. If it is between 0 and one less than **omp_get_num_devices()** then it is valid for use with all device constructs and routines; if it is outside that range, then it is only valid for use with the device memory routines and not in the **device** clause. When called from within a **target** region the effect of this routine is unspecified.

Cross References

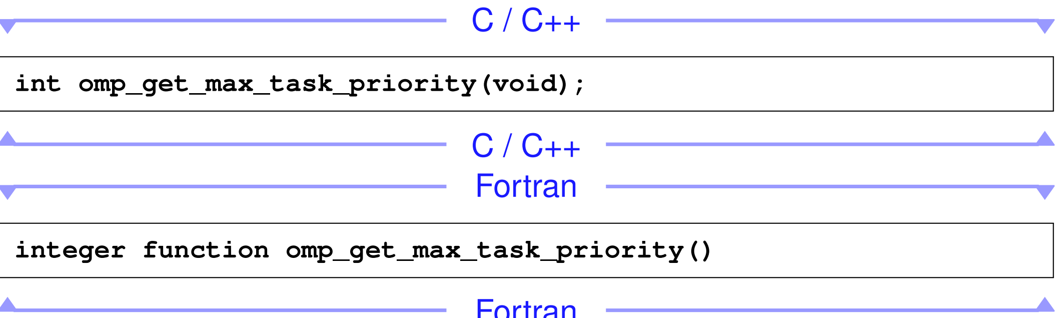
- **target** construct, see Section 2.15.5 on page 141
- Device memory routines, see Section 3.5 on page 358.

1 ### 3.2.41 omp_get_max_task_priority

2 #### Summary

3 The **omp_get_max_task_priority** routine returns the maximum value that can be specified
4 in the **priority** clause.

5 #### Format



6 #### Binding

7 The binding thread set for an **omp_get_max_task_priority** region is all threads on the
8 device. The effect of executing this routine is not related to any specific region corresponding to
9 any construct or API routine.

10 #### Effect

11 The **omp_get_max_task_priority** routine returns the value of the *max-task-priority-var*
12 ICV, which determines the maximum value that can be specified in the **priority** clause.

13 #### Cross References

14 • *max-task-priority-var*, see Section 2.4 on page 49.

15 • **task** construct, see Section 2.13.1 on page 110.

3.3 Lock Routines

The OpenMP runtime library includes a set of general-purpose lock routines that can be used for synchronization. These general-purpose lock routines operate on OpenMP locks that are represented by OpenMP lock variables. OpenMP lock variables must be accessed only through the routines described in this section; programs that otherwise access OpenMP lock variables are non-conforming.

An OpenMP lock can be in one of the following states: *uninitialized*, *unlocked*, or *locked*. If a lock is in the *unlocked* state, a task can *set* the lock, which changes its state to *locked*. The task that sets the lock is then said to *own* the lock. A task that owns a lock can *unset* that lock, returning it to the *unlocked* state. A program in which a task unsets a lock that is owned by another task is non-conforming.

Two types of locks are supported: *simple locks* and *nestable locks*. A *nestable lock* can be set multiple times by the same task before being unset; a *simple lock* cannot be set if it is already owned by the task trying to set it. *Simple lock* variables are associated with *simple locks* and can only be passed to *simple lock* routines. *Nestable lock* variables are associated with *nestable locks* and can only be passed to *nestable lock* routines.

Each type of lock can also have a *synchronization hint* that contains information about the intended usage of the lock by the application code. The effect of the hint is implementation defined. An OpenMP implementation can use this hint to select a usage-specific lock, but hints do not change the mutual exclusion semantics of locks. A conforming implementation can safely ignore the hint.

Constraints on the state and ownership of the lock accessed by each of the lock routines are described with the routine. If these constraints are not met, the behavior of the routine is unspecified.

The OpenMP lock routines access a lock variable such that they always read and update the most current value of the lock variable. It is not necessary for an OpenMP program to include explicit **flush** directives to ensure that the lock variable's value is consistent among different tasks.

Binding

The binding thread set for all lock routine regions is all threads in the contention group. As a consequence, for each OpenMP lock, the lock routine effects relate to all tasks that call the routines, without regard to which teams the threads in the contention group executing the tasks belong.

Simple Lock Routines

C / C++

The type **omp_lock_t** represents a simple lock. For the following routines, a simple lock variable must be of **omp_lock_t** type. All simple lock routines require an argument that is a pointer to a variable of type **omp_lock_t**.

C / C++

Fortran

For the following routines, a simple lock variable must be an integer variable of **kind=omp_lock_kind**.

Fortran

The simple lock routines are as follows:

- The **omp_init_lock** routine initializes a simple lock.
- The **omp_init_lock_with_hint** routine initializes a simple lock and attaches a hint to it.
- The **omp_destroy_lock** routine uninitializes a simple lock.
- The **omp_set_lock** routine waits until a simple lock is available, and then sets it.
- The **omp_unset_lock** routine unsets a simple lock.
- The **omp_test_lock** routine tests a simple lock, and sets it if it is available.

Nestable Lock Routines

C / C++

The type **omp_nest_lock_t** represents a nestable lock. For the following routines, a nestable lock variable must be of **omp_nest_lock_t** type. All nestable lock routines require an argument that is a pointer to a variable of type **omp_nest_lock_t**.

C / C++

Fortran

For the following routines, a nestable lock variable must be an integer variable of **kind=omp_nest_lock_kind**.

Fortran

The nestable lock routines are as follows:

- The **omp_init_nest_lock** routine initializes a nestable lock.
- The **omp_init_nest_lock_with_hint** routine initializes a nestable lock and attaches a hint to it.
- The **omp_destroy_nest_lock** routine uninitializes a nestable lock.
- The **omp_set_nest_lock** routine waits until a nestable lock is available, and then sets it.
- The **omp_unset_nest_lock** routine unsets a nestable lock.
- The **omp_test_nest_lock** routine tests a nestable lock, and sets it if it is available

Restrictions

OpenMP lock routines have the following restrictions:

- The use of the same OpenMP lock in different contention groups results in unspecified behavior.

3.3.1 omp_init_lock and omp_init_nest_lock

Summary

These routines initialize an OpenMP lock without a hint.

Format

C / C++

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

C / C++

Fortran

```
subroutine omp_init_lock(svar)
integer (kind=omp_lock_kind) svar

subroutine omp_init_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Fortran

Constraints on Arguments

A program that accesses a lock that is not in the uninitialized state through either routine is non-conforming.

Effect

The effect of these routines is to initialize the lock to the unlocked state; that is, no task owns the lock. In addition, the nesting count for a nestable lock is set to zero.

Events

The *lock-init* or *nest-lock-init* event occurs in the thread executing a **omp_init_lock** or **omp_init_nest_lock** region after initialization of the lock, but before finishing the region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_lock_init** callback for each occurrence of a *lock-init* or *nest-lock-init* event in that thread. This callback has the type signature **ompt_callback_mutex_acquire_t**. The callbacks occur in the task encountering the routine. The callback receives **omp_sync_hint_none** as *hint* argument and **ompt_mutex_lock** or **ompt_mutex_nest_lock** as *kind* argument as appropriate.

Cross References

- **ompt_callback_mutex_acquire_t**, see Section 4.1.4.2.13 on page 419.

3.3.2 omp_init_lock_with_hint and omp_init_nest_lock_with_hint

Summary

These routines initialize an OpenMP lock with a hint. The effect of the hint is implementation-defined. The OpenMP implementation can ignore the hint without changing program semantics.

Format

C / C++

```
void omp_init_lock_with_hint(omp_lock_t *lock,
                             omp_sync_hint_t hint);
void omp_init_nest_lock_with_hint(omp_nest_lock_t *lock,
                                  omp_sync_hint_t hint);
```

C / C++

Fortran

```
subroutine omp_init_lock_with_hint(svar, hint)
integer (kind=omp_lock_kind) svar
integer (kind=omp_sync_hint_kind) hint

subroutine omp_init_nest_lock_with_hint(nvar, hint)
integer (kind=omp_nest_lock_kind) nvar
integer (kind=omp_sync_hint_kind) hint
```

Fortran

Constraints on Arguments

A program that accesses a lock that is not in the uninitialized state through either routine is non-conforming.

The second argument passed to these routines (*hint*) is a hint as described in Section 2.18.11 on page 229.

Effect

The effect of these routines is to initialize the lock to the unlocked state and, optionally, to choose a specific lock implementation based on the hint. After initialization no task owns the lock. In addition, the nesting count for a nestable lock is set to zero.

Events

The *lock-init* or *nest-lock-init* event occurs in the thread executing a **omp_init_lock_with_hint** or **omp_init_nest_lock_with_hint** region after initialization of the lock, but before finishing the region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_lock_init** callback for each occurrence of a *lock-init* or *nest-lock-init* event in that thread. This callback has the type signature **ompt_callback_mutex_acquire_t**. The callbacks occur in the task encountering the routine. The callback receives the function's *hint* argument as *hint* argument and **ompt_mutex_lock** or **ompt_mutex_nest_lock** as *kind* argument as appropriate.

Cross References

- `ompt_callback_mutex_acquire_t`, see Section 4.1.4.2.13 on page 419.
- Synchronization Hints, see Section 2.18.11 on page 229.

3.3.3 `omp_destroy_lock` and `omp_destroy_nest_lock`

Summary

These routines ensure that the OpenMP lock is uninitialized.

Format

C / C++

```
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

C / C++

Fortran

```
subroutine omp_destroy_lock(svar)
integer (kind=omp_lock_kind) svar

subroutine omp_destroy_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Fortran

Constraints on Arguments

A program that accesses a lock that is not in the unlocked state through either routine is non-conforming.

Effect

The effect of these routines is to change the state of the lock to uninitialized.

Events

The *lock-destroy* or *nest-lock-destroy* event occurs in the thread executing a **omp_destroy_lock** or **omp_destroy_nest_lock** region before finishing the region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_lock_destroy** callback for each occurrence of a *lock-destroy* or *nest-lock-destroy* event in that thread. This callback has the type signature **ompt_callback_mutex_t**. The callbacks occur in the task encountering the routine. The callbacks receive **ompt_mutex_lock** or **ompt_mutex_nest_lock** as their *kind* argument as appropriate.

Cross References

- **ompt_callback_mutex_t**, see Section 4.1.4.2.14 on page 420.

3.3.4 omp_set_lock and omp_set_nest_lock

Summary

These routines provide a means of setting an OpenMP lock. The calling task region behaves as if it was suspended until the lock can be set by this task.

Format

C / C++

```
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

C / C++

Fortran

```
subroutine omp_set_lock(svar)
integer (kind=omp_lock_kind) svar

subroutine omp_set_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Fortran

Constraints on Arguments

A program that accesses a lock that is in the uninitialized state through either routine is non-conforming. A simple lock accessed by **omp_set_lock** that is in the locked state must not be owned by the task that contains the call or deadlock will result.

Effect

Each of these routines has an effect equivalent to suspension of the task executing the routine until the specified lock is available.

Note – The semantics of these routines is specified *as if* they serialize execution of the region guarded by the lock. However, implementations may implement them in other ways provided that the isolation properties are respected so that the actual execution delivers a result that could arise from some serialization.

A simple lock is available if it is unlocked. Ownership of the lock is granted to the task executing the routine.

A nestable lock is available if it is unlocked or if it is already owned by the task executing the routine. The task executing the routine is granted, or retains, ownership of the lock, and the nesting count for the lock is incremented.

Events

The *lock-acquire* or *nest-lock-acquire* event occurs in the thread executing a **omp_set_lock** or **omp_set_nest_lock** region before the associated lock is requested.

The *lock-acquired* or *nest-lock-acquired* event occurs in the thread executing a **omp_set_lock** or **omp_set_nest_lock** region after acquiring the associated lock, if the thread did not already own the lock, but before finishing the region.

The *nest-lock-owned* event occurs in the thread executing a **omp_set_nest_lock** region when the thread already owned the lock, before finishing the region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_mutex_acquire** callback for each occurrence of a *lock-acquire* or *nest-lock-acquire* event in that thread. This callback has the type signature **ompt_callback_mutex_acquire_t**.

A thread dispatches a registered **ompt_callback_mutex_acquired** callback for each occurrence of a *lock-acquired* or *nest-lock-acquired* event in that thread. This callback has the type signature **ompt_callback_mutex_t**.

A thread dispatches a registered **ompt_callback_nest_lock** callback for each occurrence of a *nest-lock-owned* event in that thread. This callback has the type signature **ompt_callback_nest_lock_t**. The callback receives **ompt_scope_begin** as its *endpoint* argument.

The callbacks occur in the task encountering the lock function. The callbacks receive **ompt_mutex_lock** or **ompt_mutex_nest_lock** as their *kind* argument, as appropriate.

Cross References

- **ompt_callback_mutex_acquire_t**, see Section 4.1.4.2.13 on page 419.
- **ompt_callback_mutex_t**, see Section 4.1.4.2.14 on page 420.
- **ompt_callback_nest_lock_t**, see Section 4.1.4.2.15 on page 421.

3.3.5 omp_unset_lock and omp_unset_nest_lock

Summary

These routines provide the means of unsetting an OpenMP lock.

Format

C / C++

```
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

C / C++

Fortran

```
subroutine omp_unset_lock(svar)
integer (kind=omp_lock_kind) svar

subroutine omp_unset_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Fortran

Constraints on Arguments

A program that accesses a lock that is not in the locked state or that is not owned by the task that contains the call through either routine is non-conforming.

Effect

For a simple lock, the **omp_unset_lock** routine causes the lock to become unlocked.

For a nestable lock, the **omp_unset_nest_lock** routine decrements the nesting count, and causes the lock to become unlocked if the resulting nesting count is zero.

For either routine, if the lock becomes unlocked, and if one or more task regions were effectively suspended because the lock was unavailable, the effect is that one task is chosen and given ownership of the lock.

Events

The *lock-release* or *nest-lock-release* event occurs in the thread executing a **omp_unset_lock** or **omp_unset_nest_lock** region after releasing the associated lock, but before finishing the region.

The *nest-lock-held* event occurs in the thread executing a **omp_unset_nest_lock** region when the thread still owns the lock, before finishing the region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_mutex_released** callback for each occurrence of a *lock-release* or *nest-lock-release* event in that thread. This callback has the type signature **ompt_callback_mutex_t**. The callback occurs in the task encountering the routine. The callback receives **ompt_mutex_lock** or **ompt_mutex_nest_lock** as *kind* argument as appropriate.

A thread dispatches a registered **ompt_callback_nest_lock** callback for each occurrence of a *nest-lock-held* event in that thread. This callback has the type signature **ompt_callback_nest_lock_t**. The callback receives **ompt_scope_end** as its *endpoint* argument.

Cross References

- **ompt_callback_mutex_t**, see Section 4.1.4.2.14 on page 420.
- **ompt_callback_nest_lock_t**, see Section 4.1.4.2.15 on page 421.

3.3.6 omp_test_lock and omp_test_nest_lock

Summary

These routines attempt to set an OpenMP lock but do not suspend execution of the task executing the routine.

Format

C / C++

```
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

C / C++

Fortran

```
logical function omp_test_lock(svar)
integer (kind=omp_lock_kind) svar
integer function omp_test_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Fortran

Constraints on Arguments

A program that accesses a lock that is in the uninitialized state through either routine is non-conforming. The behavior is unspecified if a simple lock accessed by **omp_test_lock** is in the locked state and is owned by the task that contains the call.

Effect

These routines attempt to set a lock in the same manner as **omp_set_lock** and **omp_set_nest_lock**, except that they do not suspend execution of the task executing the routine.

For a simple lock, the **omp_test_lock** routine returns *true* if the lock is successfully set; otherwise, it returns *false*.

For a nestable lock, the **omp_test_nest_lock** routine returns the new nesting count if the lock is successfully set; otherwise, it returns zero.

Events

The *lock-test* or *nest-lock-test* event occurs in the thread executing a **omp_test_lock** or **omp_test_nest_lock** region before the associated lock is tested.

The *lock-test-acquired* or *nest-lock-test-acquired* event occurs in the thread executing a **omp_test_lock** or **omp_test_nest_lock** region before finishing the region if the associated lock was acquired and the thread did not already own the lock.

The *nest-lock-owned* event occurs in the thread executing a **omp_test_nest_lock** region if the thread already owned the lock, before finishing the region.

Tool Callbacks

A thread dispatches a registered **ompt_callback_mutex_acquire** callback for each occurrence of a *lock-test* or *nest-lock-test* event in that thread. This callback has the type signature **ompt_callback_mutex_acquire_t**.

A thread dispatches a registered **ompt_callback_mutex_acquired** callback for each occurrence of a *lock-test-acquired* or *nest-lock-test-acquired* event in that thread. This callback has the type signature **ompt_callback_mutex_t**.

A thread dispatches a registered **ompt_callback_nest_lock** callback for each occurrence of a *nest-lock-owned* event in that thread. This callback has the type signature **ompt_callback_nest_lock_t**. The callback receives **ompt_scope_begin** as its *endpoint* argument.

The callbacks occur in the task encountering the lock function. The callbacks receive **ompt_mutex_lock** or **ompt_mutex_nest_lock** as their *kind* argument, as appropriate.

Cross References

- **ompt_callback_mutex_acquire_t**, see Section 4.1.4.2.13 on page 419.
- **ompt_callback_mutex_t**, see Section 4.1.4.2.14 on page 420.
- **ompt_callback_nest_lock_t**, see Section 4.1.4.2.15 on page 421.

3.4 Timing Routines

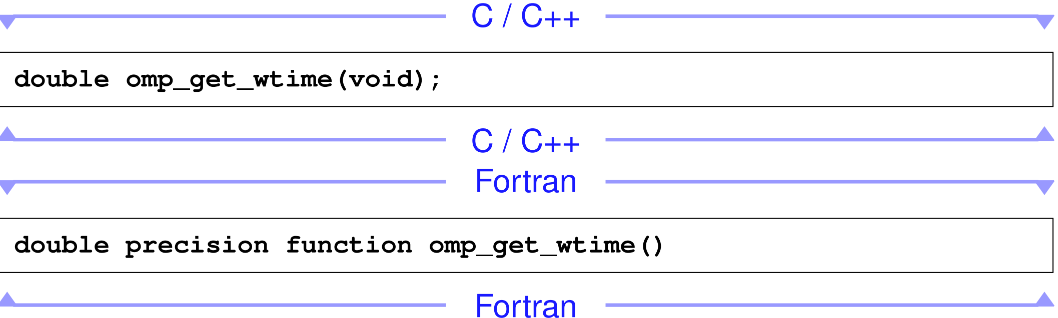
This section describes routines that support a portable wall clock timer.

3.4.1 omp_get_wtime

Summary

The **omp_get_wtime** routine returns elapsed wall clock time in seconds.

Format



Binding

The binding thread set for an **omp_get_wtime** region is the encountering thread. The routine's return value is not guaranteed to be consistent across any set of threads.

1 **Effect**

2 3 4 5 The **omp_get_wtime** routine returns a value equal to the elapsed wall clock time in seconds since some "time in the past". The actual "time in the past" is arbitrary, but it is guaranteed not to change during the execution of the application program. The time returned is a "per-thread time", so it is not required to be globally consistent across all threads participating in an application.

6 7 Note – It is anticipated that the routine will be used to measure elapsed times as shown in the following example:

C / C++

```
double start;
double end;
start = omp_get_wtime();
... work to be timed ...
end = omp_get_wtime();
printf("Work took %f seconds\n", end - start);
```

C / C++

Fortran

```
DOUBLE PRECISION START, END
START = omp_get_wtime()
... work to be timed ...
END = omp_get_wtime()
PRINT *, "Work took", END - START, "seconds"
```

Fortran

8 3.4.2 omp_get_wtick

9 **Summary**

10 The **omp_get_wtick** routine returns the precision of the timer used by **omp_get_wtime**.

1 **Format**

C / C++

```
double omp_get_wtick(void);
```

C / C++

Fortran

```
double precision function omp_get_wtick()
```

Fortran

2 **Binding**

3 The binding thread set for an **omp_get_wtick** region is the encountering thread. The routine's
4 return value is not guaranteed to be consistent across any set of threads.

5 **Effect**

6 The **omp_get_wtick** routine returns a value equal to the number of seconds between successive
7 clock ticks of the timer used by **omp_get_wtime**.

C / C++

8 # 3.5 Device Memory Routines

9 This section describes routines that support allocation of memory and management of pointers in
10 the data environments of target devices.

11 ## 3.5.1 omp_target_alloc

12 **Summary**

13 The **omp_target_alloc** routine allocates memory in a device data environment.

Format

```
void* omp_target_alloc(size_t size, int device_num);
```

Effect

The **omp_target_alloc** routine returns the device address of a storage location of *size* bytes. The storage location is dynamically allocated in the device data environment of the device specified by *device_num*, which must be greater than or equal to zero and less than the result of **omp_get_num_devices()** or the result of a call to **omp_get_initial_device()**. When called from within a **target** region the effect of this routine is unspecified.

The **omp_target_alloc** routine returns **NULL** if it cannot dynamically allocate the memory in the device data environment.

The device address returned by **omp_target_alloc** can be used in an **is_device_ptr** clause, Section 2.15.5 on page 141.

Pointer arithmetic is not supported on the device address returned by **omp_target_alloc**.

Freeing the storage returned by **omp_target_alloc** with any routine other than **omp_target_free** results in unspecified behavior.

Events

The *target-data-allocation* event occurs when a thread allocates data on a target device.

Tool Callbacks

A thread invokes a registered **ompt_callback_target_data_op** callback for each occurrence of a *target-data-allocation* event in that thread. The callback occurs in the context of the target task. The callback has type signature **ompt_callback_target_data_op_t**.

Cross References

- **target** construct, see Section 2.15.5 on page 141
- **omp_get_num_devices** routine, see Section 3.2.35 on page 337
- **omp_get_initial_device** routine, see Section 3.2.40 on page 342
- **omp_target_free** routine, see Section 3.5.2 on page 360
- **ompt_callback_target_data_op_t**, see Section 4.1.4.2.21 on page 428.

3.5.2 `omp_target_free`

Summary

The **`omp_target_free`** routine frees the device memory allocated by the **`omp_target_alloc`** routine.

Format

```
void omp_target_free(void * device_ptr, int device_num);
```

Constraints on Arguments

A program that calls **`omp_target_free`** with a non-**`NULL`** pointer that does not have a value returned from **`omp_target_alloc`** is non-conforming. The *device_num* must be greater than or equal to zero and less than the result of **`omp_get_num_devices()`** or the result of a call to **`omp_get_initial_device()`**.

Effect

The **`omp_target_free`** routine frees the memory in the device data environment associated with *device_ptr*. If *device_ptr* is **`NULL`**, the operation is ignored.

Synchronization must be inserted to ensure that all accesses to *device_ptr* are completed before the call to **`omp_target_free`**.

When called from within a **`target`** region the effect of this routine is unspecified.

Events

The *target-data-free* event occurs when a thread frees data on a target device.

Tool Callbacks

A thread invokes a registered **`ompt_callback_target_data_op`** callback for each occurrence of a *target-data-free* event in that thread. The callback occurs in the context of the target task. The callback has type signature **`ompt_callback_target_data_op_t`**.

Cross References

- **target** construct, see Section 2.15.5 on page 141
- **omp_get_num_devices** routine, see Section 3.2.35 on page 337
- **omp_get_initial_device** routine, see Section 3.2.40 on page 342
- **omp_target_alloc** routine, see Section 3.5.1 on page 358
- **ompt_callback_target_data_op_t**, see Section 4.1.4.2.21 on page 428.

3.5.3 omp_target_is_present

Summary

The **omp_target_is_present** routine tests whether a host pointer has corresponding storage on a given device.

Format

```
int omp_target_is_present(const void * ptr, int device_num);
```

Constraints on Arguments

The value of *ptr* must be a valid host pointer or **NULL**. The *device_num* must be greater than or equal to zero and less than the result of **omp_get_num_devices()** or the result of a call to **omp_get_initial_device()**.

Effect

This routine returns non-zero if the specified pointer would be found present on device *device_num* by a **map** clause; otherwise, it returns zero.

When called from within a **target** region the effect of this routine is unspecified.

Cross References

- **`target`** construct, see Section 2.15.5 on page 141
- **`map`** clause, see Section 2.20.6.1 on page 280.
- **`omp_get_num_devices`** routine, see Section 3.2.35 on page 337
- **`omp_get_initial_device`** routine, see Section 3.2.40 on page 342

3.5.4 omp_target_memcpy

Summary

The **`omp_target_memcpy`** routine copies memory between any combination of host and device pointers.

Format

```
int omp_target_memcpy(void * dst, const void * src,
                      size_t length, size_t dst_offset,
                      size_t src_offset, int dst_device_num,
                      int src_device_num);
```

Constraints on Arguments

Each device must be compatible with the device pointer specified on the same side of the copy. The *dst_device_num* and *src_device_num* must be greater than or equal to zero and less than the result of **`omp_get_num_devices()`** or equal to the result of a call to **`omp_get_initial_device()`**.

Effect

length bytes of memory at offset *src_offset* from *src* in the device data environment of device *src_device_num* are copied to *dst* starting at offset *dst_offset* in the device data environment of device *dst_device_num*. The return value is zero on success and non-zero on failure. The host device and host device data environment can be referenced with the device number returned by **`omp_get_initial_device`**. This routine contains a task scheduling point.

When called from within a **`target`** region the effect of this routine is unspecified.

Events

The *target-data-op* event occurs when a thread transfers data on a target device.

Tool Callbacks

A thread invokes a registered **ompt_callback_target_data_op** callback for each occurrence of a *target-data-op* event in that thread. The callback occurs in the context of the target task. The callback has type signature **ompt_callback_target_data_op_t**.

Cross References

- **target** construct, see Section 2.15.5 on page 141
- **omp_get_initial_device** routine, see Section 3.2.40 on page 342
- **omp_target_alloc** routine, see Section 3.5.1 on page 358
- **ompt_callback_target_data_op_t**, see Section 4.1.4.2.21 on page 428.

3.5.5 omp_target_memcpy_rect

Summary

The **omp_target_memcpy_rect** routine copies a rectangular subvolume from a multi-dimensional array to another multi-dimensional array. The copies can use any combination of host and device pointers.

Format

```
int omp_target_memcpy_rect(
               void * dst, const void * src,
               size_t element_size,
               int num_dims,
               const size_t* volume,
               const size_t* dst_offsets,
               const size_t* src_offsets,
               const size_t* dst_dimensions,
               const size_t* src_dimensions,
               int dst_device_num, int src_device_num);
```

Constraints on Arguments

The length of the offset and dimension arrays must be at least the value of *num_dims*. The **dst_device_num** and **src_device_num** must be greater than or equal to zero and less than the result of **omp_get_num_devices()** or equal to the result of a call to **omp_get_initial_device()**.

The value of *num_dims* must be between 1 and the implementation-defined limit, which must be at least three.

Effect

This routine copies a rectangular subvolume of *src*, in the device data environment of device *src_device_num*, to *dst*, in the device data environment of device *dst_device_num*. The volume is specified in terms of the size of an element, number of dimensions, and constant arrays of length *num_dims*. The maximum number of dimensions supported is at least three, support for higher dimensionality is implementation defined. The volume array specifies the length, in number of elements, to copy in each dimension from *src* to *dst*. The *dst_offsets* (*src_offsets*) parameter specifies number of elements from the origin of *dst* (*src*) in elements. The *dst_dimensions* (*src_dimensions*) parameter specifies the length of each dimension of *dst* (*src*)

The routine returns zero if successful. If both *dst* and *src* are **NULL** pointers, the routine returns the number of dimensions supported by the implementation for the specified device numbers. The host device and host device data environment can be referenced with the device number returned by **omp_get_initial_device**. Otherwise, it returns a non-zero value. The routine contains a task scheduling point.

When called from within a **target** region the effect of this routine is unspecified.

Events

The *target-data-op* event occurs when a thread transfers data on a target device.

Tool Callbacks

A thread invokes a registered **ompt_callback_target_data_op** callback for each occurrence of a *target-data-op* event in that thread. The callback occurs in the context of the target task. The callback has type signature **ompt_callback_target_data_op_t**.

Cross References

- **target** construct, see Section 2.15.5 on page 141
- **omp_get_initial_device** routine, see Section 3.2.40 on page 342
- **omp_target_alloc** routine, see Section 3.5.1 on page 358
- **ompt_callback_target_data_op_t**, see Section 4.1.4.2.21 on page 428.

3.5.6 omp_target_associate_ptr

Summary

The **omp_target_associate_ptr** routine maps a device pointer, which may be returned from **omp_target_alloc** or implementation-defined runtime routines, to a host pointer.

Format

```
int omp_target_associate_ptr(const void * host_ptr,
                             const void * device_ptr,
                             size_t size, size_t device_offset,
                             int device_num);
```

Constraints on Arguments

The value of *device_ptr* value must be a valid pointer to device memory for the device denoted by the value of *device_num*. The *device_num* argument must be greater than or equal to zero and less than the result of **omp_get_num_devices()** or equal to the result of a call to **omp_get_initial_device()**.

Effect

The **omp_target_associate_ptr** routine associates a device pointer in the device data environment of device *device_num* with a host pointer such that when the host pointer appears in a subsequent **map** clause, the associated device pointer is used as the target for data motion associated with that host pointer. The *device_offset* parameter specifies what offset into *device_ptr* will be used as the base address for the device side of the mapping. The reference count of the resulting mapping will be infinite. After being successfully associated, the buffer pointed to by the device pointer is invalidated and accessing data directly through the device pointer results in unspecified behavior. The pointer can be retrieved for other uses by disassociating it. When called from within a **target** region the effect of this routine is unspecified.

The routine returns zero if successful. Otherwise it returns a non-zero value.

Only one device buffer can be associated with a given host pointer value and device number pair. Attempting to associate a second buffer will return non-zero. Associating the same pair of pointers on the same device with the same offset has no effect and returns zero. Associating pointers that share underlying storage will result in unspecified behavior. The **omp_target_is_present** region can be used to test whether a given host pointer has a corresponding variable in the device data environment.

Events

The *target-data-associate* event occurs when a thread associates data on a target device.

Tool Callbacks

A thread invokes a registered **ompt_callback_target_data_op** callback for each occurrence of a *target-data-associate* event in that thread. The callback occurs in the context of the target task. The callback has type signature **ompt_callback_target_data_op_t**.

Cross References

- **target** construct, see Section 2.15.5 on page 141
- **map** clause, see Section 2.20.6.1 on page 280.
- **omp_target_alloc** routine, see Section 3.5.1 on page 358
- **omp_target_disassociate_ptr** routine, see Section 3.5.6 on page 365
- **ompt_callback_target_data_op_t**, see Section 4.1.4.2.21 on page 428.

3.5.7 omp_target_disassociate_ptr

Summary

The **omp_target_disassociate_ptr** removes the associated pointer for a given device from a host pointer.

Format

```
int omp_target_disassociate_ptr(const void * ptr, int device_num);
```

Constraints on Arguments

The *device_num* must be greater than or equal to zero and less than the result of **omp_get_num_devices()** or equal to the result of a call to **omp_get_initial_device()**.

Effect

The **omp_target_disassociate_ptr** removes the associated device data on device *device_num* from the presence table for host pointer *ptr*. A call to this routine on a pointer that is not **NULL** and does not have associated data on the given device results in unspecified behavior. The reference count of the mapping is reduced to zero, regardless of its current value.

When called from within a **target** region the effect of this routine is unspecified.

After a call to **omp_target_disassociate_ptr**, the contents of the device buffer are invalidated.

Events

The *target-data-disassociate* event occurs when a thread disassociates data on a target device.

Tool Callbacks

A thread invokes a registered **ompt_callback_target_data_op** callback for each occurrence of a *target-data-disassociate* event in that thread. The callback occurs in the context of the target task. The callback has type signature **ompt_callback_target_data_op_t**.

Cross References

- **target** construct, see Section 2.15.5 on page 141
- **omp_target_associate_ptr** routine, see Section 3.5.6 on page 365
- **ompt_callback_target_data_op_t**, see Section 4.1.4.2.21 on page 428.

C / C++

3.6 Memory Management Routines

This section describes routines that support memory management on the current device.

Instances of memory management types must be accessed only through the routines described in this section; programs that otherwise access instances of these types are non-conforming.

3.6.1 Memory Management Types

The following type definitions are used by the memory management routines:

C / C++

```
omp_allocator_t;

enum { OMP_NULL_ALLOCATOR = NULL };
```

C / C++

Fortran

```
integer parameter omp_allocator_kind

integer(kind=omp_allocator_kind), &
        parameter :: omp_null_allocator = 0
```

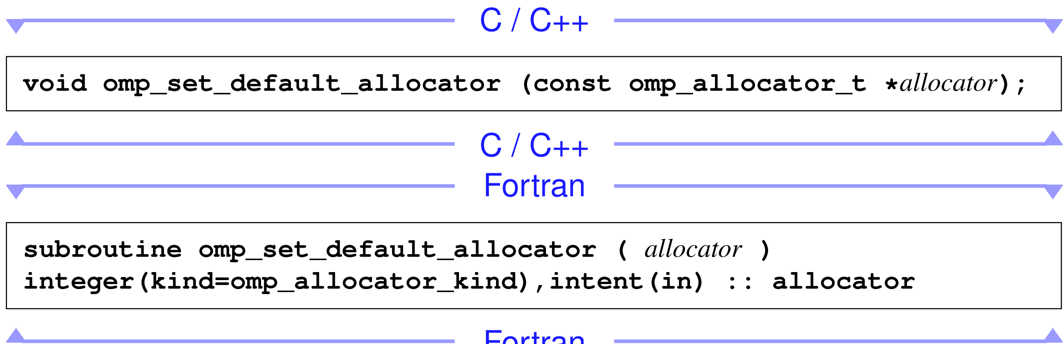
Fortran

3.6.2 omp_set_default_allocator

Summary

The **omp_set_default_allocator** routine sets the default memory allocator to be used by allocation calls, **allocate** directives and **allocate** clauses that do not specify an allocator.

Format



C / C++

```
void omp_set_default_allocator (const omp_allocator_t *allocator);
```

C / C++

Fortran

```
subroutine omp_set_default_allocator ( allocator )
integer(kind=omp_allocator_kind),intent(in) :: allocator
```

Fortran

Constraints on Arguments

The *allocator* argument must point to a valid memory allocator.

Binding

The binding task set for an **omp_set_default_allocator** region is the *binding implicit task*.

Effect

The effect of this routine is to set the value of the *def-allocator-var* ICV of the *binding implicit task* to the value specified in the *allocator* argument.

Cross References

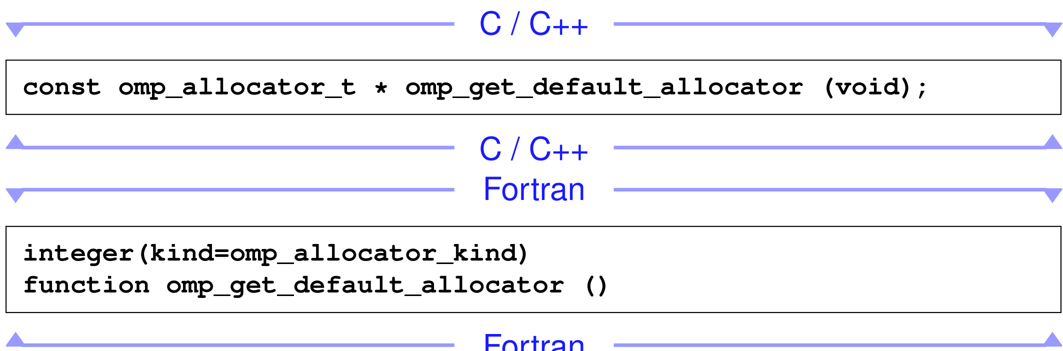
- *def-allocator-var* ICV, see Section 2.4 on page 49.
- Memory Allocators, see Section 2.7 on page 64.
- **omp_alloc** routine, see Section 3.6.4 on page 371.

3.6.3 omp_get_default_allocator

Summary

The **omp_get_default_allocator** routine returns the memory allocator to be used by allocation calls, **allocate** directives and **allocate** clauses that do not specify an allocator.

Format



C / C++

```
const omp_allocator_t * omp_get_default_allocator (void);
```

C / C++

Fortran

```
integer(kind=omp_allocator_kind)
function omp_get_default_allocator ()
```

Fortran

Binding

The binding task set for an **omp_get_default_allocator** region is the *binding implicit task*.

Effect

The effect of this routine is to return the value of the *def-allocator-var* ICV of the *binding implicit task*.

Cross References

- *def-allocator-var* ICV, see Section 2.4 on page 49.
- Memory Allocators, see Section 2.7 on page 64.
- **omp_alloc** routine, see Section 3.6.4 on page 371.

C / C++

1 ### 3.6.4 omp_alloc

2 #### Summary

3 The **omp_alloc** routine requests a memory allocation from a memory allocator.

4 #### Format

C

```
void * omp_alloc (size_t size, const omp_allocator_t *allocator);
```

C

C++

```
void * omp_alloc (size_t size,
          const omp_allocator_t *allocator=OMP_NULL_ALLOCATOR);
```

C++

5 #### Constraints on Arguments

6 For **omp_alloc** invocations appearing in **target** regions the *allocator* argument cannot be
7 **OMP_NULL_ALLOCATOR** and it must be a constant expression.

8 #### Effect

9 The **omp_alloc** routine requests a memory allocation of *size* bytes from the specified memory
10 allocator. If the *allocator* argument is **OMP_NULL_ALLOCATOR** the memory allocator used by the
11 routine will be the one specified by the *def-allocator-var* ICV of the *binding implicit task*. Upon
12 success it returns a pointer to the allocated memory. Otherwise, it returns **NULL**.

13 #### Cross References

14 • Memory allocators, see Section 2.7 on page 64.

3.6.5 omp_free

Summary

The **omp_free** routine deallocates previously allocated memory.

Format

C

```
void omp_free ( void * ptr, const omp_allocator_t *allocator);
```

C

C++

```
void omp_free ( void * ptr,
                const omp_allocator_t *allocator=OMP_NULL_ALLOCATOR);
```

C++

Effect

The **omp_free** routine deallocates the memory to which *ptr* points. The *ptr* argument must point to memory previously allocated with a memory allocator. If the *allocator* argument is specified it must be the memory allocator to which the allocation request was made. If the *allocator* argument is **OMP_NULL_ALLOCATOR** the implementation will find the memory allocator used to allocate the memory. Using **omp_free** on memory that was already deallocated results in unspecified behavior.

C / C++

3.7 Tool Control Routines

Summary

The **omp_control_tool** routine enables a program to pass commands to an active tool.

1 **Format**

C / C++

```
int omp_control_tool(int command, int modifier, void *arg);
```

C / C++

Fortran

```
integer function omp_control_tool(command, modifier)
integer (kind=omp_control_tool_kind) command
integer (kind=omp_control_tool_kind) modifier
```

Fortran

2 **Description**

3 An OpenMP program may use **omp_control_tool** to pass commands to a tool. Using
4 **omp_control_tool**, an application can request that a tool start or restart data collection when a
5 code region of interest is encountered, pause data collection when leaving the region of interest,
6 flush any data that it has collected so far, or end data collection. Additionally,
7 **omp_control_tool** can be used to pass tool-specific commands to a particular tool.

C / C++

```
typedef enum omp_control_tool_result_t {
  omp_control_tool_notool = -2,
  omp_control_tool_nocallback = -1,
  omp_control_tool_success = 0,
  omp_control_tool_ignored = 1
} omp_control_tool_result_t;
```

C / C++

Fortran

```
integer (kind=omp_control_tool_result_kind), &
        parameter :: omp_control_tool_notool = -2
integer (kind=omp_control_tool_result_kind), &
        parameter :: omp_control_tool_nocallback = -1
integer (kind=omp_control_tool_result_kind), &
        parameter :: omp_control_tool_success = 0
integer (kind=omp_control_tool_result_kind), &
        parameter :: omp_control_tool_ignored = 1
```

Fortran

If no tool is active, the OpenMP implementation will return **omp_control_tool_notool**. If a tool is active, but it has not registered a callback for the *tool-control* event, the OpenMP implementation will return **omp_control_tool_nocallback**. An OpenMP implementation may return other implementation-defined negative values < -64; an application may assume that any negative return value indicates that a tool has not received the command. A return value of **omp_control_tool_success** indicates that the tool has performed the specified command. A return value of **omp_control_tool_ignored** indicates that the tool has ignored the specified command. A tool may return other positive values > 64 that are tool-defined.

Constraints on Arguments

The following enumeration type defines four standard commands. Table 3.1 describes the actions that these commands request from a tool.

C / C++

```
typedef enum omp_control_tool_t {
  omp_control_tool_start = 1,
  omp_control_tool_pause = 2,
  omp_control_tool_flush = 3,
  omp_control_tool_end = 4
} omp_control_tool_t;
```

C / C++

Fortran

```
integer (kind=omp_control_tool_kind), &
          parameter :: omp_control_tool_start = 1
integer (kind=omp_control_tool_kind), &
          parameter :: omp_control_tool_pause = 2
integer (kind=omp_control_tool_kind), &
          parameter :: omp_control_tool_flush = 3
integer (kind=omp_control_tool_kind), &
          parameter :: omp_control_tool_end = 4
```

Fortran

1 Tool-specific values for *command* must be ≥ 64. Tools must ignore *command* values that they are
2 not explicitly designed to handle. Other values accepted by a tool for *command*, and any values for
3 *modifier* and *arg* are tool-defined.

TABLE 3.1: Standard tool control commands.

Command	Action
omp_control_tool_start	Start or restart monitoring if it is off. If monitoring is already on, this command is idempotent. If monitoring has already been turned off permanently, this command will have no effect.
omp_control_tool_pause	Temporarily turn monitoring off. If monitoring is already off, it is idempotent.
omp_control_tool_flush	Flush any data buffered by a tool. This command may be applied whether monitoring is on or off.
omp_control_tool_end	Turn monitoring off permanently; the tool finalizes itself and flushes all output.

4 **Events**

5 The *tool-control* event occurs in the thread encountering a call to **omp_control_tool** at a point
6 inside its associated OpenMP region.

7 **Tool Callbacks**

8 A thread dispatches a registered **ompt_callback_control_tool** callback for each
9 occurrence of a *tool-control* event. The callback executes in the context of the call that occurs in the
10 user program. This callback has type signature **ompt_callback_control_tool_t**.The

callback may return any non-negative value, which will be returned to the application by the OpenMP implementation as the return value of the **omp_control_tool** call that triggered the callback.

Arguments passed to the callback are those passed by the user to **omp_control_tool**. If the call is made in Fortran, the tool will be passed a **NULL** as the third argument to the callback. If any of the four standard commands is presented to a tool, the tool will ignore the *modifier* and *arg* argument values.

Cross References

- Tool Interface, see Chapter 4 on page 377
- **ompt_callback_control_tool_t**, see Section 4.1.4.2.26 on page 436

CHAPTER 4

Tool Support

This chapter describes OMPT and OMPD, which are a pair of interfaces for first-party and third-party tools, respectively. Section 4.1 describes OMPT—an interface for first-party tools. The section begins with a description of how to initialize (Section 4.1.1) and finalize (Sections 4.1.2) a tool. Subsequent sections describe details of the interface, including data types shared between an OpenMP implementation and a tool (Section 4.1.3), an interface that enables an OpenMP implementation to determine that a tool is available (Section 4.1.1.1), type signatures for tool callbacks that an OpenMP implementation may dispatch for OpenMP events (Section 4.1.4), and *runtime entry points*—function interfaces provided by an OpenMP implementation for use by a tool (Section 4.1.5).

Section 4.2 describes OMPD—an interface for third-party tools such as debuggers. Unlike OMPT, a third-party tool exists in a separate process from the OpenMP program. An OpenMP implementation need not maintain any extra information to support OMPD inquiries from third-party tools *unless* it is explicitly instructed to do so. Section 4.2.1 discusses the mechanisms for activating support for OMPD in the OpenMP runtime. Section 4.2.2 describes the data types shared between the OMPD plugin and a third-party tool. Section 4.1.5.3.1 describes the API provided by the OMPD plugin for use by a third-party tool. An OMPD implementation will not interact directly with the OpenMP runtime for which it is designed to operate. Instead, the third-party tool must provide the OMPD with a set of callbacks that the OMPD uses to access the OpenMP runtime. This interface is given in Section 4.1.5.3.1. In general, a third-party's tool's OpenMP-related activity will be conducted through the OMPD interface. However, there are a few instances where the third-party tool needs to access the OpenMP runtime directly; these cases are discussed in Section 4.2.5.

4.1 OMPT

The OMPT interface defines mechanisms for initializing a tool, exploring the details of an OpenMP implementation, examining OpenMP state associated with an OpenMP thread, interpreting an OpenMP thread's call stack, receiving notification about OpenMP *events*, tracing activity on OpenMP target devices, and controlling a tool from an OpenMP application.

4.1.1 Activating an OMPT Tool

There are three steps to activating a tool. First, an OpenMP implementation determines whether a tool should be initialized. If so, the OpenMP implementation invokes the tool's initializer, enabling the tool to prepare to monitor the execution on the host. Finally, a tool may arrange to monitor computation that execute on target devices. This section explains how the tool and an OpenMP implementation interact to accomplish these tasks.

4.1.1.1 Determining Whether an OMPT Tool Should be Initialized

A tool indicates its interest in using the OMPT interface by providing a non-**NULL** pointer to an **ompt_start_tool_result_t** structure to an OpenMP implementation as a return value from **ompt_start_tool**. There are three ways that a tool can provide a definition of **ompt_start_tool** to an OpenMP implementation:

- statically-linking the tool's definition of **ompt_start_tool** into an OpenMP application,
- introducing a dynamically-linked library that includes the tool's definition of **ompt_start_tool** into the application's address space, or
- 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.

Immediately before an OpenMP implementation initializes itself, it determines whether it should check for the presence of a tool interested in using the OMPT interface by examining the *tool-var* ICV. If value of *tool-var* is *disabled*, the OpenMP implementation will initialize itself without even checking whether a tool is present and the functionality of the OMPT interface will be unavailable as the program executes.

If the value of *tool-var* is *enabled*, the OpenMP implementation will check to see if a tool has provided an implementation of **ompt_start_tool**. The OpenMP implementation first checks if a tool-provided implementation of **ompt_start_tool** is available in the address space, either statically-linked into the application or in a dynamically-linked library loaded in the address space.

If multiple implementations of **ompt_start_tool** are available, the OpenMP implementation will use the first tool-provided implementation of **ompt_start_tool** found.

If no tool-provided implementation of **ompt_start_tool** is found in the address space, the OpenMP implementation will consult the *tool-libraries-var* ICV, which contains a (possibly empty) list of dynamically-linked libraries. As described in detail in Section 5.19, the libraries in *tool-libraries-var*, will be searched for the first usable implementation of **ompt_start_tool** provided by one of the libraries in the list.

If a tool-provided definition of **ompt_start_tool** is found using either method, the OpenMP implementation will invoke it; if it returns a non-**NULL** pointer to an **ompt_start_tool_result_t** structure, the OpenMP implementation will know that a tool expects to use the OMPT interface.

Next, the OpenMP implementation will initialize itself. If a tool provided a non-**NULL** pointer to an **ompt_start_tool_result_t** structure, the OpenMP runtime will prepare itself for use of the OMPT interface by a tool.

Cross References

- *tool-libraries-var* ICV, see Section 2.4 on page 49.
- *tool-var* ICV, see Section 2.4 on page 49.
- **ompt_start_tool_result_t**, see Section 4.1.3.1 on page 388.
- **ompt_start_tool**, see Section 4.3.2.1 on page 558.

4.1.1.2 Initializing an OMPT 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*.

A tool's initializer, described in Section 4.1.4.1.1 on page 404 uses its argument *lookup* to look up pointers to OMPT interface runtime entry points provided by the OpenMP implementation; this process is described in Section 4.1.1.2.1 on page 380. Typically, a tool initializer will first obtain a pointer to the OpenMP runtime entry point known as **ompt_set_callback** with type signature **ompt_set_callback_t** and then use this runtime entry point to register tool callbacks for OpenMP events, as described in Section 4.1.1.3 on page 381.

A tool initializer may use the OMPT interface runtime entry points known as **ompt_enumerate_states** and **ompt_enumerate_mutex_impls**, which have type signatures **ompt_enumerate_states_t** and **ompt_enumerate_mutex_impls_t**, to determine what thread states and implementations of mutual exclusion a particular OpenMP implementation employs.

If a tool initializer returns a non-zero value, the tool will be *activated* for the execution; otherwise, the tool will be inactive.

Cross References

- **ompt_start_tool_result_t**, see Section 4.1.3.1 on page 388.
- **ompt_start_tool**, see Section 4.3.2.1 on page 558.
- **ompt_initialize_t**, see Section 4.1.4.1.1 on page 404.
- **ompt_callback_thread_begin_t**, see Section 4.1.4.2.1 on page 406.
- **ompt_enumerate_states_t**, see Section 4.1.5.1.1 on page 441.
- **ompt_enumerate_mutex_impls_t**, see Section 4.1.5.1.2 on page 443.
- **ompt_set_callback_t**, see Section 4.1.5.1.3 on page 445.
- **ompt_function_lookup_t**, see Section 4.1.5.3.1 on page 475.

4.1.1.2.1 Binding Entry Points in the OMPT Callback Interface

Functions that an OpenMP implementation provides to support the OMPT interface are not defined as global function symbols. Instead, they are defined as runtime entry points that a tool can only identify using the *lookup* function provided as an argument to the tool's initializer. This design avoids tool implementations that will fail in certain circumstances when functions defined as part of the OpenMP runtime are not visible to a tool, even though the tool and the OpenMP runtime are both present in the same address space. It also prevents inadvertent use of a tool support routine by applications.

A tool's initializer receives a function pointer to a *lookup* runtime entry point with type signature **ompt_function_lookup_t** as its first argument. Using this function, a tool initializer may obtain a pointer to each of the runtime entry points that an OpenMP implementation provides to support the OMPT interface. Once a tool has obtained a *lookup* function, it may employ it at any point in the future.

For each runtime entry point in the OMPT interface for the host device, Table 4.1 provides the string name by which it is known and its associated type signature. Implementations can provide additional, implementation specific names and corresponding entry points as long as they don't use names that start with the prefix "**ompt_**". These are reserved for future extensions in the OpenMP specification.

During initialization, a tool should look up each runtime entry point in the OMPT interface by name and bind a pointer maintained by the tool that it can use later to invoke the entry point as needed. The entry points described in Table 4.1 enable a tool to assess what thread states and mutual exclusion implementations that an OpenMP runtime supports, register tool callbacks,

inspect callbacks registered, introspect OpenMP state associated with threads, and use tracing to monitor computations that execute on target devices.

Detailed information about each runtime entry point listed in Table 4.1 is included as part of the description of its type signature.

Cross References

- **ompt_enumerate_states_t**, see Section 4.1.5.1.1 on page 441.
- **ompt_enumerate_mutex_impls_t**, see Section 4.1.5.1.2 on page 443.
- **ompt_set_callback_t**, see Section 4.1.5.1.3 on page 445.
- **ompt_get_callback_t**, see Section 4.1.5.1.4 on page 447.
- **ompt_get_thread_data_t**, see Section 4.1.5.1.5 on page 448.
- **ompt_get_num_places_t**, see Section 4.1.5.1.7 on page 449.
- **ompt_get_place_proc_ids_t**, see Section 4.1.5.1.8 on page 450.
- **ompt_get_place_num_t**, see Section 4.1.5.1.9 on page 452.
- **ompt_get_partition_place_nums_t**, see Section 4.1.5.1.10 on page 452.
- **ompt_get_procid_t**, see Section 4.1.5.1.11 on page 454.
- **ompt_get_state_t**, see Section 4.1.5.1.12 on page 454.
- **ompt_get_parallel_info_t**, see Section 4.1.5.1.13 on page 455.
- **ompt_get_task_info_t**, see Section 4.1.5.1.14 on page 457.
- **ompt_get_target_info_t**, see Section 4.1.5.1.15 on page 459.
- **ompt_get_num_devices_t**, see Section 4.1.5.1.16 on page 460.
- **ompt_get_num_procs_t**, see Section 4.1.5.1.6 on page 449.
- **ompt_get_unique_id_t**, see Section 4.1.5.1.17 on page 461.
- **ompt_function_lookup_t**, see Section 4.1.5.3.1 on page 475.

4.1.1.3 Monitoring Activity on the Host with OMPT

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**. The possible return codes for **ompt_set_callback** and their meanings are shown in Table 4.4.

TABLE 4.1: OMPT callback interface runtime entry point names and their type signatures.

Entry Point String Name	Type signature
"ompt_enumerate_states"	ompt_enumerate_states_t
"ompt_enumerate_mutex_impls"	ompt_enumerate_mutex_impls_t
"ompt_set_callback"	ompt_set_callback_t
"ompt_get_callback"	ompt_get_callback_t
"ompt_get_thread_data"	ompt_get_thread_data_t
"ompt_get_num_places"	ompt_get_num_places_t
"ompt_get_place_proc_ids"	ompt_get_place_proc_ids_t
"ompt_get_place_num"	ompt_get_place_num_t
"ompt_get_partition_place_nums"	ompt_get_partition_place_nums_t
"ompt_get_proc_id"	ompt_get_proc_id_t
"ompt_get_state"	ompt_get_state_t
"ompt_get_parallel_info"	ompt_get_parallel_info_t
"ompt_get_task_info"	ompt_get_task_info_t
"ompt_get_num_devices"	ompt_get_num_devices_t
"ompt_get_num_procs"	ompt_get_num_procs_t
"ompt_get_target_info"	ompt_get_target_info_t
"ompt_get_unique_id"	ompt_get_unique_id_t

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 a precise type signatures to set and get the callback for each unique runtime entry point type signature.

Table 4.2 indicates the return codes permissible when trying to register various callbacks. For callbacks where the only registration return code allowed is **ompt_set_always**, an OpenMP implementation must guarantee that the callback will be invoked every time a runtime event associated with it occurs. Support for such callbacks is required in a minimal implementation of the OMPT interface. For other callbacks where registration is allowed to return values other than **ompt_set_always**, its implementation-defined whether an OpenMP implementation invokes a registered callback never, sometimes, or always. If registration for a callback allows a return code of **omp_set_never**, support for invoking such a callback need not be present in a minimal implementation of the OMPT interface. The return code when a callback is registered enables a tool to know what to expect when the level of support for the callback can be implementation defined.

To avoid a tool interface specification that enables a tool to register unique callbacks for an overwhelming number of events, the interface was collapsed in several ways. First, in cases where events are naturally paired, e.g., the beginning and end of a region, and the arguments needed by the callback at each endpoint were identical, the pair of events was collapsed so that a tool registers a single callback that will be invoked at both endpoints with **ompt_scope_begin** or **ompt_scope_end** provided as an argument to identify which endpoint the callback invocation reflects. Second, when a whole class of events is amenable to uniform treatment, only a single callback is provided for a family of events, e.g., a **ompt_callback_sync_region_wait** callback is used for multiple kinds of synchronization regions, i.e., barrier, taskwait, and taskgroup regions. Some events involve both kinds of collapsing: the aforementioned **ompt_callback_sync_region_wait** represents a callback that will be invoked at each endpoint for different kinds of synchronization regions.

Cross References

- **ompt_set_callback_t**, see Section 4.1.5.1.3 on page 445.
- **ompt_get_callback_t**, see Section 4.1.5.1.4 on page 447.

4.1.1.4 Tracing Activity on Target Devices with OMPT

A target device may or may not initialize a full OpenMP runtime system. Unless it does, it may not be possible to monitor activity on a device using a tool interface based on callbacks. To accommodate such cases, the OMPT interface defines a monitoring interface for tracing activity on target devices. Tracing activity on a target device involves the following steps:

TABLE 4.2: Valid return codes of **ompt_set_callback** for each callback.

Return code abbreviation	N	S/P	A
ompt_callback_thread_begin			*
ompt_callback_thread_end			*
ompt_callback_parallel_begin			*
ompt_callback_parallel_end			*
ompt_callback_task_create			*
ompt_callback_task_schedule			*
ompt_callback_implicit_task			*
ompt_callback_target			*
ompt_callback_target_data_op			*
ompt_callback_target_submit			*
ompt_callback_control_tool			*
ompt_callback_device_initialize			*
ompt_callback_device_finalize			*
ompt_callback_device_load			*
ompt_callback_device_unload			*
ompt_callback_sync_region_wait	*	*	*
ompt_callback_mutex_released	*	*	*
ompt_callback_task_dependences	*	*	*
ompt_callback_task_dependence	*	*	*
ompt_callback_work	*	*	*
ompt_callback_master	*	*	*
ompt_callback_target_map	*	*	*
ompt_callback_sync_region	*	*	*
ompt_callback_lock_init	*	*	*
ompt_callback_lock_destroy	*	*	*
ompt_callback_mutex_acquire	*	*	*
ompt_callback_mutex_acquired	*	*	*
ompt_callback_nest_lock	*	*	*
ompt_callback_flush	*	*	*
ompt_callback_cancel	*	*	*
ompt_callback_idle	*	*	*

N = **ompt_set_never**

S = **ompt_set_sometimes**

P = **ompt_set_sometimes_paired**

A = **ompt_set_always**

- To prepare to trace activity on a target device, a tool must register for an **ompt_callback_device_initialize** callback. A tool may also register for an **ompt_callback_device_load** callback to be notified when code is loaded onto a target device or an **ompt_callback_device_unload** callback to be notified when code is unloaded from a target device. A tool may also optionally register an **ompt_callback_device_finalize** callback.

- When an OpenMP implementation initializes a target device, the OpenMP implementation will dispatch the tool's device initialization callback on the host device. If the OpenMP implementation or target device does not support tracing, the OpenMP implementation will pass a **NULL** to the tool's device initializer for its *lookup* argument; otherwise, the OpenMP implementation will pass a pointer to a device-specific runtime entry point with type signature **ompt_function_lookup_t** to the tool's device initializer.

- If the device initializer for the tool receives a non-**NULL** *lookup* pointer, the tool may use it to query which runtime entry points in the tracing interface are available for a target device and bind the function pointers returned to tool variables. Table 4.3 indicates the names of the runtime entry points that a target device may provide for use by a tool. Implementations can provide additional, implementation specific names and corresponding entry points as long as they don't use names that start with the prefix "**ompt_**". Theses are reserved for future extensions in the OpenMP specification.

  If *lookup* is non-**NULL**, the driver for a device will provide runtime entry points that enable a tool to control the device's interface for collecting traces in its *native* trace format, which may be device specific. The kinds of trace records available for a device will typically be implementation-defined. Some devices may also allow a tool to collect traces of records in a standard format known as OMPT format, described in this document. If so, the *lookup* function will return values for the runtime entry points **ompt_set_trace_ompt** and **ompt_get_record_ompt**, which support collecting and decoding OMPT traces. These runtime entry points are not required for all devices and will only be available for target devices that support collection of standard traces in OMPT format. For some devices, their native tracing format may be OMPT format. In that case, tracing can be controlled using either the runtime entry points for native or OMPT tracing.

- The tool will use the **ompt_set_trace_native** and/or the **ompt_set_trace_ompt** runtime entry point to specify what types of events or activities to monitor on the target device. If the **ompt_set_trace_native** and/or the **ompt_set_trace_ompt** runtime entry point is called outside a device initializer, registration of supported callbacks may fail with a return code of **ompt_set_error**.

- The tool will initiate tracing on the target device by invoking **ompt_start_trace**. Arguments to **ompt_start_trace** include two tool callbacks for use by the OpenMP implementation to manage traces associated with the target device: one to allocate a buffer where the target device can deposit trace events and a second to process a buffer of trace events from the target device.

TABLE 4.3: OMPT tracing interface runtime entry point names and their type signatures.

Entry Point String Name	Type Signature
"**ompt_get_device_num_procs**"	**ompt_get_device_num_procs_t**
"**ompt_get_device_time**"	**ompt_get_device_time_t**
"**ompt_translate_time**"	**ompt_translate_time_t**
"**ompt_set_trace_ompt**"	**ompt_set_trace_ompt_t**
"**ompt_set_trace_native**"	**ompt_set_trace_native_t**
"**ompt_start_trace**"	**ompt_start_trace_t**
"**ompt_pause_trace**"	**ompt_pause_trace_t**
"**ompt_stop_trace**"	**ompt_stop_trace_t**
"**ompt_advance_buffer_cursor**"	**ompt_advance_buffer_cursor_t**
"**ompt_get_record_type**"	**ompt_get_record_type_t**
"**ompt_get_record_ompt**"	**ompt_get_record_ompt_t**
"**ompt_get_record_native**"	**ompt_get_record_native_t**
"**ompt_get_record_abstract**"	**ompt_get_record_abstract_t**

- When the target device needs a trace buffer, the OpenMP implementation will invoke the tool-supplied callback function on the host device to request a new buffer.
- The OpenMP implementation will monitor execution of OpenMP constructs on the target device as directed and record a trace of events or activities into a trace buffer. If the device is capable, device trace records will be marked with a *host_op_id*—an identifier used to associate device activities with the target operation initiated on the host that caused these activities. To correlate activities on the host with activities on a device, a tool can register a **ompt_callback_target_submit** callback. Before the host initiates each distinct activity associated with a structured block for a **target** construct on a target device, the OpenMP implementation will dispatch the **ompt_callback_target_submit** callback on the host in the thread executing the task that encounters the **target** construct. Examples of activities that could cause an **ompt_callback_target_submit** callback to be dispatched include an explicit data copy between a host and target device or execution of a computation. This callback provides the tool with a pair of identifiers: one that identifies the target region and a second that uniquely identifies an activity associated with that region. These identifiers help the tool correlate activities on the target device with their target region.
- When appropriate, e.g., when a trace buffer fills or needs to be flushed, the OpenMP implementation will invoke the tool-supplied buffer completion callback to process a non-empty sequence of records in a trace buffer associated with the target device.

- The tool-supplied buffer completion callback may return immediately, ignoring records in the trace buffer, or it may iterate through them using the **ompt_advance_buffer_cursor** entry point and inspect each one. A tool may inspect the type of the record at the current cursor position using the **ompt_get_record_type** runtime entry point. A tool may choose to inspect the contents of some or all records in a trace buffer using the **ompt_get_record_ompt**, **ompt_get_record_native**, or **ompt_get_record_abstract** runtime entry point. Presumably, a tool that chooses to use the **ompt_get_record_native** runtime entry point to inspect records will have some knowledge about a device's native trace format. A tool may always use the **ompt_get_record_abstract** runtime entry point to inspect a trace record; this runtime entry point will decode the contents of a native trace record and summarize them in a standard format, namely, a **ompt_record_abstract_t** record. Only a record in OMPT format can be retrieved using the **ompt_get_record_ompt** runtime entry point.
- Once tracing has been started on a device, a tool may pause or resume tracing on the device at any time by invoking **ompt_pause_trace** with an appropriate flag value as an argument.
- A tool may start or stop tracing on a device at any time using the **ompt_start_trace** or **ompt_stop_trace** runtime entry points, respectively. When tracing is stopped on a device, the OpenMP implementation will eventually gather all trace records already collected on the device and present to the tool using the buffer completion callback provided by the tool.
- It is legal to shut down an OpenMP implementation while device tracing is in progress.
- When an OpenMP implementation begins to shut down, the OpenMP implementation will finalize each target device. Device finalization occurs in three steps. First, the OpenMP implementation halts any tracing in progress for the device. Second, the OpenMP implementation flushes all trace records collected for the device and presents them to the tool using the buffer completion callback associated with that device. Finally, the OpenMP implementation dispatches any **ompt_callback_device_finalize** callback that was previously registered by the tool.

Cross References

- **ompt_callback_device_initialize_t**, see Section 4.1.4.2.28 on page 438.
- **ompt_callback_device_finalize_t**, see Section 4.1.4.2.29 on page 440.
- **ompt_get_device_num_procs_t**, see Section 4.1.5.2.1 on page 461.
- **ompt_get_device_time**, see Section 4.1.5.2.2 on page 462.
- **ompt_translate_time**, see Section 4.1.5.2.3 on page 463.
- **ompt_set_trace_ompt**, see Section 4.1.5.2.4 on page 464.
- **ompt_set_trace_native**, see Section 4.1.5.2.5 on page 466.
- **ompt_start_trace**, see Section 4.1.5.2.6 on page 467.

- **ompt_pause_trace**, see Section 4.1.5.2.7 on page 468.
- **ompt_stop_trace**, see Section 4.1.5.2.8 on page 469.
- **ompt_advance_buffer_cursor**, see Section 4.1.5.2.9 on page 470.
- **ompt_get_record_type**, see Section 4.1.5.2.10 on page 471.
- **ompt_get_record_ompt**, see Section 4.1.5.2.11 on page 472.
- **ompt_get_record_native**, see Section 4.1.5.2.12 on page 473.
- **ompt_get_record_abstract**, see Section 4.1.5.2.13 on page 474.

4.1.2 Finalizing an OMPT Tool

If **ompt_start_tool** returned a non-**NULL** pointer when an OpenMP implementation was initialized, the tool finalizer, of type signature **ompt_finalize_t**, specified by the *finalize* field in this structure will be called as the OpenMP implementation shuts down.

Cross References

- **ompt_finalize_t**, Section 4.1.4.1.2 on page 405

4.1.3 OMPT Data Types

4.1.3.1 Tool Initialization and Finalization

Summary

A tool's implementation of **ompt_start_tool** returns a pointer to an **ompt_start_tool_result_t** structure, which contains pointers to the tool's initialization and finalization callbacks as well as an **ompt_data_t** object for use by the tool.

C / C++

```
typedef struct ompt_start_tool_result_t {
  ompt_initialize_t initialize;
  ompt_finalize_t finalize;
  ompt_data_t tool_data;
} ompt_start_tool_result_t;
```

C / C++

Restrictions

The *initialize* and *finalize* callback pointer values in an **ompt_start_tool_result_t** structure returned by **ompt_start_tool** must be non-**NULL**.

Cross References

- **ompt_data_t**, see Section 4.1.3.4.3 on page 395.
- **ompt_finalize_t**, see Section 4.1.4.1.2 on page 405.
- **ompt_initialize_t**, see Section 4.1.4.1.1 on page 404.
- **ompt_start_tool**, see Section 4.3.2.1 on page 558.

4.1.3.2 Callbacks

The following enumeration type indicates the integer codes used to identify OpenMP callbacks when registering or querying them.

C / C++

```
typedef enum ompt_callbacks_t {
  ompt_callback_thread_begin             = 1,
  ompt_callback_thread_end               = 2,
  ompt_callback_parallel_begin           = 3,
  ompt_callback_parallel_end             = 4,
  ompt_callback_task_create              = 5,
  ompt_callback_task_schedule            = 6,
  ompt_callback_implicit_task            = 7,
  ompt_callback_target                   = 8,
  ompt_callback_target_data_op           = 9,
  ompt_callback_target_submit            = 10,
  ompt_callback_control_tool             = 11,
  ompt_callback_device_initialize        = 12,
  ompt_callback_device_finalize          = 13,
  ompt_callback_device_load              = 14,
  ompt_callback_device_unload            = 15,
  ompt_callback_sync_region_wait         = 16,
  ompt_callback_mutex_released           = 17,
  ompt_callback_task_dependences         = 18,
  ompt_callback_task_dependence          = 19,
  ompt_callback_work                     = 20,
  ompt_callback_master                   = 21,
  ompt_callback_target_map               = 22,
  ompt_callback_sync_region              = 23,
  ompt_callback_lock_init                = 24,
  ompt_callback_lock_destroy             = 25,
  ompt_callback_mutex_acquire            = 26,
  ompt_callback_mutex_acquired           = 27,
  ompt_callback_nest_lock                = 28,
  ompt_callback_flush                    = 29,
  ompt_callback_cancel                   = 30,
  ompt_callback_idle                     = 31

} ompt_callbacks_t;
```

C / C++

1 ### 4.1.3.3 Tracing

2 #### 4.1.3.3.1 Record Type

C / C++

```
typedef enum ompt_record_type_t {
  ompt_record_ompt                  = 1,
  ompt_record_native                = 2,
  ompt_record_invalid               = 3
} ompt_record_type_t;
```

C / C++

3 #### 4.1.3.3.2 Native Record Kind

C / C++

```
typedef enum ompt_record_native_kind_t {
  ompt_record_native_info  = 1,
  ompt_record_native_event = 2
} ompt_record_native_kind_t;
```

C / C++

4 #### 4.1.3.3.3 Native Record Abstract Type

C / C++

```
typedef struct ompt_record_abstract_t {
  ompt_record_native_kind_t rclass;
  const char *type;
  ompt_device_time_t start_time;
  ompt_device_time_t end_time;
  ompt_hwid_t hwid;
} ompt_record_abstract_t;
```

C / C++

Description

A **`ompt_record_abstract_t`** record contains several pieces of information that a tool can use to process a native record that it may not fully understand. The *rclass* field indicates whether the record is informational or represents an event; knowing this can help a tool determine how to present the record. The record *type* field points to a statically-allocated, immutable character string that provides a meaningful name that a tool might want to use to describe the event to a user. The *start_time* and *end_time* fields are used to place an event in time. The times are relative to the device clock. If an event has no associated *start_time* and/or *end_time*, its value will be **`ompt_time_none`**. The hardware id field, *hwid*, is used to indicate the location on the device where the event occurred. A *hwid* may represent a hardware abstraction such as a core or a hardware thread id. The meaning of a *hwid* value for a device is defined by the implementer of the software stack for the device. If there is no hardware abstraction associated with the record, the value of *hwid* will be **`ompt_hwid_none`**.

1 #### 4.1.3.3.4 Record Type

C / C++

```
typedef struct ompt_record_ompt_t {
  ompt_callbacks_t type;
  ompt_device_time_t time;
  ompt_id_t thread_id;
  ompt_id_t target_id;
  union {
    ompt_record_thread_begin_t thread_begin;
    ompt_record_idle_t idle;
    ompt_record_parallel_begin_t parallel_begin;
    ompt_record_parallel_end_t parallel_end;
    ompt_record_task_create_t task_create;
    ompt_record_task_dependences_t task_deps;
    ompt_record_task_dependence_t task_dep;
    ompt_record_task_schedule_t task_sched;
    ompt_record_implicit_t implicit;
    ompt_record_sync_region_t sync_region;
    ompt_record_target_t target_record;
    ompt_record_target_data_op_t target_data_op;
    ompt_record_target_map_t target_map;
    ompt_record_target_kernel_t kernel;
    ompt_record_lock_init_t lock_init;
    ompt_record_lock_destroy_t lock_destroy;
    ompt_record_mutex_acquire_t mutex_acquire;
    ompt_record_mutex_t mutex;
    ompt_record_nest_lock_t nest_lock;
    ompt_record_master_t master;
    ompt_record_work_t work;
    ompt_record_flush_t flush;
  } record;
} ompt_record_ompt_t;
```

C / C++

2 ## Description

3 The field *type* specifies the type of record provided by this structure. According to the type, event
4 specific information is stored in the matching *record* entry.

1 **Restrictions**

2 If the *type* is set to **ompt_callback_thread_end_t**, the value of *record* is undefined.

3 4.1.3.4 Miscellaneous Type Definitions

4 This section describes miscellaneous types and enumerations used by the tool interface.

5 4.1.3.4.1 ompt_callback_t

6 Pointers to tool callback functions with many different type signatures are passed to the
7 **ompt_set_callback** runtime entry point and returned by the **ompt_get_callback**
8 runtime entry point. For convenience, these runtime entry points expect all type signatures to be
9 cast to a dummy type **ompt_callback_t**.

C / C++

```
typedef void (*ompt_callback_t)(void);
```

C / C++

10 4.1.3.4.2 ompt_id_t

11 When tracing asynchronous activity on OpenMP devices, tools need identifiers to correlate target
12 regions and operations initiated by the host with associated activities on a target device. In addition,
13 tools need identifiers to refer to parallel regions and tasks that execute on a device. OpenMP
14 implementations use identifiers of type **ompt_id_t** type for each of these purposes. The value
15 **ompt_id_none** is reserved to indicate an invalid id.

C / C++

```
typedef uint64_t ompt_id_t;
#define ompt_id_none 0
```

C / C++

16 Identifiers created on each device must be unique from the time an OpenMP implementation is
17 initialized until it is shut down. Specifically, this means that (1) identifiers for each target region
18 and target operation instance initiated by the host device must be unique over time on the host, and
19 (2) identifiers for parallel and task region instances that execute on a device must be unique over
20 time within that device.

21 Tools should not assume that **ompt_id_t** values are small or densely allocated.

4.1.3.4.3 `ompt_data_t`

Threads, parallel regions, and task regions each have an associated data object of type **ompt_data_t** reserved for use by a tool. When an OpenMP implementation creates a thread or an instance of a parallel or task region, it will initialize its associated **ompt_data_t** object with the value **ompt_data_none**.

C / C++

```
typedef union ompt_data_t {
  uint64_t value;
  void *ptr;
} ompt_data_t;

static const ompt_data_t ompt_data_none = {0};
```

C / C++

4.1.3.4.4 `ompt_device_t`

ompt_device_t is an opaque object representing a device.

C / C++

```
typedef void ompt_device_t;
```

C / C++

4.1.3.4.5 `ompt_device_time_t`

ompt_device_time_t is an opaque object representing a raw time value from a device. **ompt_time_none** refers to an uknown or unspecified time.

C / C++

```
typedef uint64_t ompt_device_time_t;
#define ompt_time_none 0
```

C / C++

4.1.3.4.6 ompt_buffer_t

ompt_buffer_t is an opaque object handle for a target buffer.

C / C++

```
typedef void ompt_buffer_t;
```

C / C++

4.1.3.4.7 ompt_buffer_cursor_t

ompt_buffer_cursor_t is an opaque handle for a position in a target buffer.

C / C++

```
typedef uint64_t ompt_buffer_cursor_t;
```

C / C++

4.1.3.4.8 ompt_task_dependence_t

ompt_task_dependence_t is a task dependence.

C / C++

```
typedef struct ompt_task_dependence_t {
  void *variable_addr;
  unsigned int dependence_flags;
} ompt_task_dependence_t;
```

C / C++

Description

ompt_task_dependence_t is a structure to hold information about a depend clause. The element *variable_addr* points to the storage location the dependency. The element *dependence_flags* indicates the kind of dependency described. If the dependency is an out dependency, **(dependence_flags & ompt_task_dependence_type_out)** evaluates to *true*. If the dependency is an in dependency, **(dependence_flags & ompt_task_dependence_type_in)** evaluates to *true*.

Cross References

- **ompt_task_dependence_flag_t**, see Section 4.1.3.4.21 on page 402.

4.1.3.4.9 ompt_thread_type_t

ompt_thread_type_t is an enumeration that defines the valid thread type values.

C / C++

```
typedef enum ompt_thread_type_t {
  ompt_thread_initial                    = 1,
  ompt_thread_worker                     = 2,
  ompt_thread_other                      = 3,
  ompt_thread_unknown                    = 4
} ompt_thread_type_t;
```

C / C++

Any *initial thread* has thread type **ompt_thread_initial**. All *OpenMP threads* that are not initial threads have thread type **ompt_thread_worker**. A thread employed by an OpenMP implementation that does not execute user code has thread type **ompt_thread_other**. Any thread created outside an OpenMP implementation that is not an *initial thread* has thread type **ompt_thread_unknown**.

4.1.3.4.10 ompt_scope_endpoint_t

ompt_scope_endpoint_t is an enumeration that defines valid scope endpoint values.

C / C++

```
typedef enum ompt_scope_endpoint_t {
  ompt_scope_begin                       = 1,
  ompt_scope_end                         = 2
} ompt_scope_endpoint_t;
```

C / C++

1
4.1.3.4.11 ompt_sync_region_kind_t

2 **ompt_sync_region_kind_t** is an enumeration that defines the valid sync region kind values.

C / C++

```
typedef enum ompt_sync_region_kind_t {
  ompt_sync_region_barrier               = 1,
  ompt_sync_region_taskwait              = 2,
  ompt_sync_region_taskgroup             = 3
} ompt_sync_region_kind_t;
```

C / C++

3
4.1.3.4.12 ompt_target_data_op_t

4 **ompt_target_data_op_t** is an enumeration that defines the valid target data operation values.

C / C++

```
typedef enum ompt_target_data_op_t {
  ompt_target_data_alloc                 = 1,
  ompt_target_data_transfer_to_dev       = 2,
  ompt_target_data_transfer_from_dev     = 3,
  ompt_target_data_delete                = 4,
  ompt_target_data_associate             = 5,
  ompt_target_data_disassociate          = 6
} ompt_target_data_op_t;
```

C / C++

5
4.1.3.4.13 ompt_work_type_t

6 **ompt_work_type_t** is an enumeration that defines the valid work type values.

C / C++

```
typedef enum ompt_work_type_t {
  ompt_work_loop               = 1,
  ompt_work_sections           = 2,
  ompt_work_single_executor    = 3,
  ompt_work_single_other       = 4,
  ompt_work_workshare          = 5,
  ompt_work_distribute         = 6,
  ompt_work_taskloop           = 7
} ompt_work_type_t;
```

C / C++

4.1.3.4.14 ompt_mutex_kind_t

ompt_mutex_kind_t is an enumeration that defines the valid mutex kind values.

C / C++

```
typedef enum ompt_mutex_kind_t {
  ompt_mutex                          = 0x10,
  ompt_mutex_lock                     = 0x11,
  ompt_mutex_nest_lock                = 0x12,
  ompt_mutex_critical                 = 0x13,
  ompt_mutex_atomic                   = 0x14,
  ompt_mutex_ordered                  = 0x20
} ompt_mutex_kind_t;
```

C / C++

4.1.3.4.15 ompt_native_mon_flags_t

ompt_native_mon_flags_t is an enumeration that defines the valid native monitoring flag values.

C / C++

```
typedef enum ompt_native_mon_flags_t {
  ompt_native_data_motion_explicit    = 1,
  ompt_native_data_motion_implicit    = 2,
  ompt_native_kernel_invocation       = 4,
  ompt_native_kernel_execution        = 8,
  ompt_native_driver                  = 16,
  ompt_native_runtime                 = 32,
  ompt_native_overhead                = 64,
  ompt_native_idleness                = 128
} ompt_native_mon_flags_t;
```

C / C++

4.1.3.4.16 ompt_task_type_t

ompt_task_type_t is an enumeration that defines the valid task type values. The least significant byte provides information about the general classification of the task. The other bits represent properties of the task.

C / C++

```
typedef enum ompt_task_type_t {
  ompt_task_initial                   = 0x1,
  ompt_task_implicit                  = 0x2,
  ompt_task_explicit                  = 0x4,
  ompt_task_target                    = 0x8,
  ompt_task_undeferred                = 0x8000000,
  ompt_task_untied                    = 0x10000000,
  ompt_task_final                     = 0x20000000,
  ompt_task_mergeable                 = 0x40000000,
  ompt_task_merged                    = 0x80000000
} ompt_task_type_t;
```

C / C++

4.1.3.4.17 ompt_task_status_t

ompt_task_status_t is an enumeration that explains the reasons for switching a task that reached a task scheduling point.

C / C++

```
typedef enum ompt_task_status_t {
  ompt_task_complete  = 1,
  ompt_task_yield     = 2,
  ompt_task_cancel    = 3,
  ompt_task_others    = 4
} ompt_task_status_t;
```

C / C++

The **ompt_task_complete** indicates the completion of task that encountered the task scheduling point. The **ompt_task_yield** indicates that the task encountered a **taskyield** construct. The **ompt_task_cancel** indicates that the taks is canceled due to the encountering of an active cancellation point resulting in the cancellation of that task. The **ompt_task_others** is used in the remaining cases.

4.1.3.4.18 ompt_target_type_t

ompt_target_type_t is an enumeration that defines the valid target type values.

C / C++

```
typedef enum ompt_target_type_t {
  ompt_target                       = 1,
  ompt_target_enter_data            = 2,
  ompt_target_exit_data             = 3,
  ompt_target_update                = 4
} ompt_target_type_t;
```

C / C++

4.1.3.4.19 ompt_invoker_t

ompt_invoker_t is an enumeration that defines the valid invoker values.

C / C++

```
typedef enum ompt_invoker_t {
  ompt_invoker_program = 1,  /* program invokes master task */
  ompt_invoker_runtime = 2   /* runtime invokes master task */
} ompt_invoker_t;
```

C / C++

1 #### 4.1.3.4.20 ompt_target_map_flag_t

2 **ompt_target_map_flag_t** is an enumeration that defines the valid target map flag values.

C / C++

```
typedef enum ompt_target_map_flag_t {
  ompt_target_map_flag_to          = 1,
  ompt_target_map_flag_from        = 2,
  ompt_target_map_flag_alloc       = 4,
  ompt_target_map_flag_release     = 8,
  ompt_target_map_flag_delete      = 16,
  ompt_target_map_flag_implicit    = 32
} ompt_target_map_flag_t;
```

C / C++

3 #### 4.1.3.4.21 ompt_task_dependence_flag_t

4 **ompt_task_dependence_flag_t** is an enumeration that defines the valid task dependence
5 flag values.

C / C++

```
typedef enum ompt_task_dependence_flag_t {
  ompt_task_dependence_type_out              = 1,
  ompt_task_dependence_type_in               = 2,
  ompt_task_dependence_type_inout            = 3,
  ompt_task_dependence_type_mutexinoutset    = 4
} ompt_task_dependence_flag_t;
```

C / C++

4.1.3.4.22 ompt_cancel_flag_t

ompt_cancel_flag_t is an enumeration that defines the valid cancel flag values.

C / C++

```
typedef enum ompt_cancel_flag_t {
  ompt_cancel_parallel       = 0x1,
  ompt_cancel_sections       = 0x2,
  ompt_cancel_do             = 0x4,
  ompt_cancel_taskgroup      = 0x8,
  ompt_cancel_activated      = 0x10,
  ompt_cancel_detected       = 0x20,
  ompt_cancel_discarded_task = 0x40
} ompt_cancel_flag_t;
```

C / C++

Cross References

- ompt_cancel_t data type, see Section 4.1.4.2.27 on page 437.

4.1.3.4.23 ompt_hwid_t

ompt_hwid_t is an opaque object representing a hardware identifier for a target device. ompt_hwid_none refers to an uknown or unspecified hardware id. If there is no hwid associated with a ompt_record_abstract_t, the value of hwid shall be ompt_hwid_none.

C / C++

```
typedef uint64_t ompt_hwid_t;
#define ompt_hwid_none 0
```

C / C++

4.1.4 OMPT Tool Callback Signatures and Trace Records

Restrictions

Tool callbacks may not use OpenMP directives or call any runtime library routines described in Section 3.

4.1.4.1 Initialization and Finalization Callback Signature

4.1.4.1.1 `ompt_initialize_t`

Summary

A tool implements an initializer with the type signature **`ompt_initialize_t`** to initialize the tool's use of the OMPT interface.

Format

C / C++

```
typedef int (*ompt_initialize_t) (
  ompt_function_lookup_t lookup,
  ompt_data_t *tool_data
);
```

C / C++

Description

For a tool to use the OMPT interface of an OpenMP implementation, the tool's implementation of **`ompt_start_tool`** must return a non-**`NULL`** pointer to an **`ompt_start_tool_result_t`** structure that contains a non-**`NULL`** pointer to a tool initializer with type signature **`ompt_initialize_t`**. An OpenMP implementation will call the tool initializer after fully initializing itself but before beginning execution of any OpenMP construct or completing execution of any environment routine invocation.

The initializer returns a non-zero value if it succeeds.

Description of Arguments

The argument *lookup* is a callback to an OpenMP runtime routine that a tool must use to obtain a pointer to each runtime entry point in the OMPT interface. The argument *tool_data* is a pointer to the *tool_data* field in the **`ompt_start_tool_result_t`** structure returned by **`ompt_start_tool`**. The expected actions of a tool initializer are described in Section 4.1.1.2 on page 379.

Cross References

- **ompt_start_tool_result_t**, see Section 4.1.3.1 on page 388.
- **ompt_data_t**, see Section 4.1.3.4.3 on page 395.
- **ompt_start_tool**, see Section 4.3.2.1 on page 558.
- **ompt_function_lookup_t**, see Section 4.1.5.3.1 on page 475.

4.1.4.1.2 ompt_finalize_t

Summary

A tool implements a finalizer with the type signature **ompt_finalize_t** to finalize the tool's use of the OMPT interface.

Format

C / C++

```
typedef void (*ompt_finalize_t) (
  ompt_data_t *tool_data
);
```

C / C++

Description

For a tool to use the OMPT interface of an OpenMP implementation, the tool's implementation of **ompt_start_tool** must return a non-**NULL** pointer to an **ompt_start_tool_result_t** structure that contains a non-**NULL** pointer to a tool finalizer with type signature **ompt_finalize_t**. An OpenMP implementation will call the tool finalizer after the last OMPT *event* as the OpenMP implementation shuts down.

Description of Arguments

The argument *tool_data* is a pointer to the *tool_data* field in the **ompt_start_tool_result_t** structure returned by **ompt_start_tool**.

Cross References

- **ompt_start_tool_result_t**, see Section 4.1.3.1 on page 388.
- **ompt_data_t**, see Section 4.1.3.4.3 on page 395.
- **ompt_start_tool**, see Section 4.3.2.1 on page 558.

4.1.4.2 Event Callback Signatures and Trace Records

This section describes the signatures of tool callback functions that an OMPT tool might register and that are called during runtime of an OpenMP program.

4.1.4.2.1 ompt_callback_thread_begin_t

Format

C / C++

```
typedef void (*ompt_callback_thread_begin_t) (
  ompt_thread_type_t thread_type,
  ompt_data_t *thread_data
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_thread_begin_t {
  ompt_thread_type_t thread_type;
} ompt_record_thread_begin_t;
```

C / C++

Description of Arguments

The argument *thread_type* indicates the type of the new thread: initial, worker, or other.

The binding of argument *thread_data* is the new thread.

1 ### Cross References

2 • **ompt_data_t** type, see Section 4.1.3.4.3 on page 395.

3 • **ompt_thread_type_t** type, see Section 4.1.3.4.9 on page 397.

4 ## 4.1.4.2.2 ompt_callback_thread_end_t

5 ### Format

C / C++

```
typedef void (*ompt_callback_thread_end_t) (
  ompt_data_t *thread_data
);
```

C / C++

6 ### Description of Arguments

7 The binding of argument *thread_data* is the thread that is terminating.

8 ### Cross References

9 • **ompt_data_t** type, see Section 4.1.3.4.3 on page 395.

10 • **ompt_record_ompt_t** type, see Section 4.1.3.3.4 on page 393.

11 ## 4.1.4.2.3 ompt_callback_idle_t

12 ### Format

C / C++

```
typedef void (*ompt_callback_idle_t) (
  ompt_scope_endpoint_t endpoint
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_idle_t {
  ompt_scope_endpoint_t endpoint;
} ompt_record_idle_t;
```

C / C++

Description of Arguments

The argument *endpoint* indicates whether the callback is signalling the beginning or end of an idle interval.

Cross References

- **ompt_scope_endpoint_t** type, see Section 4.1.3.4.10 on page 397.

4.1.4.2.4 ompt_callback_parallel_begin_t

Format

C / C++

```
typedef void (*ompt_callback_parallel_begin_t) (
  ompt_data_t *encountering_task_data,
  const ompt_frame_t *encountering_task_frame,
  ompt_data_t *parallel_data,
  unsigned int requested_team_size,
  ompt_invoker_t invoker,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_parallel_begin_t {
  ompt_id_t encountering_task_id;
  ompt_id_t parallel_id;
  unsigned int requested_team_size;
  ompt_invoker_t invoker;
  const void *codeptr_ra;
} ompt_record_parallel_begin_t;
```

C / C++

Description of Arguments

The binding of argument *encountering_task_data* is the encountering task.

The argument *encountering_task_frame* points to the frame object associated with the encountering task.

The binding of argument *parallel_data* is the parallel region that is beginning.

The argument *requested_team_size* indicates the number of threads requested by the user.

The argument *invoker* indicates whether the code for the parallel region is inlined into the application or invoked by the runtime.

The argument *codeptr_ra* is used to relate the implementation of an OpenMP region back to its source code. In cases where a runtime routine implements the region associated with this callback, *codeptr_ra* is expected to contain the return address of the call to the runtime routine. In cases where the implementation of this feature is inlined, *codeptr_ra* is expected to contain the return address of the invocation of this callback. In cases where attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- **ompt_data_t** type, see Section 4.1.3.4.3 on page 395.
- **omp_frame_t** type, see Section 4.3.1.2 on page 556.
- **ompt_invoker_t** type, see Section 4.1.3.4.19 on page 401.

4.1.4.2.5 ompt_callback_parallel_end_t

Format

C / C++

```
typedef void (*ompt_callback_parallel_end_t) (
  ompt_data_t *parallel_data,
  ompt_data_t *encountering_task_data,
  ompt_invoker_t invoker,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_parallel_end_t {
  ompt_id_t parallel_id;
  ompt_id_t encountering_task_id;
  ompt_invoker_t invoker;
  const void *codeptr_ra;
} ompt_record_parallel_end_t;
```

C / C++

Description of Arguments

The binding of argument *parallel_data* is the parallel region that is ending.

The binding of argument *encountering_task_data* is the encountering task.

The argument *invoker* explains whether the execution of the parallel region code is inlined into the application code or started by the runtime.

The argument *codeptr_ra* is used to relate the implementation of an OpenMP region back to its source code. In cases where a runtime routine implements the region associated with this callback, *codeptr_ra* is expected to contain the return address of the call to the runtime routine. In cases where the implementation of this feature is inlined, *codeptr_ra* is expected to contain the return address of the invocation of this callback. In cases where attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- **ompt_data_t** type signature, see Section 4.1.3.4.3 on page 395.
- **ompt_invoker_t** type signature, see Section 4.1.3.4.19 on page 401.

4.1.4.2.6 ompt_callback_master_t

Format

C / C++

```
typedef void (*ompt_callback_master_t) (
  ompt_scope_endpoint_t endpoint,
  ompt_data_t *parallel_data,
  ompt_data_t *task_data,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_master_t {
  ompt_scope_endpoint_t endpoint;
  ompt_id_t parallel_id;
  ompt_id_t task_id;
  const void *codeptr_ra;
} ompt_record_master_t;
```

C / C++

Description of Arguments

The argument *endpoint* indicates whether the callback is signalling the beginning or the end of a scope.

The binding of argument *parallel_data* is the current parallel region.

The binding of argument *task_data* is the encountering task.

The argument *codeptr_ra* is used to relate the implementation of an OpenMP region back to its source code. In cases where a runtime routine implements the region associated with this callback,

codeptr_ra is expected to contain the return address of the call to the runtime routine. In cases where the implementation of this feature is inlined, *codeptr_ra* is expected to contain the return address of the invocation of this callback. In cases where attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- **ompt_data_t** type signature, see Section 4.1.3.4.3 on page 395.
- **ompt_scope_endpoint_t** type, see Section 4.1.3.4.10 on page 397.

4.1.4.2.7 ompt_callback_task_create_t

Format

C / C++

```
typedef void (*ompt_callback_task_create_t) (
  ompt_data_t *encountering_task_data,
  const ompt_frame_t *encountering_task_frame,
  ompt_data_t *new_task_data,
  int type,
  int has_dependences,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_task_create_t {
  ompt_id_t encountering_task_id;
  ompt_id_t new_task_id;
  int type;
  int has_dependences;
  const void *codeptr_ra;
} ompt_record_task_create_t;
```

C / C++

Description of Arguments

The binding of argument *encountering_task_data* is the encountering task. This parameter is **NULL** for an initial task.

The argument *encountering_task_frame* points to the frame object associated with the encountering task. This parameter is **NULL** for an initial task.

The binding of argument *new_task_data* is the created task.

The argument *type* indicates the kind of the task: initial, explicit or target. Values for *type* are composed by or-ing elements of enum **ompt_task_type_t**.

The argument *has_dependences* indicates whether created task has dependences.

The argument *codeptr_ra* is used to relate the implementation of an OpenMP region back to its source code. In cases where a runtime routine implements the region associated with this callback, *codeptr_ra* is expected to contain the return address of the call to the runtime routine. In cases where the implementation of this feature is inlined, *codeptr_ra* is expected to contain the return address of the invocation of this callback. In cases where attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- **ompt_data_t** type, see Section 4.1.3.4.3 on page 395.
- **omp_frame_t** type, see Section 4.3.1.2 on page 556.
- **ompt_task_type_t** type, see Section 4.1.3.4.16 on page 400.

4.1.4.2.8 ompt_callback_task_dependences_t

Format

C / C++

```
typedef void (*ompt_callback_task_dependences_t) (
  ompt_data_t *task_data,
  const ompt_task_dependence_t *deps,
  int ndeps
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_task_dependences_t {
  ompt_id_t task_id;
  ompt_task_dependence_t dep;
  int ndeps;
} ompt_record_task_dependences_t;
```

C / C++

Description of Arguments

The binding of argument *task_data* is the task being created.

The argument *deps* lists all dependences of a new task.

The argument *ndeps* specifies the length of the list. The memory for *deps* is owned by the caller; the tool cannot rely on the data after the callback returns.

The performance monitor interface for tracing activity on target devices will provide one record per dependence.

Cross References

- **ompt_data_t** type, see Section 4.1.3.4.3 on page 395.
- **ompt_task_dependence_t** type, see Section 4.1.3.4.8 on page 396.

4.1.4.2.9 ompt_callback_task_dependence_t

Format

C / C++

```
typedef void (*ompt_callback_task_dependence_t) (
  ompt_data_t *src_task_data,
  ompt_data_t *sink_task_data
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_task_dependence_t {
  ompt_id_t src_task_id;
  ompt_id_t sink_task_id;
} ompt_record_task_dependence_t;
```

C / C++

Description of Arguments

The binding of argument *src_task_data* is a running task with an outgoing dependence.

The binding of argument *sink_task_data* is a task with an unsatisfied incoming dependence.

Cross References

- **ompt_data_t** type signature, see Section 4.1.3.4.3 on page 395.

4.1.4.2.10 ompt_callback_task_schedule_t

Format

C / C++

```
typedef void (*ompt_callback_task_schedule_t) (
  ompt_data_t *prior_task_data,
  ompt_task_status_t prior_task_status,
  ompt_data_t *next_task_data
);
```

C / C++

1 **Trace Record**

C / C++

```
typedef struct ompt_record_task_schedule_t {
  ompt_id_t prior_task_id;
  ompt_task_status_t prior_task_status,
  ompt_id_t next_task_id;
} ompt_record_task_schedule_t;
```

C / C++

2 **Description of Arguments**

3 The argument *prior_task_status* indicates the status of the task that arrived at a task scheduling
4 point.

5 The binding of argument *prior_task_data* is the task that arrived at the scheduling point.

6 The binding of argument *next_task_data* is the task that will resume at the scheduling point.

7 **Cross References**

8 • **ompt_data_t** type, see Section 4.1.3.4.3 on page 395.

9 • **ompt_task_status_t** type, see Section 4.1.3.4.17 on page 400.

10 #### 4.1.4.2.11 ompt_callback_implicit_task_t

11 **Format**

C / C++

```
typedef void (*ompt_callback_implicit_task_t) (
  ompt_scope_endpoint_t endpoint,
  ompt_data_t *parallel_data,
  ompt_data_t *task_data,
  unsigned int team_size,
  unsigned int thread_num
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_implicit_t {
  ompt_scope_endpoint_t endpoint;
  ompt_id_t parallel_id;
  ompt_id_t task_id;
  unsigned int team_size,
  unsigned int thread_num;
} ompt_record_implicit_t;
```

C / C++

Description of Arguments

The argument *endpoint* indicates whether the callback is signalling the beginning or the end of a scope.

The binding of argument *parallel_data* is the current parallel region. For the *implicit-task-end* event, this argument is **NULL**.

The binding of argument *task_data* is the implicit task executing the parallel region's structured block.

The argument *team_size* indicates the number of threads in the team.

The argument *thread_num* indicates the thread number of the calling thread, within the team executing the parallel region to which the implicit region binds.

Cross References

- **ompt_data_t** type, see Section 4.1.3.4.3 on page 395.
- **ompt_scope_endpoint_t** enumeration type, see Section 4.1.3.4.10 on page 397.

1 #### 4.1.4.2.12 `ompt_callback_sync_region_t`

2 **Format**

C / C++

```
typedef void (*ompt_callback_sync_region_t) (
  ompt_sync_region_kind_t kind,
  ompt_scope_endpoint_t endpoint,
  ompt_data_t *parallel_data,
  ompt_data_t *task_data,
  const void *codeptr_ra
);
```

C / C++

3 **Trace Record**

C / C++

```
typedef struct ompt_record_sync_region_t {
  ompt_sync_region_kind_t kind;
  ompt_scope_endpoint_t endpoint;
  ompt_id_t parallel_id;
  ompt_id_t task_id;
  const void *codeptr_ra;
} ompt_record_sync_region_t;
```

C / C++

4 **Description of Arguments**

5 The argument *kind* indicates the kind of synchronization region.

6 The argument *endpoint* indicates whether the callback is signalling the beginning or the end of a
7 scope.

8 The binding of argument *parallel_data* is the current parallel region. For the *barrier-end* event at
9 the end of a parallel region, this argument is **NULL**.

10 The binding of argument *task_data* is the current task.

11 The argument *codeptr_ra* is used to relate the implementation of an OpenMP region back to its
12 source code. In cases where a runtime routine implements the region associated with this callback,
13 *codeptr_ra* is expected to contain the return address of the call to the runtime routine. In cases
14 where the implementation of this feature is inlined, *codeptr_ra* is expected to contain the return

1 address of the invocation of this callback. In cases where attribution to source code is impossible or
2 inappropriate, *codeptr_ra* may be **NULL**.

3 **Cross References**

4 • **ompt_data_t** type, see Section 4.1.3.4.3 on page 395.
5 • **ompt_sync_region_kind_t** type, see Section 4.1.3.4.11 on page 398.
6 • **ompt_scope_endpoint_t** type, see Section 4.1.3.4.10 on page 397.

7 #### 4.1.4.2.13 ompt_callback_mutex_acquire_t

8 **Format**

C / C++

```
typedef void (*ompt_callback_mutex_acquire_t) (
  ompt_mutex_kind_t kind,
  unsigned int hint,
  unsigned int impl,
  omp_wait_id_t wait_id,
  const void *codeptr_ra
);
```

C / C++

9 **Trace Record**

C / C++

```
typedef struct ompt_record_mutex_acquire_t {
  ompt_mutex_kind_t kind;
  unsigned int hint;
  unsigned int impl;
  omp_wait_id_t wait_id;
  const void *codeptr_ra;
} ompt_record_mutex_acquire_t;
```

C / C++

Description of Arguments

The argument *kind* indicates the kind of the lock.

The argument *hint* indicates the hint provided when initializing an implementation of mutual exclusion. If no hint is available when a thread initiates acquisition of mutual exclusion, the runtime may supply **omp_lock_hint_none** as the value for *hint*.

The argument *impl* indicates the mechanism chosen by the runtime to implement the mutual exclusion.

The argument *wait_id* indicates the object being awaited.

The argument *codeptr_ra* is used to relate the implementation of an OpenMP region back to its source code. In cases where a runtime routine implements the region associated with this callback, *codeptr_ra* is expected to contain the return address of the call to the runtime routine. In cases where the implementation of this feature is inlined, *codeptr_ra* is expected to contain the return address of the invocation of this callback. In cases where attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- **omp_wait_id_t** type, see Section 4.3.1.3 on page 558.
- **ompt_mutex_kind_t** type, see Section 4.1.3.4.14 on page 399.

4.1.4.2.14 ompt_callback_mutex_t

Format

C / C++

```
typedef void (*ompt_callback_mutex_t) (
  ompt_mutex_kind_t kind,
  omp_wait_id_t wait_id,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_mutex_t {
  ompt_mutex_kind_t kind;
  omp_wait_id_t wait_id;
  const void *codeptr_ra;
} ompt_record_mutex_t;
```

C / C++

Description of Arguments

The argument *kind* indicates the kind of mutual exclusion event.

The argument *wait_id* indicates the object being awaited.

The argument *codeptr_ra* is used to relate the implementation of an OpenMP region back to its source code. In cases where a runtime routine implements the region associated with this callback, *codeptr_ra* is expected to contain the return address of the call to the runtime routine. In cases where the implementation of this feature is inlined, *codeptr_ra* is expected to contain the return address of the invocation of this callback. In cases where attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- **omp_wait_id_t** type signature, see Section 4.3.1.3 on page 558.
- **ompt_mutex_kind_t** type signature, see Section 4.1.3.4.14 on page 399.

4.1.4.2.15 ompt_callback_nest_lock_t

Format

C / C++

```
typedef void (*ompt_callback_nest_lock_t) (
  ompt_scope_endpoint_t endpoint,
  omp_wait_id_t wait_id,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_nest_lock_t {
  ompt_scope_endpoint_t endpoint;
  omp_wait_id_t wait_id;
  const void *codeptr_ra;
} ompt_record_nest_lock_t;
```

C / C++

Description of Arguments

The argument *endpoint* indicates whether the callback is signalling the beginning or the end of a scope.

The argument *wait_id* indicates the object being awaited.

The argument *codeptr_ra* is used to relate the implementation of an OpenMP region back to its source code. In cases where a runtime routine implements the region associated with this callback, *codeptr_ra* is expected to contain the return address of the call to the runtime routine. In cases where the implementation of this feature is inlined, *codeptr_ra* is expected to contain the return address of the invocation of this callback. In cases where attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- **omp_wait_id_t** type signature, see Section 4.3.1.3 on page 558.
- **ompt_scope_endpoint_t** type signature, see Section 4.1.3.4.10 on page 397.

4.1.4.2.16 `ompt_callback_work_t`

Format

C / C++

```
typedef void (*ompt_callback_work_t) (
  ompt_work_type_t wstype,
  ompt_scope_endpoint_t endpoint,
  ompt_data_t *parallel_data,
  ompt_data_t *task_data,
  uint64_t count,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_work_t {
  ompt_work_type_t wstype;
  ompt_scope_endpoint_t endpoint;
  ompt_id_t parallel_id;
  ompt_id_t task_id;
  uint64_t count;
  const void *codeptr_ra;
} ompt_record_work_t;
```

C / C++

Description of Arguments

The argument *wstype* indicates the kind of worksharing region.

The argument *endpoint* indicates whether the callback is signalling the beginning or the end of a scope.

The binding of argument *parallel_data* is the current parallel region.

The binding of argument *task_data* is the current task.

The argument *count* is a measure of the quantity of work involved in the worksharing construct. For a loop construct, *count* represents the number of iterations of the loop. For a **taskloop** construct, *count* represents the number of iterations in the iteration space, which may be the result of

collapsing several associated loops. For a **sections** construct, *count* represents the number of sections. For a **workshare** construct, *count* represents the units of work, as defined by the **workshare** construct. For a **single** construct, *count* is always 1. When the *endpoint* argument is signaling the end of a scope, a *count* value of 0 indicates that the actual *count* value is not available.

The argument *codeptr_ra* is used to relate the implementation of an OpenMP region back to its source code. In cases where a runtime routine implements the region associated with this callback, *codeptr_ra* is expected to contain the return address of the call to the runtime routine. In cases where the implementation of this feature is inlined, *codeptr_ra* is expected to contain the return address of the invocation of this callback. In cases where attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- worksharing constructs, see Section 2.10 on page 77.
- **ompt_data_t** type signature, see Section 4.1.3.4.3 on page 395.
- **ompt_scope_endpoint_t** type signature, see Section 4.1.3.4.10 on page 397.
- **ompt_work_type_t** type signature, see Section 4.1.3.4.13 on page 398.

4.1.4.2.17 ompt_callback_flush_t

Format

C / C++

```
typedef void (*ompt_callback_flush_t) (
  ompt_data_t *thread_data,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_flush_t {
  const void *codeptr_ra;
} ompt_record_flush_t;
```

C / C++

Description of Arguments

The argument *codeptr_ra* is used to relate the implementation of an OpenMP region back to its source code. In cases where a runtime routine implements the region associated with this callback, *codeptr_ra* is expected to contain the return address of the call to the runtime routine. In cases where the implementation of this feature is inlined, *codeptr_ra* is expected to contain the return address of the invocation of this callback. In cases where attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- **ompt_data_t** type signature, see Section 4.1.3.4.3 on page 395.

4.1.4.2.18 ompt_callback_target_t

Format

C / C++

```
typedef void (*ompt_callback_target_t) (
  ompt_target_type_t kind,
  ompt_scope_endpoint_t endpoint,
  uint64_t device_num,
  ompt_data_t *task_data,
  ompt_id_t target_id,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_target_t {
  ompt_target_type_t kind;
  ompt_scope_endpoint_t endpoint;
  uint64_t device_num;
  ompt_data_t *task_data;
  ompt_id_t target_id;
  const void *codeptr_ra;
} ompt_record_target_t;
```

C / C++

Description of Arguments

The argument *kind* indicates the kind of target region.

The argument *endpoint* indicates whether the callback is signalling the beginning or the end of a scope.

The argument *device_num* indicates the id of the device which will execute the target region.

The binding of argument *task_data* is the target task.

The binding of argument *target_id* is the target region.

The argument *codeptr_ra* is used to relate the implementation of an OpenMP region back to its source code. In cases where a runtime routine implements the region associated with this callback, *codeptr_ra* is expected to contain the return address of the call to the runtime routine. In cases where the implementation of this feature is inlined, *codeptr_ra* is expected to contain the return address of the invocation of this callback. In cases where attribution to source code is impossible or inappropriate, *codeptr_ra* may be **`NULL`**.

Cross References

- **`ompt_id_t`** type, see Section 4.1.3.4.2 on page 394.
- **`ompt_data_t`** type signature, see Section 4.1.3.4.3 on page 395.
- **`ompt_scope_endpoint_t`** type signature, see Section 4.1.3.4.10 on page 397.
- **`ompt_target_type_t`** type signature, see Section 4.1.3.4.18 on page 401.

4.1.4.2.19 `ompt_callback_device_load_t`

Summary

The OpenMP runtime invokes this callback to notify a tool immediately after loading code onto the specified device.

1
Format

C / C++

```
typedef void (*ompt_callback_device_load_t) (
  uint64_t device_num,
  const char *filename,
  int64_t offset_in_file,
  void *vma_in_file,
  size_t bytes
  void *host_addr,
  void *device_addr,
  uint64_t module_id
);

#define ompt_addr_unknown  ((void *) ~0)
```

C / C++

2
Description of Arguments

3 The argument *device_num* specifies the device.

4 The argument *filename* indicates the name of a file in which the device code can be found. A NULL
5 *filename* indicates that the code is not available in a file in the file system.

6 The argument *offset_in_file* indicates an offset into *filename* at which the code can be found. A
7 value of -1 indicates that no offset is provided.

8 The argument *vma_in_file* indicates an virtual address in *filename* at which the code can be found.
9 A value of *ompt_addr_unknown* indicates that a virtual address in the file is not available.

10 The argument *bytes* indicates the size of the device code object in bytes.

11 The argument *host_addr* indicates where a copy of the device code is available in host memory. A
12 value of *ompt_addr_unknown* indicates that a host code address is not available.

13 The argument *device_addr* indicates where the device code has been loaded in device memory. A
14 value of *ompt_addr_unknown* indicates that a device code address is not available.

15 The argument *module_id* is an identifier that is associated with the device code object.

1 #### 4.1.4.2.20 `ompt_callback_device_unload_t`

2 **Summary**

3 The OpenMP runtime invokes this callback to notify a tool immediately prior to unloading code
4 from the specified device.

5 **Format**

C / C++

```
typedef void (*ompt_callback_device_unload_t) (
  uint64_t device_num,
  uint64_t module_id
);
```

C / C++

6 **Description of Arguments**

7 The argument *device_num* specifies the device.

8 The argument *module_id* is an identifier that is associated with the device code object.

9 #### 4.1.4.2.21 `ompt_callback_target_data_op_t`

10 **Format**

C / C++

```
typedef void (*ompt_callback_target_data_op_t) (
  ompt_id_t target_id,
  ompt_id_t host_op_id,
  ompt_target_data_op_t optype,
  void *src_addr,
  int src_device_num,
  void *dest_addr,
  int dest_device_num,
  size_t bytes,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_target_data_op_t {
  ompt_id_t host_op_id;
  ompt_target_data_op_t optype;
  void *src_addr;
  int src_device_num;
  void *dest_addr;
  int dest_device_num;
  size_t bytes;
  ompt_device_time_t end_time;
  const void *codeptr_ra;
} ompt_record_target_data_op_t;
```

C / C++

Description

An OpenMP implementation will dispatch a registered **ompt_callback_target_data_op** callback when device memory is allocated or freed, as well as when data is copied to or from a device.

Note –

An OpenMP implementation may aggregate program variables and data operations upon them. For instance, an OpenMP implementation may synthesize a composite to represent multiple scalars and then allocate, free, or copy this composite as a whole rather than performing data operations on each scalar individually. For that reason, a tool should not expect to see separate data operations on each variable.

Description of Arguments

The argument *host_op_id* is a unique identifer for a data operations on a target device.

The argument *optype* indicates the kind of data mapping.

The argument *src_addr* indicates the address of data before the operation, where applicable.

The argument *src_device_num* indicates the source device number for the data operation, where applicable.

The argument *dest_addr* indicates the address of data after the operation.

The argument *dest_device_num* indicates the destination device number for the data operation.

It is implementation defined whether in some operations *src_addr* or *dest_addr* might point to an intermediate buffer.

The argument *bytes* indicates the size of data.

The argument *codeptr_ra* is used to relate the implementation of an OpenMP region back to its source code. In cases where a runtime routine implements the region associated with this callback, *codeptr_ra* is expected to contain the return address of the call to the runtime routine. In cases where the implementation of this feature is inlined, *codeptr_ra* is expected to contain the return address of the invocation of this callback. In cases where attribution to source code is impossible or inappropriate, *codeptr_ra* may be **`NULL`**.

Cross References

- **`ompt_id_t`** type, see Section 4.1.3.4.2 on page 394.
- **`ompt_target_data_op_t`** type signature, see Section 4.1.3.4.12 on page 398.

4.1.4.2.22 `ompt_callback_target_map_t`

Format

C / C++

```
typedef void (*ompt_callback_target_map_t) (
  ompt_id_t target_id,
  unsigned int nitems,
  void **host_addr,
  void **device_addr,
  size_t *bytes,
  unsigned int *mapping_flags,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_target_map_t {
  ompt_id_t target_id;
  unsigned int nitems;
  void **host_addr;
  void **device_addr;
  size_t *bytes;
  unsigned int *mapping_flags;
  const void *codeptr_ra;
} ompt_record_target_map_t;
```

C / C++

Description

An instance of a **target**, **target data**, or **target enter data** construct may contain one or more **map** clauses. An OpenMP implementation may report the set of mappings associated with **map** clauses for a construct with a single **ompt_callback_target_map** callback to report the effect of all mappings or multiple **ompt_callback_target_map** callbacks with each reporting a subset of the mappings. Furthermore, an OpenMP implementation may omit mappings that it determines are unnecessary. If an OpenMP implementation issues multiple **ompt_callback_target_map** callbacks, these callbacks may be interleaved with

ompt_callback_target_data_op callbacks used to report data operations associated with the mappings.

Description of Arguments

The binding of argument *target_id* is the target region.

The argument *nitems* indicates the number of data mappings being reported by this callback.

The argument *host_addr* indicates an array of addresses of data on host side.

The argument *device_addr* indicates an array of addresses of data on device side.

The argument *bytes* indicates an array of size of data.

The argument *mapping_flags* indicates the kind of data mapping. Flags for a mapping include one or more values specified by the type **ompt_target_map_flag_t**.

The argument *codeptr_ra* is used to relate the implementation of an OpenMP region back to its source code. In cases where a runtime routine implements the region associated with this callback, *codeptr_ra* is expected to contain the return address of the call to the runtime routine. In cases where the implementation of this feature is inlined, *codeptr_ra* is expected to contain the return address of the invocation of this callback. In cases where attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- **ompt_id_t** type, see Section 4.1.3.4.2 on page 394.
- **ompt_callback_target_data_op_t**, see Section 4.1.4.2.21 on page 428.
- **ompt_target_map_flag_t** type, see Section 4.1.3.4.20 on page 402.

4.1.4.2.23 ompt_callback_target_submit_t

Format

C / C++

```
typedef void (*ompt_callback_target_submit_t) (
  ompt_id_t target_id,
  ompt_id_t host_op_id,
  unsigned int requested_num_teams
);
```

C / C++

Description

A thread dispatches a registered **ompt_callback_target_submit** callback on the host when a target task creates an initial task on a target device.

Description of Arguments

The argument *target_id* is a unique identifier for the associated target region.

The argument *host_op_id* is a unique identifer for the initial task on the target device.

The argument *requested_num_teams* is the number of teams that the host is requesting to execute the kernel. The actual number of teams that execute the kernel may be smaller and generally won't be known until the kernel begins to execute on the device.

Constraints on Arguments

The argument *target_id* indicates the instance of the target construct to which the computation belongs.

The argument *host_op_id* provides a unique host-side identifier that represents the computation on the device.

Trace Record

C / C++

```
typedef struct ompt_record_target_kernel_t {
  ompt_id_t host_op_id;
  unsigned int requested_num_teams;
  unsigned int granted_num_teams;
  ompt_device_time_t end_time;
} ompt_record_target_kernel_t;
```

C / C++

If a tool has configured a device to trace kernel execution using **ompt_set_trace_ompt**, the device will log a **ompt_record_target_kernel_t** record in a trace. The fields in the record are as follows:

- The *host_op_id* field contains a unique identifier that a tool can use to correlate a **ompt_record_target_kernel_t** record with its associated **ompt_callback_target_submit** callback on the host.
- The *requested_num_teams* field contains the number of teams that the host requested to execute the kernel.

- The *granted_num_teams* field contains the number of teams that the device actually used to execute the kernel.
- The time when the initial task began execution on the device is recorded in the *time* field of an enclosing **ompt_record_t** structure; the time when the initial task completed execution on the device is recorded in the *end_time* field.

Cross References

- **ompt_id_t** type, see Section 4.1.3.4.2 on page 394.

4.1.4.2.24 ompt_callback_buffer_request_t

Summary

The OpenMP runtime will invoke a callback with type signature **ompt_callback_buffer_request_t** to request a buffer to store event records for a device.

Format

C / C++

```
typedef void (*ompt_callback_buffer_request_t) (
  uint64_t device_num,
  ompt_buffer_t **buffer,
  size_t *bytes
);
```

C / C++

Description

This callback requests a buffer to store trace records for the specified device.

A buffer request callback may set **bytes* to 0 if it does not want to provide a buffer for any reason. If a callback sets **bytes* to 0, further recording of events for the device will be disabled until the next invocation of **ompt_start_trace**. This will cause the device to drop future trace records until recording is restarted.

The buffer request callback is not required to be *async signal safe*.

Description of Arguments

The argument *device_num* specifies the device.

A tool should set **buffer* to point to a buffer where device events may be recorded and **bytes* to the length of that buffer.

Cross References

- **ompt_buffer_t** type, see Section 4.1.3.4.6 on page 396.

4.1.4.2.25 ompt_callback_buffer_complete_t

Summary

A device triggers a call to **ompt_callback_buffer_complete_t** when no further records will be recorded in an event buffer and all records written to the buffer are valid.

Format

C / C++

```
typedef void (*ompt_callback_buffer_complete_t) (
  uint64_t device_num,
  ompt_buffer_t *buffer,
  size_t bytes,
  ompt_buffer_cursor_t begin,
  int buffer_owned
);
```

C / C++

Description

This callback provides a tool with a buffer containing trace records for the specified device. Typically, a tool will iterate through the records in the buffer and process them.

The OpenMP implementation will make these callbacks on a thread that is not an OpenMP master or worker.

The callee may delete the buffer if the argument *buffer_owned*=0.

The buffer completion callback is not required to be *async signal safe*.

Description of Arguments

The argument *device_num* indicates the device whose events the buffer contains.

The argument *buffer* is the address of a buffer previously allocated by a *buffer request* callback.

The argument *bytes* indicates the full size of the buffer.

The argument *begin* is an opaque cursor that indicates the position at the beginning of the first record in the buffer.

The argument *buffer_owned* is 1 if the data pointed to by buffer can be deleted by the callback and 0 otherwise. If multiple devices accumulate trace events into a single buffer, this callback might be invoked with a pointer to one or more trace records in a shared buffer with *buffer_owned* = 0. In this case, the callback may not delete the buffer.

Cross References

- **ompt_buffer_t** type, see Section 4.1.3.4.6 on page 396.
- **ompt_buffer_cursor_t** type, see Section 4.1.3.4.7 on page 396.

4.1.4.2.26 ompt_callback_control_tool_t

Format

C / C++

```
typedef int (*ompt_callback_control_tool_t) (
  uint64_t command,
  uint64_t modifier,
  void *arg,
  const void *codeptr_ra
);
```

C / C++

Description

The tool control callback may return any non-negative value, which will be returned to the application by the OpenMP implementation as the return value of the **omp_control_tool** call that triggered the callback.

Description of Arguments

The argument *command* passes a command from an application to a tool. Standard values for *command* are defined by **omp_control_tool_t**. defined in Section 3.7 on page 372.

The argument *modifier* passes a command modifier from an application to a tool.

This callback allows tool-specific values for *command* and *modifier*. Tools must ignore *command* values that they are not explicitly designed to handle.

The argument *arg* is a void pointer that enables a tool and an application to pass arbitrary state back and forth. The argument *arg* may be **NULL**.

The argument *codeptr_ra* is used to relate the implementation of an OpenMP region back to its source code. In cases where a runtime routine implements the region associated with this callback, *codeptr_ra* is expected to contain the return address of the call to the runtime routine. In cases where the implementation of this feature is inlined, *codeptr_ra* is expected to contain the return address of the invocation of this callback. In cases where attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Constraints on Arguments

Tool-specific values for *command* must be ≥ 64.

Cross References

- **omp_control_tool_t** enumeration type, see Section 3.7 on page 372.

4.1.4.2.27 ompt_callback_cancel_t

Format

C / C++

```
typedef void (*ompt_callback_cancel_t) (
  ompt_data_t *task_data,
  int flags,
  const void *codeptr_ra
  );
```

C / C++

Description of Arguments

The argument *task_data* corresponds to the task encountering a **cancel** construct, a **cancellation point** construct, or a construct defined as having an implicit cancellation point.

The argument *flags*, defined by the enumeration **ompt_cancel_flag_t**, indicates whether the cancel is activated by the current task, or detected as being activated by another task. The construct being canceled is also described in the *flags*. When several constructs are detected as being concurrently canceled, each corresponding bit in the flags will be set.

The argument *codeptr_ra* is used to relate the implementation of an OpenMP region back to its source code. In cases where a runtime routine implements the region associated with this callback, *codeptr_ra* is expected to contain the return address of the call to the runtime routine. In cases where the implementation of this feature is inlined, *codeptr_ra* is expected to contain the return address of the invocation of this callback. In cases where attribution to source code is impossible or inappropriate, *codeptr_ra* may be **NULL**.

Cross References

- **omp_cancel_flag_t** enumeration type, see Section 4.1.3.4.22 on page 403.

4.1.4.2.28 ompt_callback_device_initialize_t

Summary

The tool callback with type signature **ompt_callback_device_initialize_t** initializes a tool's tracing interface for a device.

Format

C / C++

```
typedef void (*ompt_callback_device_initialize_t) (
  uint64_t device_num,
  const char *type,
  ompt_device_t *device,
  ompt_function_lookup_t lookup,
  const char *documentation
);
```

C / C++

Description

A tool that wants to asynchronously collect a trace of activities on a device should register a callback with type signature **`ompt_callback_device_initialize_t`** for the **`ompt_callback_device_initialize`** OpenMP event. An OpenMP implementation will invoke this callback for a device after OpenMP is initialized for the device but before beginning execution of any OpenMP construct on the device.

Description of Arguments

The argument *device_num* identifies the logical device being initialized.

The argument *type* is a character string indicating the type of the device. A device type string is a semicolon separated character string that includes at a minimum the vendor and model name of the device. This may be followed by a semicolon-separated sequence of properties that describe a device's hardware or software.

The argument *device* is a pointer to an opaque object that represents the target device instance. The pointer to the device instance object is used by functions in the device tracing interface to identify the device being addressed.

The argument *lookup* is a pointer to a runtime callback that a tool must use to obtain pointers to runtime entry points in the device's OMPT tracing interface. If a device does not support tracing, it should provide **`NULL`** for *lookup*.

The argument *documentation* is a string that describes how to use any device-specific runtime entry points that can be obtained using *lookup*. This documentation string could simply be a pointer to external documentation, or it could be inline descriptions that includes names and type signatures for any device-specific interfaces that are available through *lookup* along with descriptions of how to use these interface functions to control monitoring and analysis of device traces.

Constraints on Arguments

The arguments *type* and *documentation* must be immutable strings that are defined for the lifetime of a program execution.

Effect

A tool's device initializer has several duties. First, it should use *type* to determine whether the tool has any special knowledge about a device's hardware and/or software. Second, it should use *lookup* to look up pointers to runtime entry points in the OMPT tracing interface for the device. Finally, using these runtime entry points, it can then set up tracing for a device.

Initializing tracing for a target device is described in section Section 4.1.1.4 on page 383.

Cross References

- **ompt_function_lookup_t**, see Section 4.1.5.3.1 on page 475.

4.1.4.2.29 ompt_callback_device_finalize_t

Summary

The tool callback with type signature **ompt_callback_device_finalize_t** finalizes a tool's tracing interface for a device.

Format

C / C++

```
typedef void (*ompt_callback_device_finalize_t) (
  uint64_t device_num
);
```

C / C++

Description of Arguments

The argument *device_num* identifies the logical device being finalized.

Description

A device finalization callback for a tool occurs after all tracing activity on a device is complete and immediately prior to the finalization of the device by the OpenMP implementation.

4.1.5 OMPT Runtime Entry Points for Tools

The OMPT interface supports two principal sets of runtime entry points for tools. One set of runtime entry points enables a tool to register callbacks for OpenMP events and to inspect the state of an OpenMP thread while executing in a tool callback or a signal handler. The second set of runtime entry points enables a tool to trace activities on a device. When directed by the tracing interface, an OpenMP implementation will trace activities on a device, collect buffers full of trace records, and invoke callbacks on the host to process these records. Runtime entry points for tools in

an OpenMP implementation should not be global symbols since tools cannot rely on the visibility of such symbols in general.

In addition, the OMPT interface supports runtime entry points for two classes of lookup routines. The first class of lookup routines contains a single member: a routine that returns runtime entry points in the OMPT callback interface. The second class of lookup routines includes a unique lookup routine for each kind of device that can return runtime entry points in a device's OMPT tracing interface.

4.1.5.1 Entry Points in the OMPT Callback Interface

Entry points in the OMPT callback interface enable a tool to register callbacks for OpenMP events and to inspect the state of an OpenMP thread while executing in a tool callback or a signal handler. A tool obtains pointers to these runtime entry points using the lookup function passed to the tool's initializer for the callback interface.

4.1.5.1.1 `ompt_enumerate_states_t`

Summary

A runtime entry point known as **`ompt_enumerate_states`** with type signature **`ompt_enumerate_states_t`** enumerates the thread states supported by an OpenMP implementation.

Format

C / C++

```
typedef int (*ompt_enumerate_states_t)(
  int current_state,
  int *next_state,
  const char **next_state_name
);
```

C / C++

Description

An OpenMP implementation may support only a subset of the states defined by the **omp_states_t** enumeration type. In addition, an OpenMP implementation may support implementation-specific states. The **ompt_enumerate_states** runtime entry point enables a tool to enumerate the thread states supported by an OpenMP implementation.

When a thread state supported by an OpenMP implementation is passed as the first argument to the runtime entry point, the runtime entry point will assign the next thread state in the enumeration to the variable passed by reference as the runtime entry point's second argument and assign the name associated with the next thread state to the character pointer passed by reference as the third argument.

Whenever one or more states are left in the enumeration, the enumerate states runtime entry point will return 1. When the last state in the enumeration is passed as the first argument, the runtime entry point will return 0 indicating that the enumeration is complete.

Description of Arguments

The argument *current_state* must be a thread state supported by the OpenMP implementation. To begin enumerating the states that an OpenMP implementation supports, a tool should pass **omp_state_undefined** as *current_state*. Subsequent invocations of the runtime entry point by the tool should pass the value assigned to the variable passed by reference as the second argument to the previous call.

The argument *next_state* is a pointer to an integer where the entry point will return the value of the next state in the enumeration.

The argument *next_state_name* is a pointer to a character string pointer, where the entry point will return a string describing the next state.

Constraints on Arguments

Any string returned through the argument *next_state_name* must be immutable and defined for the lifetime of a program execution.

Note – The following example illustrates how a tool can enumerate all states supported by an OpenMP implementation. The example assumes that a function pointer to enumerate the thread states supported by an OpenMP implementation has previously been assigned to **ompt_enumerate_states_fn**.

C / C++

```
int state = omp_state_undefined;
const char *state_name;
while (ompt_enumerate_states_fn(state, &state, &state_name)) {
  // note that the runtime supports a state value "state"
  // associated with the name "state_name"
}
```

C / C++

Cross References

- **omp_state_t**, see Section 4.3.1.1 on page 551.

4.1.5.1.2 ompt_enumerate_mutex_impls_t

Summary

A runtime entry point known as **ompt_enumerate_mutex_impls** with type signature **ompt_enumerate_mutex_impls_t** enumerates the kinds of mutual exclusion implementations that an OpenMP implementation employs.

Format

C / C++

```
typedef int (*ompt_enumerate_mutex_impls_t)(
  int current_impl,
  int *next_impl,
  const char **next_impl_name
);

#define ompt_mutex_impl_unknown 0
```

C / C++

Description

An OpenMP implementation may implement mutual exclusion for locks, nest locks, critical sections, and atomic regions in several different ways. The **ompt_enumerate_mutex_impls** runtime entry point enables a tool to enumerate the kinds of mutual exclusion implementations that an OpenMP implementation employs. The value **ompt_mutex_impl_unknown** is reserved to indicate an invalid implementation.

When a mutex kind supported by an OpenMP implementation is passed as the first argument to the runtime entry point, the runtime entry point will assign the next mutex kind in the enumeration to the variable passed by reference as the runtime entry point's second argument and assign the name associated with the next mutex kind to the character pointer passed by reference as the third argument.

Whenever one or more mutex kinds are left in the enumeration, the runtime entry point to enumerate mutex implementations will return 1. When the last mutex kind in the enumeration is passed as the first argument, the runtime entry point will return 0 indicating that the enumeration is complete.

Description of Arguments

The argument *current_impl* must be a mutex implementation kind supported by an OpenMP implementation. To begin enumerating the mutex implementation kinds that an OpenMP implementation supports, a tool should pass **ompt_mutex_impl_unknown** as the first argument of the enumerate mutex kinds runtime entry point. Subsequent invocations of the runtime entry point by the tool should pass the value assigned to the variable passed by reference as the second argument to the previous call.

The argument *next_impl* is a pointer to an integer where the entry point will return the value of the next mutex implementation in the enumeration.

The argument *next_impl_name* is a pointer to a character string pointer, where the entry point will return a string describing the next mutex implementation.

Constraints on Arguments

Any string returned through the argument *next_impl_name* must be immutable and defined for the lifetime of a program execution.

Note – The following example illustrates how a tool can enumerate all types of mutex implementations supported by an OpenMP runtime. The example assumes that a function pointer to enumerate the mutex implementations supported by an OpenMP runtime has previously been assigned to **ompt_enumerate_mutex_impls_fn**.

C / C++

```
int kind = ompt_mutex_impl_unknown;
const char *impl_name;
while (ompt_enumerate_mutex_impls_fn(impl, &impl, &impl_name)) {
  // note that the runtime supports a mutex value "impl"
  // associated with the name "impl_name"
}
```

C / C++

4.1.5.1.3 ompt_set_callback_t

Summary

A runtime entry point known as **ompt_set_callback** with type signature **ompt_set_callback_t** registers a pointer to a tool callback that an OpenMP implementation will invoke when a host OpenMP event occurs.

Format

C / C++

```
typedef int (*ompt_set_callback_t)(
  ompt_callbacks_t which,
  ompt_callback_t callback
);
```

C / C++

Description

OpenMP implementations can inform tools about events that occur during the execution of an OpenMP program using callbacks. To register a tool callback for an OpenMP event on the current device, a tool uses the runtime entry point known as **ompt_set_callback** with type signature **ompt_set_callback_t**.

The return value of the **ompt_set_callback** runtime entry point may indicate several possible outcomes. Callback registration may fail if it is called outside the initializer for the callback interface, returning **omp_set_error**. Otherwise, the return value of **ompt_set_callback** indicates whether *dispatching* a callback leads to its invocation. A return value of **ompt_set_never** indicates that the callback will never be invoked at runtime. A return value of **ompt_set_sometimes** indicates that the callback will be invoked at runtime for an implementation-defined subset of associated event occurrences. A return value of **ompt_set_sometimes_paired** is similar to **ompt_set_sometimes**, but provides an additional guarantee for callbacks with an *endpoint* parameter. Namely, it guarantees that a callback with an *endpoint* value of **ompt_scope_begin** is invoked if and only if the same callback with *endpoint* value of **ompt_scope_end** will also be invoked sometime in the future. A return value of **ompt_set_always** indicates that the callback will be always invoked at runtime for associated event occurrences.

Description of Arguments

The argument *which* indicates the callback being registered.

The argument *callback* is a tool callback function.

A tool may pass a **NULL** value for *callback* to disable any callback associated with *which*. If disabling was successful, **ompt_set_always** is returned.

Constraints on Arguments

When a tool registers a callback for an event, the type signature for the callback must match the type signature appropriate for the event.

Cross References

- **ompt_callbacks_t** enumeration type, see Section 4.1.3.2 on page 389.
- **ompt_callback_t** type, see Section 4.1.3.4.1 on page 394.
- **ompt_get_callback_t** host callback type signature, see Section 4.1.5.1.4 on page 447.

TABLE 4.4: Return codes for **ompt_set_callback** and **ompt_set_trace_ompt**.

```
typedef enum ompt_set_result_t {
  ompt_set_error            = 0,
  ompt_set_never            = 1,
  ompt_set_sometimes        = 2,
  ompt_set_sometimes_paired = 3,
  ompt_set_always           = 4
} ompt_set_result_t;
```

4.1.5.1.4 ompt_get_callback_t

Summary

A runtime entry point known as **ompt_get_callback** with type signature **ompt_get_callback_t** retrieves a pointer to a tool callback routine (if any) that an OpenMP implementation will invoke when an OpenMP event occurs.

Format

C / C++

```
typedef int (*ompt_get_callback_t)(
  ompt_callbacks_t which,
  ompt_callback_t *callback
);
```

C / C++

Description

A tool uses the runtime entry point known as **ompt_get_callback** with type signature **ompt_get_callback_t** to obtain a pointer to the tool callback that an OpenMP implementation will invoke when a host OpenMP event occurs. If a non-**NULL** tool callback is registered for the specified event, the pointer to the tool callback will be assigned to the variable passed by reference as the second argument and the entry point will return 1; otherwise, it will return 0. If the entry point returns 0, the value of the variable passed by reference as the second argument is undefined.

Description of Arguments

The argument *which* indicates the callback being inspected.

The argument *callback* returns a pointer to the callback being inspected.

Constraints on Arguments

The second argument passed to the entry point must be a reference to a variable of specified type.

Cross References

- **ompt_callbacks_t** enumeration type, see Section 4.1.3.2 on page 389.
- **ompt_callback_t** type, see Section 4.1.3.4.1 on page 394.
- **ompt_set_callback_t** type signature, see Section 4.1.5.1.3 on page 445.

4.1.5.1.5 ompt_get_thread_data_t

Summary

A runtime entry point known as **ompt_get_thread_data** with type signature **ompt_get_thread_data_t** returns the address of the thread data object for the current thread.

Format

C / C++

```
typedef ompt_data_t *(*ompt_get_thread_data_t)(void);
```

C / C++

Description

Each OpenMP thread has an associated thread data object of type **ompt_data_t**. A tool uses the runtime entry point known as **ompt_get_thread_data** with type signature **ompt_get_thread_data_t** to obtain a pointer to the thread data object, if any, associated with the current thread. If the current thread is unknown to the OpenMP runtime, the entry point returns **NULL**.

A tool may use a pointer to an OpenMP thread's data object obtained from this runtime entry point to inspect or modify the value of the data object. When an OpenMP thread is created, its data object will be initialized with value **ompt_data_none**.

This runtime entry point is *async signal safe*.

Cross References

- **ompt_data_t** type, see Section 4.1.3.4.3 on page 395.

4.1.5.1.6 ompt_get_num_procs_t

Summary

A runtime entry point known as **ompt_get_num_procs** with type signature **ompt_get_num_procs_t** returns the number of processors currently available to the execution environment on the host device.

Format

C / C++

```
typedef int (*ompt_get_num_procs_t)(void);
```

C / C++

Binding

The binding thread set for runtime entry point known as **ompt_get_num_procs** is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Description

The **ompt_get_num_procs** runtime entry point returns the number of processors that are available on the host device at the time the routine is called. This value may change between the time that it is determined and the time that it is read in the calling context due to system actions outside the control of the OpenMP implementation.

This runtime entry point is *async signal safe*.

4.1.5.1.7 ompt_get_num_places_t

Summary

A runtime entry point known as **ompt_get_num_places** with type signature **ompt_get_num_places_t** returns the number of places available to the execution environment in the place list.

Format

C / C++

```
typedef int (*ompt_get_num_places_t)(void);
```

C / C++

Binding

The binding thread set for the region of the runtime entry point known as **ompt_get_num_places** is all threads on a device. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

Description

The runtime entry point known as **ompt_get_num_places** returns the number of places in the place list. This value is equivalent to the number of places in the *place-partition-var* ICV in the execution environment of the initial task.

This runtime entry point is *async signal safe*.

Cross References

- *place-partition-var* ICV, see Section 2.4 on page 49.
- **OMP_PLACES** environment variable, see Section 5.5 on page 567.

4.1.5.1.8 ompt_get_place_proc_ids_t

Summary

A runtime entry point known as **ompt_get_place_proc_ids** with type signature **ompt_get_place_proc_ids_t** returns the numerical identifiers of the processors available to the execution environment in the specified place.

1 **Format**

C / C++

```
typedef int (*ompt_get_place_proc_ids_t)(
    int place_num,
    int ids_size,
    int *ids
  );
```

C / C++

2 **Binding**

3 The binding thread set for the region of the runtime entry point known as
4 **ompt_get_place_proc_ids** is all threads on a device. The effect of executing this routine is
5 not related to any specific region corresponding to any construct or API routine.

6 **Description**

7 The runtime entry point known as **ompt_get_place_proc_ids** with type signature
8 **ompt_get_place_proc_ids_t** returns the numerical identifiers of each processor associated
9 with the specified place. The numerical identifiers returned are non-negative, and their meaning is
10 implementation defined.

11 **Description of Arguments**

12 The argument *place_num* specifies the place being queried.

13 The argument *ids_size* indicates the size of the result array specified by argument *ids*.

14 The argument *ids* is an array where the routine can return a vector of processor identifiers in the
15 specified place.

16 **Effect**

17 If the array *ids* of size *ids_size* is large enough to contain all identifiers, they are returned in *ids* and
18 their order in the array is implementation defined.

19 Otherwise, if the *ids* array is too small, the values in *ids* when the function returns are unspecified.

20 In both cases, the routine returns the number of numerical identifiers available to the execution
21 environment in the specified place.

4.1.5.1.9 ompt_get_place_num_t

Summary

A runtime entry point known as **ompt_get_place_num** with type signature **ompt_get_place_num_t** returns the place number of the place to which the current thread is bound.

Format

C / C++

```
typedef int (*ompt_get_place_num_t)(void);
```

C / C++

Binding

The binding thread set of the runtime entry point known as **ompt_get_place_num** is the current thread.

Description

When the current thread is bound to a place, the runtime entry point known as **ompt_get_place_num** returns the place number associated with the thread. The returned value is between 0 and one less than the value returned by runtime entry point known as **ompt_get_num_places**, inclusive. When the current thread is not bound to a place, the routine returns -1.

This runtime entry point is *async signal safe*.

4.1.5.1.10 ompt_get_partition_place_nums_t

Summary

A runtime entry point known as **ompt_get_partition_place_nums** with type signature **ompt_get_partition_place_nums_t** returns the list of place numbers corresponding to the places in the *place-partition-var* ICV of the innermost implicit task.

Format

C / C++

```
typedef int (*ompt_get_partition_place_nums_t)(
    int place_nums_size,
    int *place_nums
  );
```

C / C++

Binding

The binding task set for the runtime entry point known as **ompt_get_partition_place_nums** is the current implicit task.

Description

The runtime entry point known as **ompt_get_partition_place_nums** with type signature **ompt_get_partition_place_nums_t** returns the list of place numbers corresponding to the places in the *place-partition-var* ICV of the innermost implicit task.

This runtime entry point is *async signal safe*.

Description of Arguments

The argument *place_nums_size* indicates the size of the result array specified by argument *place_nums*.

The argument *place_nums* is an array where the routine can return a vector of place identifiers.

Effect

If the array *place_nums* of size *place_nums_size* is large enough to contain all identifiers, they are returned in *place_nums* and their order in the array is implementation defined.

Otherwise, if the *place_nums* array is too small, the values in *place_nums* when the function returns are unspecified.

In both cases, the routine returns the number of places in the *place-partition-var* ICV of the innermost implicit task.

Cross References

- *place-partition-var* ICV, see Section 2.4 on page 49.
- **OMP_PLACES** environment variable, see Section 5.5 on page 567.

4.1.5.1.11 `ompt_get_proc_id_t`

Summary

A runtime entry point known as **`ompt_get_proc_id`** with type signature **`ompt_get_proc_id_t`** returns the numerical identifier of the processor of the current thread.

Format

C / C++

```
typedef int (*ompt_get_proc_id_t)(void);
```

C / C++

Binding

The binding thread set for the runtime entry point known as **`ompt_get_proc_id`** is the current thread.

Description

The runtime entry point known as **`ompt_get_proc_id`** returns the numerical identifier of the processor of the current thread. A defined numerical identifier is non-negative, and its meaning is implementation defined. A negative number indicates a failure to retrieve the numerical identifier.

This runtime entry point is *async signal safe*.

4.1.5.1.12 `ompt_get_state_t`

Summary

A runtime entry point known as **`ompt_get_state`** with type signature **`ompt_get_state_t`** returns the state and the wait identifier of the current thread.

Format

C / C++

```
typedef omp_state_t (*ompt_get_state_t)(
  omp_wait_id_t *wait_id
);
```

C / C++

Description

Each OpenMP thread has an associated state and a wait identifier. If a thread's state indicates that the thread is waiting for mutual exclusion, the thread's wait identifier will contain an opaque handle that indicates the data object upon which the thread is waiting.

To retrieve the state and wait identifier for the current thread, a tool uses the runtime entry point known as **ompt_get_state** with type signature **ompt_get_state_t**.

If the returned state indicates that the thread is waiting for a lock, nest lock, critical section, atomic region, or ordered region the value of the thread's wait identifier will be assigned to a non-**NULL** wait identifier passed as an argument.

This runtime entry point is *async signal safe*.

Description of Arguments

The argument *wait_id* is a pointer to an opaque handle available to receive the value of the thread's wait identifier. If the *wait_id* pointer is not **NULL**, the entry point will assign the value of the thread's wait identifier **wait_id*. If the returned state is not one of the specified wait states, the value of **wait_id* is undefined after the call.

Constraints on Arguments

The argument passed to the entry point must be a reference to a variable of the specified type or **NULL**.

Cross References

- **omp_wait_id_t** type, see Section 4.3.1.3 on page 558.

4.1.5.1.13 ompt_get_parallel_info_t

Summary

A runtime entry point known as **ompt_get_parallel_info** with type signature **ompt_get_parallel_info_t** returns information about the parallel region, if any, at the specified ancestor level for the current execution context.

Format

C / C++

```
typedef int (*ompt_get_parallel_info_t)(
  int ancestor_level,
  ompt_data_t **parallel_data,
  int *team_size
);
```

C / C++

Description

During execution, an OpenMP program may employ nested parallel regions. To obtain information about a parallel region, a tool uses the runtime entry point known as **ompt_get_parallel_info** with type signature **ompt_get_parallel_info_t**. This runtime entry point can be used to obtain information about the current parallel region, if any, and any enclosing parallel regions for the current execution context.

The entry point returns 2 if there is a parallel region at the specified ancestor level and the information is available, 1 if there is a parallel region at the specified ancestor level but the information is currently unavailable, and 0 otherwise.

A tool may use the pointer to a parallel region's data object that it obtains from this runtime entry point to inspect or modify the value of the data object. When a parallel region is created, its data object will be initialized with the value **ompt_data_none**.

This runtime entry point is *async signal safe*.

Description of Arguments

The argument *ancestor_level* specifies the parallel region of interest to a tool by its ancestor level. Ancestor level 0 refers to the innermost parallel region; information about enclosing parallel regions may be obtained using larger ancestor levels.

If a parallel region exists at the specified ancestor level and the information is currently available, information will be returned in the variables *parallel_data* and *team_size* passed by reference to the entry point. Specifically, a reference to the parallel region's associated data object will be assigned to **parallel_data* and the number of threads in the parallel region's team will be assigned to **team_size*.

If no enclosing parallel region exists at the specified ancestor level, or the information is currently unavailable, the values of variables passed by reference **parallel_data* and **team_size* will be undefined when the entry point returns.

Constraints on Arguments

While argument *ancestor_level* is passed by value, all other arguments to the entry point must be references to variables of the specified types.

Restrictions

Between a *parallel-begin* event and an *implicit-task-begin* event, a call to **ompt_get_parallel_info(0,...)** may return information about the outer parallel team, the new parallel team or an inconsistent state.

If a thread is in the state **omp_state_wait_barrier_implicit_parallel**, a call to **ompt_get_parallel_info** may return a pointer to a copy of the specified parallel region's *parallel_data* rather than a pointer to the data word for the region itself. This convention enables the master thread for a parallel region to free storage for the region immediately after the region ends, yet avoid having some other thread in the region's team potentially reference the region's *parallel_data* object after it has been freed.

Cross References

- **ompt_data_t** type, see Section 4.1.3.4.3 on page 395.

4.1.5.1.14 ompt_get_task_info_t

Summary

A runtime entry point known as **ompt_get_task_info** with type signature **ompt_get_task_info_t** provides information about the task, if any, at the specified ancestor level in the current execution context.

Format

C / C++

```
typedef int (*ompt_get_task_info_t)(
  int ancestor_level,
  int *type,
  ompt_data_t **task_data,
  omp_frame_t **task_frame,
  ompt_data_t **parallel_data,
  int *thread_num
);
```

C / C++

Description

During execution, an OpenMP thread may be executing an OpenMP task. Additionally, the thread's stack may contain procedure frames associated with suspended OpenMP tasks or OpenMP runtime system routines. To obtain information about any task on the current thread's stack, a tool uses the runtime entry point known as **`ompt_get_task_info`** with type signature **`ompt_get_task_info_t`**.

Ancestor level 0 refers to the active task; information about other tasks with associated frames present on the stack in the current execution context may be queried at higher ancestor levels.

The **`ompt_get_task_info`** runtime entry point returns 2 if there is a task region at the specified ancestor level and the information is available, 1 if there is a task region at the specified ancestor level but the information is currently unavailable, and 0 otherwise.

If a task exists at the specified ancestor level and the information is available, information will be returned in the variables passed by reference to the entry point. If no task region exists at the specified ancestor level or the information is unavailable, the values of variables passed by reference to the entry point will be undefined when the entry point returns.

A tool may use a pointer to a data object for a task or parallel region that it obtains from this runtime entry point to inspect or modify the value of the data object. When either a parallel region or a task region is created, its data object will be initialized with the value **`ompt_data_none`**.

This runtime entry point is *async signal safe*.

Description of Arguments

The argument *ancestor_level* specifies the task region of interest to a tool by its ancestor level. Ancestor level 0 refers to the active task; information about ancestor tasks found in the current execution context may be queried at higher ancestor levels.

The argument *type* returns the task type if the argument is not **`NULL`**.

The argument *task_data* returns the task data if the argument is not **`NULL`**.

The argument *task_frame* returns the task frame pointer if the argument is not **`NULL`**.

The argument *parallel_data* returns the parallel data if the argument is not **`NULL`**.

The argument *thread_num* returns the thread number if the argument is not **`NULL`**.

Effect

If the runtime entry point returns 0 or 1, no argument is modified. Otherwise, the entry point has the effects described below.

If a non-**`NULL`** value was passed for *type*, the value returned in **type* represents the type of the task at the specified level. Task types that a tool may observe on a thread's stack include initial, implicit, explicit, and target tasks.

If a non-**NULL** value was passed for *task_data*, the value returned in **task_data* is a pointer to a data word associated with the task at the specified level.

If a non-**NULL** value was passed for *task_frame*, the value returned in **task_frame* is a pointer to the **omp_frame_t** structure associated with the task at the specified level. Appendix D discusses an example that illustrates the use of **omp_frame_t** structures with multiple threads and nested parallelism.

If a non-**NULL** value was passed for *parallel_data*, the value returned in **parallel_data* is a pointer to a data word associated with the parallel region containing the task at the specified level. If the task at the specified level is an initial task, the value of **parallel_data* will be **NULL**.

If a non-**NULL** value was passed for *thread_num*, the value returned in **thread_num* indicates the number of the thread in the parallel region executing the task.

Cross References

- **ompt_data_t** type, see Section 4.1.3.4.3 on page 395.
- **omp_frame_t** type, see Section 4.3.1.2 on page 556.
- **ompt_task_type_t** type, see Section 4.1.3.4.16 on page 400.

4.1.5.1.15 ompt_get_target_info_t

Summary

A runtime entry point known as **ompt_get_target_info** with type signature **ompt_get_target_info_t** returns identifiers that specify a thread's current target region and target operation id, if any.

Format

C / C++

```
typedef int (*ompt_get_target_info_t)(
  uint64_t *device_num,
  ompt_id_t *target_id,
  ompt_id_t *host_op_id
);
```

C / C++

Description

A tool can query whether an OpenMP thread is in a target region by invoking the entry point known as **ompt_get_target_info** with type signature **ompt_get_target_info_t**. This runtime entry point returns 1 if the current thread is in a target region and 0 otherwise. If the entry point returns 0, the values of the variables passed by reference as its arguments are undefined.

If the current thread is in a target region, the entry point will return information about the current device, active target region, and active host operation, if any.

This runtime entry point is *async signal safe*.

Description of Arguments

If the host is in a **target** region, *device_num* returns the target device.

If the host is in a **target** region, *target_id* returns the **target** region identifier.

If the current thread is in the process of initiating an operation on a target device (e.g., copying data to or from an accelerator or launching a kernel) *host_op_id* returns the identifier for the operation; otherwise, *host_op_id* returns **ompt_id_none**.

Constraints on Arguments

Arguments passed to the entry point must be valid references to variables of the specified types.

Cross References

- **ompt_id_t** type, see Section 4.1.3.4.2 on page 394.

4.1.5.1.16 ompt_get_num_devices_t

Summary

A runtime entry point known as **ompt_get_num_devices** with type signature **ompt_get_num_devices_t** returns the number of available devices.

Format

C / C++

```
typedef int (*ompt_get_num_devices_t)(void);
```

C / C++

Description

An OpenMP program may execute on one or more devices. A tool may determine the number of devices available to an OpenMP program by invoking a runtime entry point known as **ompt_get_num_devices** with type signature **ompt_get_num_devices_t**.

This runtime entry point is *async signal safe*.

4.1.5.1.17 ompt_get_unique_id_t

Summary

A runtime entry point known as **ompt_get_unique_id** with type signature **ompt_get_unique_id_t** returns a unique number.

Format

C / C++

```
typedef uint64_t (*ompt_get_unique_id_t)(void);
```

C / C++

Description

A tool may obtain a number that is unique for the duration of an OpenMP program by invoking a runtime entry point known as **ompt_get_unique_id** with type signature **ompt_get_unique_id_t**. Successive invocations may not result in consecutive or even increasing numbers.

This runtime entry point is *async signal safe*.

4.1.5.2 Entry Points in the OMPT Device Tracing Interface

4.1.5.2.1 ompt_get_device_num_procs_t

Summary

A runtime entry point for a device known as **ompt_get_device_num_procs** with type signature **ompt_get_device_num_procs_t** returns the number of processors currently available to the execution environment on the specified device.

Format

C / C++

```
typedef int (*ompt_get_device_num_procs_t)(
  ompt_device_t *device
);
```

C / C++

Description

A runtime entry point for a device known as **ompt_get_device_num_procs** with type signature **ompt_get_device_num_procs_t** returns the number of processors that are available on the device at the time the routine is called. This value may change between the time that it is determined and the time that it is read in the calling context due to system actions outside the control of the OpenMP implementation.

Description of Arguments

The argument *device* is a pointer to an opaque object that represents the target device instance. The pointer to the device instance object is used by functions in the device tracing interface to identify the device being addressed.

Cross References

- **ompt_device_t**, see Section 4.1.3.4.4 on page 395.

4.1.5.2.2 ompt_get_device_time_t

Summary

A runtime entry point for a device known as **ompt_get_device_time** with type signature **ompt_get_device_time_t** returns the current time on the specified device.

Format

C / C++

```
typedef ompt_device_time_t (*ompt_get_device_time_t)(
  ompt_device_t *device
);
```

C / C++

Description

Host and target devices are typically distinct and run independently. If host and target devices are different hardware components, they may use different clock generators. For this reason, there may be no common time base for ordering host-side and device-side events.

A runtime entry point for a device known as **ompt_get_device_time** with type signature **ompt_get_device_time_t** returns the current time on the specified device. A tool can use this information to align time stamps from different devices.

Description of Arguments

The argument *device* is a pointer to an opaque object that represents the target device instance. The pointer to the device instance object is used by functions in the device tracing interface to identify the device being addressed.

Cross References

- **ompt_device_t**, see Section 4.1.3.4.4 on page 395.
- **ompt_device_time_t**, see Section 4.1.3.4.5 on page 395.

4.1.5.2.3 ompt_translate_time_t

Summary

A runtime entry point for a device known as **ompt_translate_time** with type signature **ompt_translate_time_t** translates a time value obtained from the specified device to a corresponding time value on the host device.

Format

C / C++

```
typedef double (*ompt_translate_time_t)(
  ompt_device_t *device,
  ompt_device_time_t time
);
```

C / C++

Description

A runtime entry point for a device known as **ompt_translate_time** with type signature **ompt_translate_time_t** translates a time value obtained from the specified device to a corresponding time value on the host device. The returned value for the host time has the same meaning as the value returned from **omp_get_wtime**.

Note –

The accuracy of time translations may degrade if they are not performed promptly after a device time value is received if either the host or device vary their clock speeds. Prompt translation of device times to host times is recommended.

Description of Arguments

The argument *device* is a pointer to an opaque object that represents the target device instance. The pointer to the device instance object is used by functions in the device tracing interface to identify the device being addressed.

The argument *time* is a time from the specified device.

Cross References

- **ompt_device_t**, see Section 4.1.3.4.4 on page 395.
- **ompt_device_time_t**, see Section 4.1.3.4.5 on page 395.

4.1.5.2.4 ompt_set_trace_ompt_t

Summary

A runtime entry point for a device known as **ompt_set_trace_ompt** with type signature **ompt_set_trace_ompt_t** enables or disables the recording of trace records for one or more types of OMPT events.

1 **Format**

C / C++

```
typedef int (*ompt_set_trace_ompt_t)(
  ompt_device_t *device,
  unsigned int enable,
  unsigned int etype
);
```

C / C++

2 **Description of Arguments**

3 The argument *device* is a pointer to an opaque object that represents the target device instance. The
4 pointer to the device instance object is used by functions in the device tracing interface to identify
5 the device being addressed.

6 The argument *enable* indicates whether tracing should be enabled or disabled for the event or
7 events specified by argument *etype*. A positive value for *enable* indicates that recording of one or
8 more events specified by *etype* should be enabled; a value of 0 for *enable* indicates that recording of
9 events should be disabled by this invocation.

10 An argument *etype* value 0 indicates that traces for all event types will be enabled or disabled.
11 Passing a positive value for *etype* inidicates that recording should be enabled or disabled for the
12 event in **ompt_callbacks_t** that matches *etype*.

13 **Effect**

14 Table 4.5 shows the possible return codes for **ompt_set_trace_ompt**. If a single invocation of
15 **ompt_set_trace_ompt** is used to enable or disable more than one event (i.e., **etype**=0), the
16 return code will be 3 if tracing is possible for one or more events but not for others.

TABLE 4.5: Meaning of return codes for **ompt_set_trace_ompt** and **ompt_set_trace_native**.

Return Code	Meaning
0	error
1	event will never occur

table continued on next page

table continued from previous page

Return Code	Meaning
2	event may occur but no tracing is possible
3	event may occur and will be traced when convenient
4	event may occur and will always be traced if event occurs

Cross References

- **ompt_callbacks_t**, see Section 4.1.3.2 on page 389.
- **ompt_device_t**, see Section 4.1.3.4.4 on page 395.

4.1.5.2.5 ompt_set_trace_native_t

Summary

A runtime entry point for a device known as **ompt_set_trace_native** with type signature **ompt_set_trace_native_t** enables or disables the recording of native trace records for a device.

Format

C / C++

```
typedef int (*ompt_set_trace_native_t)(
  ompt_device_t *device,
  int enable,
  int flags
);
```

C / C++

Description

This interface is designed for use by a tool with no knowledge about an attached device. If a tool knows how to program a particular attached device, it may opt to invoke native control functions directly using pointers obtained through the *lookup* function associated with the device and described in the *documentation* string that is provided to the device initializer callback.

1 **Description of Arguments**

2 The argument *device* is a pointer to an opaque object that represents the target device instance. The
3 pointer to the device instance object is used by functions in the device tracing interface to identify
4 the device being addressed.

5 The argument *enable* indicates whether recording of events should be enabled or disabled by this
6 invocation.

7 The argument *flags* specifies the kinds of native device monitoring to enable or disable. Each kind
8 of monitoring is specified by a flag bit. Flags can be composed by using logical `or` to combine
9 enumeration values from type **ompt_native_mon_flags_t**. Table 4.5 shows the possible
10 return codes for **ompt_set_trace_native**. If a single invocation of
11 **ompt_set_trace_ompt** is used to enable/disable more than one kind of monitoring, the return
12 code will be 3 if tracing is possible for one or more kinds of monitoring but not for others.

13 To start, pause, or stop tracing for a specific target device associated with the handle *device*, a tool
14 calls the functions **ompt_start_trace**, **ompt_pause_trace**, or **ompt_stop_trace**.

15 **Cross References**

16 • **ompt_device_t**, see Section 4.1.3.4.4 on page 395.

17 ### 4.1.5.2.6 ompt_start_trace_t

18 **Summary**

19 A runtime entry point for a device known as **ompt_start_trace** with type signature
20 **ompt_start_trace_t** starts tracing of activity on a specific device.

21 **Format**

C / C++

```
typedef int (*ompt_start_trace_t)(
  ompt_device_t *device,
  ompt_callback_buffer_request_t request,
  ompt_callback_buffer_complete_t complete
);
```

C / C++

Description

A tool may initiate tracing on a device by invoking the device's **ompt_start_trace** runtime entry point.

Under normal operating conditions, every event buffer provided to a device by a tool callback will be returned to the tool before the OpenMP runtime shuts down. If an exceptional condition terminates execution of an OpenMP program, the OpenMP runtime may not return buffers provided to the device.

An invocation of **ompt_start_trace** returns 1 if the command succeeded and 0 otherwise.

Description of Arguments

The argument *device* is a pointer to an opaque object that represents the target device instance. The pointer to the device instance object is used by functions in the device tracing interface to identify the device being addressed.

The argument *buffer request* specifies a tool callback that will supply a device with a buffer to deposit events.

The argument *buffer complete* specifies a tool callback that will be invoked by the OpenMP implmementation to empty a buffer containing event records.

Cross References

- **ompt_device_t**, see Section 4.1.3.4.4 on page 395.
- **ompt_callback_buffer_request_t**, see Section 4.1.4.2.24 on page 434.
- **ompt_callback_buffer_complete_t**, see Section 4.1.4.2.25 on page 435.

4.1.5.2.7 ompt_pause_trace_t

Summary

A runtime entry point for a device known as **ompt_pause_trace** with type signature **ompt_pause_trace_t** pauses or restarts activity tracing on a specific device.

C / C++

```
typedef int (*ompt_pause_trace_t)(
  ompt_device_t *device,
  int begin_pause
);
```

C / C++

Description

A tool may pause or resume tracing on a device by invoking the device's **ompt_pause_trace** runtime entry point. An invocation of **ompt_pause_trace** returns 1 if the command succeeded and 0 otherwise.

Redundant pause or resume commands are idempotent and will return 1 indicating success.

Description of Arguments

The argument *device* is a pointer to an opaque object that represents the target device instance. The pointer to the device instance object is used by functions in the device tracing interface to identify the device being addressed.

The argument *begin_pause* indicates whether to pause or resume tracing. To resume tracing, zero should be supplied for *begin_pause*.

Cross References

- **ompt_device_t**, see Section 4.1.3.4.4 on page 395.

4.1.5.2.8 ompt_stop_trace_t

Summary

A runtime entry point for a device known as **ompt_stop_trace** with type signature **ompt_stop_trace_t** stops tracing for a device.

C / C++

```
typedef int (*ompt_stop_trace_t)(
  ompt_device_t *device
);
```

C / C++

Description

A tool may halt tracing on a device and request that the device flush any pending trace records by invoking the **ompt_stop_trace** runtime entry point for the device. An invocation of **ompt_stop_trace** returns 1 if the command succeeded and 0 otherwise.

Description of Arguments

The argument *device* is a pointer to an opaque object that represents the target device instance. The pointer to the device instance object is used by functions in the device tracing interface to identify the device being addressed.

Cross References

- **ompt_device_t**, see Section 4.1.3.4.4 on page 395.

4.1.5.2.9 ompt_advance_buffer_cursor_t

Summary

A runtime entry point for a device known as **ompt_advance_buffer_cursor** with type signature **ompt_advance_buffer_cursor_t** advances a trace buffer cursor to the next record.

Format

C / C++

```
typedef int (*ompt_advance_buffer_cursor_t)(
  ompt_buffer_t *buffer,
  size_t size,
  ompt_buffer_cursor_t current,
  ompt_buffer_cursor_t *next
);
```

C / C++

Description

It returns *true* if the advance is successful and the next position in the buffer is valid.

Description of Arguments

The argument *device* is a pointer to an opaque object that represents the target device instance. The pointer to the device instance object is used by functions in the device tracing interface to identify the device being addressed.

The argument *buffer* indicates a trace buffer associated with the cursors.

The argument *size* indicates the size of *buffer* in bytes.

The argument *current* is an opaque buffer cursor.

The argument *next* returns the next value of a opaque buffer cursor.

Cross References

- **ompt_device_t**, see Section 4.1.3.4.4 on page 395.
- **ompt_buffer_cursor_t**, see Section 4.1.3.4.7 on page 396.

4.1.5.2.10 ompt_get_record_type_t

Summary

A runtime entry point for a device known as **ompt_get_record_type** with type signature **ompt_get_record_type_t** inspects the type of a trace record for a device.

Format

C / C++

```
typedef ompt_record_type_t (*ompt_get_record_type_t)(
  ompt_buffer_t *buffer,
  ompt_buffer_cursor_t current
);
```

C / C++

Description

Trace records for a device may be in one of two forms: a *native* record format, which may be device-specific, or an *OMPT* record format, where each trace record corresponds to an OpenMP *event* and fields in the record structure are mostly the arguments that would be passed to the OMPT callback for the event.

A runtime entry point for a device known as **ompt_get_record_type** with type signature **ompt_get_record_type_t** inspects the type of a trace record and indicates whether the record at the current position in the provided trace buffer is an OMPT record, a native record, or an invalid record. An invalid record type is returned if the cursor is out of bounds.

Description of Arguments

The argument *buffer* indicates a trace buffer.

The argument *current* is an opaque buffer cursor.

Cross References

- **ompt_buffer_t**, see Section 4.1.3.4.6 on page 396.
- **ompt_buffer_cursor_t**, see Section 4.1.3.4.7 on page 396.

4.1.5.2.11 ompt_get_record_ompt_t

Summary

A runtime entry point for a device known as **ompt_get_record_ompt** with type signature **ompt_get_record_ompt_t** obtains a pointer to an OMPT trace record from a trace buffer associated with a device.

Format

C / C++

```
typedef ompt_record_ompt_t *(*ompt_get_record_ompt_t)(
  ompt_buffer_t *buffer,
  ompt_buffer_cursor_t current
);
```

C / C++

Description

This function returns a pointer that may point to a record in the trace buffer, or it may point to a record in thread local storage where the information extracted from a record was assembled. The information available for an event depends upon its type.

The return value of type **ompt_record_ompt_t** defines a union type that can represent information for any OMPT event record type. Another call to the runtime entry point may overwrite the contents of the fields in a record returned by a prior invocation.

Description of Arguments

The argument *buffer* indicates a trace buffer.

The argument *current* is an opaque buffer cursor.

Cross References

- **ompt_record_ompt_t**, see Section 4.1.3.3.4 on page 393.
- **ompt_device_t**, see Section 4.1.3.4.4 on page 395.
- **ompt_buffer_cursor_t**, see Section 4.1.3.4.7 on page 396.

4.1.5.2.12 ompt_get_record_native_t

Summary

A runtime entry point for a device known as **ompt_get_record_native** with type signature **ompt_get_record_native_t** obtains a pointer to a native trace record from a trace buffer associated with a device.

Format

C / C++

```
typedef void *(*ompt_get_record_native_t)(
  ompt_buffer_t *buffer,
  ompt_buffer_cursor_t current,
  ompt_id_t *host_op_id
);
```

C / C++

Description

The pointer returned may point into the specified trace buffer, or into thread local storage where the information extracted from a trace record was assembled. The information available for a native event depends upon its type. If the function returns a non-NULL result, it will also set **`*host_op_id`** to identify host-side identifier for the operation associated with the record. A subsequent call to **`ompt_get_record_native`** may overwrite the contents of the fields in a record returned by a prior invocation.

Description of Arguments

The argument *buffer* indicates a trace buffer.

The argument *current* is an opaque buffer cursor.

The argument *host_op_id* is a pointer to an identifier that will be returned by the function. The entry point will set **host_op_id* to the value of a host-side identifier for an operation on a target device that was created when the operation was initiated by the host.

Cross References

- **`ompt_id_t`**, see Section 4.1.3.4.2 on page 394.
- **`ompt_buffer_t`**, see Section 4.1.3.4.6 on page 396.
- **`ompt_buffer_cursor_t`**, see Section 4.1.3.4.7 on page 396.

4.1.5.2.13 `ompt_get_record_abstract_t`

Summary

A runtime entry point for a device known as **`ompt_get_record_abstract`** with type signature **`ompt_get_record_abstract_t`** summarizes the context of a native (device-specific) trace record.

Format

C / C++

```
typedef ompt_record_abstract_t *
(*ompt_get_record_abstract_t)(
  void *native_record
);
```

C / C++

Description

An OpenMP implementation may execute on a device that logs trace records in a native (device-specific) format unknown to a tool. A tool can use the **ompt_get_record_abstract** runtime entry point for the device with type signature **ompt_get_record_abstract_t** to decode a native trace record that it does not understand into a standard form that it can interpret.

Description of Arguments

The argument *native_record* is a pointer to a native trace record.

Cross References

- **ompt_record_abstract_t**, see Section 4.1.3.3.3 on page 391.

4.1.5.3 Lookup Entry Point

4.1.5.3.1 ompt_function_lookup_t

Summary

A tool uses a lookup routine with type signature **ompt_function_lookup_t** to obtain pointers to runtime entry points that are part of the OMPT interface.

Format

C / C++

```
typedef void (*ompt_interface_fn_t)(void);

typedef ompt_interface_fn_t (*ompt_function_lookup_t) (
  const char *interface_function_name
);
```

C / C++

Description

An OpenMP implementation provides a pointer to a lookup routine as an argument to tool callbacks used to initialize tool support for monitoring an OpenMP device using either tracing or callbacks.

When an OpenMP implementation invokes a tool initializer to configure the OMPT callback interface, the OpenMP implementation will pass the initializer a lookup function that the tool can use to obtain pointers to runtime entry points that implement routines that are part of the OMPT callback interface.

When an OpenMP implementation invokes a tool initializer to configure the OMPT tracing interface for a device, the Open implementation will pass the device tracing initializer a lookup function that the tool can use to obtain pointers to runtime entry points that implement tracing control routines appropriate for that device.

A tool can call the lookup function to obtain a pointer to a runtime entry point.

Description of Arguments

The argument *interface_function_name* is a C string that represents the name of a runtime entry point.

Cross References

- Entry points in the OMPT callback interface, see Table 4.1 on page 382 for a list and Section 4.1.5.1 on page 441 for detailed definitions.
- Tool initializer for a device's OMPT tracing interface, Section 4.1.1.4 on page 383.
- Entry points in the OMPT tracing interface, see Table 4.3 on page 386 for a list and Section 4.1.5.2 on page 461 for detailed definitions.
- Tool initializer for the OMPT callback interface, Section 4.1.4.1.1 on page 404

4.2 OMPD

The OMPD interface is designed to allow a *third-party tool* such as a debugger to inspect the OpenMP state of a live program or core in an implementation agnostic manner. That is, a tool that uses OMPD should work with any conforming OpenMP implementation. The model for OMPD is that an OpenMP implementor provides a shared library plugin that a third-party tool can load. Using the interface exported by the plugin, the external tool can inspect the OpenMP state of a program using that implementation of OpenMP. In order to satisfy requests from the third-party

tool, the OMPD plugin may need to read data from, or find the addresses of symbols in, the OpenMP program. The plugin does this by using the callback interface the third-party tool must make available to the OMPD plugin.

The diagram shown in Section E on page 637 shows how the different components fits together. The third-party tool loads the OMPD plugin that matches the OpenMP runtime being used by the OpenMP program. The plugin exports the API defined later in this document, which the tool uses to get OpenMP information about the OpenMP program. The OMPD plugin will need to look up the symbols, or read data out of the program. It does not do this directly, but instead asks the tool to perform these operations for it using a callback interface exported by the tool.

This architectural layout insulates the tool from the details of the internal structure of the OpenMP runtime. Similarly, the OMPD plugin does not need to be concerned about how to access the OpenMP program. Decoupling the plugin and tool in this way allows for great flexibility in how the OpenMP program and tool are deployed, so that, for example, there is no requirement that tool and OpenMP program execute on the same machine. Generally the tool does not interact directly with the OpenMP runtime in the OpenMP program, and instead uses the OMPD plugin for this purpose. However, there are a few instances where the tool does need to access the OpenMP runtime directly. These cases fall into two broad categories. The first is during initialization, where the tool needs to be able to look up symbols and read variables in OpenMP runtime in order to identify the OMPD plugin it should use. This is discussed in Sections 4.3.2.2 and 4.3.2.3.

The second category relates to arranging for the tool to be notified when certain events occur during the execution of the OpenMP program. The model used for this purpose is that the OpenMP implementation is required to define certain symbols in the runtime code. This is discussed in Section 4.2.5.1. Each of these symbols corresponds to an event type. The runtime must ensure that control passes through the appropriate named location when events occur. If the tool wants to get notification of an event, it can plant a breakpoint at the matching location.

The code locations can, but do not need to, be functions. They can, for example, simply be labels. However, the names must have external `C` linkage.

4.2.1 Activating an OMPD Tool

The tool and the OpenMP program the tool controls exist as separate processes.; thus coordination is required between the OpenMP runtime and the external tool for OMPD to be used successfully.

4.2.1.1 Enabling the Runtime for OMPD

In order to support external tools, the OpenMP runtime may need to collect and maintain information that it might otherwise not do, perhaps for performance reasons, or because it is not otherwise needed. The OpenMP runtime collects whatever information is necessary to support OMPD if at least one of the following conditions is true:

1. the environment variable **OMPD_ENABLED** is set
2. the OpenMP program defines the function **ompd_enable** (See Section 4.2.6.1 on page 550) and it returns logical true (i.e., non-zero) when the OpenMP runtime calls this function. Immediately before the OpenMP implementation initializes itself it will check to see whether the **ompd_enable** function is defined in the OpenMP program or one of its dynamically-linked libraries. If a definition is found, the OpenMP will call the function it finds to determine what it returns.

In some cases it may not be possible to control an OpenMP program's environment. **ompd_enable** allows an OpenMP program itself to turn on data collection for OMPD. The function can be positioned in an otherwise empty DLL the programmer can link with the OpenMP program. This leaves the OpenMP program code unmodified. A tool implementor may choose to distribute a DLL defining **ompd_enable** as a convenience to programmers.

Cross References

- **OMPD_ENABLED**, Section 5.20 on page 580
- **ompd_enable**, Section 4.2.6.1 on page 550
- Activating an OMPT Tool, Section 4.1.1 on page 378

4.2.1.2 Finding the OMPD plugin

An OpenMP runtime may have more than one matching OMPD plugin for tools to use. The tool must be able to locate the right plugin to use for the OpenMP program it is examining.

As part of the OpenMP interface, OMPD requires that the OpenMP runtime system provides a public variable **ompd_dll_locations**, which is an **argv**-style vector of filename string pointers that provides the pathnames(s) of any compatible OMPD plugin implementations. **ompd_dll_locations** must have **C** linkage. The tool uses the name of the variable verbatim, and in particular, will not apply any name mangling before performing the look up. The pathnames may be relative to the directory containing the OpenMP runtime or absolute.

ompd_dll_locations points to a NULL-terminated vector of zero or more NULL-terminated pathname strings. There are no filename conventions for pathname strings. The last entry in the vector is NULL. The vector of string pointers must be fully initialized *before*

ompd_dll_locations is set to a non-NULL value, such that if the tool, such as a debugger, stops execution of the OpenMP program at any point where **ompd_dll_locations** is non-NULL, then the vector of strings it points to is valid and complete.

The programming model or architecture of the tool (and hence that of the required OMPD) does not have to match that of the OpenMP program being examined. It is the responsibility of the tool to interpret the contents of **ompd_dll_locations** to find a suitable OMPD that matches its own architectural characteristics. On platforms that support different programming models (*e.g.*, 32-bit vs 64-bit), OpenMP implementers are encouraged to provide OMPD library for all models, and which can handle OpenMP programs of any model. Thus, for example, a 32-bit debugger using OMPD should be able to debug a 64-bit OpenMP program by loading a 32-bit OMPD that can manage a 64-bit OpenMP runtime.

Cross References

- Identifying the Matching OMPD, Section 4.3.2.2 on page 560

4.2.2 OMPD Data Types

In this section, we define the types, structures, and functions for the OMPD API.

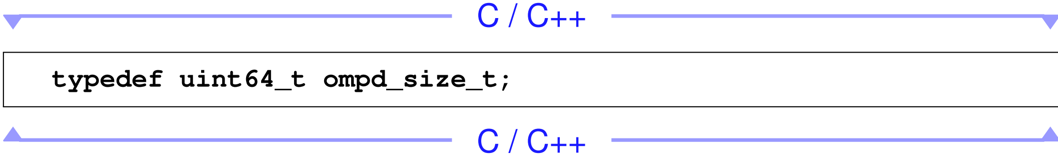
4.2.2.1 Basic Types

The following describes the basic OMPD API types.

4.2.2.1.1 Size Type

This type is used to specify the number of bytes in opaque data objects passed across the OMPD API.

Format



C / C++

```
typedef uint64_t ompd_size_t;
```

C / C++

4.2.2.1.2 Wait ID Type

This type identifies what a thread is waiting for.

Format

C / C++

```
typedef uint64_t ompd_wait_id_t;
```

C / C++

4.2.2.1.3 Basic Value Types

These definitions represent a word, address, and segment value types.

Format

C / C++

```
typedef uint64_t ompd_addr_t;
typedef int64_t  ompd_word_t;
typedef uint64_t ompd_seg_t;
```

C / C++

The *ompd_addr_t* type represents an unsigned integer address in an OpenMP process. The *ompd_word_t* type represents a signed version of *ompd_addr_t* to hold a signed integer of the OpenMP process. The *ompd_seg_t* type represents an unsigned integer segment value.

4.2.2.1.4 Address Type

This type is a structure that OMPD uses to specify device addresses, which may or may not be segmented.

Format

C / C++

```
typedef struct {
  ompd_seg_t segment;
  ompd_addr_t address;
} ompd_address_t;

#define OMPD_SEGMENT_UNSPECIFIED 0
```

C / C++

For non segmented architectures, use OMPD_SEGMENT_UNSPECIFIED in the *segment* field of **ompd_address_t**.

4.2.2.2 System Device Identifiers

Different OpenMP runtimes may utilize different underlying devices. The type used to hold an device identifier can vary in size and format, and therefore is not explicitly represented in the OMPD API. Device identifiers are passed across the interface using a device-identifier 'kind', a pointer to where the device identifier is stored, and the size of the device identifier in bytes. The OMPD library and tool using it must agree on the format of what is being passed. Each different kind of device identifier uses a unique unsigned 64-bit integer value.

Recommended values of **omp_device_kind_t** are defined in the **ompd_types.h** header file, which is available on http://www.openmp.org/.

Format

C / C++

```
typedef uint64_t omp_device_kind_t;
```

C / C++

4.2.2.3 Thread Identifiers

Different OpenMP runtimes may use different underlying native thread implementations. The type used to hold a thread identifier can vary in size and format, and therefore is not explicitly represented in the OMPD API. Thread identifiers are passed across the interface using a

thread-identifier 'kind', a pointer to where the thread identifier is stored, and the size of the thread identifier in bytes. The OMPD library and tool using it must agree on the format of what is being passed. Each different kind of thread identifier uses a unique unsigned 64-bit integer value.

Recommended values of **ompd_thread_id_kind_t** are defined in the **ompd_types.h** header file, which is available on http://www.openmp.org/.

Format

C / C++

```
typedef uint64_t ompd_thread_id_kind_t;
```

C / C++

4.2.2.4 OMPD Handle Types

Each operation of the OMPD interface that applies to a particular address space, thread, parallel region, or task must explicitly specify a *handle* for the operation. OMPD employs handles for address spaces (for a host or target device), threads, parallel regions, and tasks. A handle for an entity is constant while the entity itself is alive. Handles are defined by the OMPD plugin, and are opaque to the tool. The following defines the OMPD handle types:

Format

C / C++

```
typedef struct _ompd_aspace_handle_s ompd_address_space_handle_t;
typedef struct _ompd_thread_handle_s ompd_thread_handle_t;
typedef struct _ompd_parallel_handle_s ompd_parallel_handle_t;
typedef struct _ompd_task_handle_s ompd_task_handle_t;
```

C / C++

Defining externally visible type names in this way introduces type safety to the interface, and helps to catch instances where incorrect handles are passed by the tool to the OMPD library. The **struct**s do not need to be defined at all. The OMPD library must cast incoming (pointers to) handles to the appropriate internal, private types.

4.2.2.5 Tool Context Types

A third-party tool uses contexts to uniquely identifies abstractions. These contexts are opaque to the OMPD library and are defined as follows:

1 **Format**

C / C++

```
typedef struct _ompd_aspace_cont_s ompd_address_space_context_t;
typedef struct _ompd_thread_cont_s ompd_thread_context_t;
```

C / C++

2 #### 4.2.2.6 Return Code Types

3 Each OMPD operation has a return code. The return code types and their semantics are defined as
4 follows:

5 **Format**

C / C++

```
typedef enum {
  ompd_rc_ok = 0,
  ompd_rc_unavailable = 1,
  ompd_rc_stale_handle = 2,
  ompd_rc_bad_input = 3,
  ompd_rc_error = 4,
  ompd_rc_unsupported = 5,
  ompd_rc_needs_state_tracking = 6,
  ompd_rc_incompatible = 7,
  ompd_rc_device_read_error = 8,
  ompd_rc_device_write_error = 9,
  ompd_rc_nomem = 10,
} ompd_rc_t;
```

C / C++

6 **Description**

7 **ompd_rc_ok** is returned when the operation is successful.

8 **ompd_rc_unavailable** is returned when information is not available for the specified context.

9 **ompd_rc_stale_handle** is returned when the specified handle is no longer valid.

10 **ompd_rc_bad_input** is returned when the input parameters (other than handle) are invalid.

ompd_rc_error is returned when a fatal error occurred.

ompd_rc_unsupported is returned when the requested operation is not supported.

ompd_rc_needs_state_tracking is returned when the state tracking operation failed because state tracking is not currently enabled.

ompd_rc_incompatible is returned when this OMPD is incompatible with the OpenMP program.

ompd_rc_device_read_error is returned when a read operation failed on the device

ompd_rc_device_write_error is returned when a write operation failed to the device.

ompd_rc_nomem is returned when unable to allocate memory.

4.2.2.7 OpenMP Scheduling Types

The following enumeration defines **ompd_sched_t**, which is the OMPD API definition corresponding to the OpenMP enumeration type **omp_sched_t** (see

Section 3.2.12 on page 311). **ompd_sched_t** also defines **ompd_sched_vendor_lo** and **ompd_sched_vendor_hi** to define the range of implementation-specific **omp_sched_t** values than can be handle by the OMPD API.

Format

C / C++

```
typedef enum {
  ompd_sched_static = 1,
  ompd_sched_dynamic = 2,
  ompd_sched_guided = 3,
  ompd_sched_auto = 4,
  ompd_sched_vendor_lo = 5,
  ompd_sched_vendor_hi = 0x7fffffff
} ompd_sched_t;
```

C / C++

4.2.2.8 OpenMP Proc Binding Types

The following enumeration defines **ompd_proc_bind_t**, which is the OMPD API definition corresponding to the OpenMP enumeration type **omp_proc_bind_t** (Section 3.2.22 on page 323).

Format

C / C++

```
typedef enum {
  ompd_proc_bind_false = 0,
  ompd_proc_bind_true = 1,
  ompd_proc_bind_master = 2,
  ompd_proc_bind_close = 3,
  ompd_proc_bind_spread = 4
} ompd_proc_bind_t;
```

C / C++

4.2.2.9 Primitive Types

The following structure contains members that the OMPD library can use to interrogate the tool about the "sizeof" of primitive types in the OpenMP architecture address space.

Format

C / C++

```
typedef struct {
  uint8_t sizeof_char;
  uint8_t sizeof_short;
  uint8_t sizeof_int;
  uint8_t sizeof_long;
  uint8_t sizeof_long_long;
  uint8_t sizeof_pointer;
} ompd_device_type_sizes_t;
```

C / C++

Description

The fields of **ompd_device_type_sizes_t** give the sizes of the eponymous basic types used by the OpenMP runtime. As the tool and the OMPD plugin, by definition, have the same architecture and programming model, the size of the fields can be given as **int**.

Cross References

- **ompd_callback_sizeof_fn_t**, Section 4.2.3.3.1 on page 493

4.2.2.10 Runtime State Types

The OMPD runtime states mirror those in OMPT (Section 4.1.5.1.12 on page 454). Note that there is no guarantee that the numeric values of the corresponding members of the enumerations are identical.

Format

C / C++

```
typedef enum {
  ompd_state_work_serial = 0x00,
  ompd_state_work_parallel = 0x01,
  ompd_state_work_reduction = 0x02,
  ompd_state_idle = 0x10,
  ompd_state_overhead = 0x20,
  ompd_state_wait_barrier = 0x40,
  ompd_state_wait_barrier_implicit = 0x41,
  ompd_state_wait_barrier_explicit = 0x42,
  ompd_state_wait_taskwait = 0x50,
  ompd_state_wait_taskgroup = 0x51,
  ompd_state_wait_mutex = 0x60,
  ompd_state_wait_lock = 0x61,
  ompd_state_wait_critical = 0x62,
  ompd_state_wait_atomic = 0x63,
  ompd_state_wait_ordered = 0x64,
  ompd_state_undefined = 0x70,
  ompd_state_first = 0x71,
} ompd_state_t;
```

C / C++

Description

ompd_state_work_serial - working outside parallel

ompd_state_work_parallel - working within parallel

ompd_state_work_reduction - performing a reduction

ompd_state_idle - waiting for work

ompd_state_overhead - non-wait overhead

ompd_state_wait_barrier - generic barrier

ompd_state_wait_barrier_implicit - implicit barrier

ompd_state_wait_barrier_explicit - explicit barrier

ompd_state_wait_taskwait - waiting at a taskwait

ompd_state_wait_taskgroup - waiting at a taskgroup

ompd_state_wait_mutex - waiting for any mutex kind

ompd_state_wait_lock - waiting for lock

ompd_state_wait_critical - waiting for critical

ompd_state_wait_atomic - waiting for atomic

ompd_state_wait_ordered - waiting for ordered

ompd_state_undefined - undefined thread state

ompd_state_first - initial enumeration state

4.2.3 OMPD Tool Callback Interface

For the OMPD plugin to provide information about the internal state of the OpenMP runtime system in an OpenMP process or core file, it must have a means to extract information from the OpenMP process that the tool is debugging. The OpenMP process that the tool is operating on may be either a “live” process or a core file, and a thread may be either a “live” thread in an OpenMP process, or a thread in a core file. To enable OMPD to extract state information from an OpenMP process or core file, the tool must supply the OMPD with callback functions to inquire about the size of primitive types in the device of the OpenMP process, look up the addresses of symbols, as well as read and write memory in the device. OMPD then uses these callbacks to implement its interface operations. Signatures for the tool callbacks used by OMPD are given below.

4.2.3.1 Memory Management of OMPD Plugin

The OMPD library must not access the heap manager directly. Instead, if it needs heap memory it must use the memory allocation and deallocation callback functions that are described in this section, **ompd_callback_memory_alloc_fn_t** and **ompd_callback_memory_free_fn_t**, which are provided by the tool to obtain and release heap memory. This will ensure that the library does not interfere with any custom memory management scheme that the tool may use.

If the OMPD library is implemented in **C++**, memory management operators like **new** and **delete** in all their variants, *must all* be overloaded and implemented in terms of the callbacks provided by the tool. The OMPD library must be coded so that any of its definitions of **new** or **delete** do not interfere with any defined by the tool.

In some cases, the OMPD library will need to allocate memory to return results to the tool. This memory will then be 'owned' by the tool, which will be responsible for releasing it. It is therefore vital that the OMPD library and the tool use the same memory manager.

OMPD handles are created by the OMPD library. These are opaque to the tool, and depending on the specific implementation of OMPD may have complex internal structure. The tool cannot know whether the handle pointers returned by the API correspond to discrete heap allocations. Consequently, the tool must not simply deallocate a handle by passing an address it receives from the OMPD library to its own memory manager. Instead, the API includes functions that the tool must use when it no longer needs a handle.

Contexts are created by a tool and passed to the OMPD library. The OMPD library does not need to release contexts; instead this will be done by the tool after it releases any handles that may be referencing the contexts.

4.2.3.1.1 ompd_callback_memory_alloc_fn_t

Summary

The type signature of the callback routine provided by the tool to be used by the OMPD plugin to allocate memory.

C

```
typedef ompd_rc_t (*ompd_callback_memory_alloc_fn_t) (
  ompd_size_t    nbytes,
  void          **ptr
);
```

C

Description

The OMPD plugin may call the **ompd_callback_memory_alloc_fn_t** callback function to allocate memory.

Description of Arguments

The argument *nbytes* gives the size in bytes of the block of memory the OMPD wants allocated.

The address of the newly allocated block of memory is returned in **ptr*. The newly allocated block is suitably aligned for any type of variable, and is not guaranteed to be zeroed.

Cross References

- **ompd_size_t**, Section 4.2.2.1.1 on page 479
- **ompd_rc_t**, Section 4.2.2.6 on page 483

4.2.3.1.2 ompd_callback_memory_free_fn_t

Summary

The type signature of the callback routine provided by the tool to be used by the OMPD plugin to deallocate memory.

C

```
typedef ompd_rc_t (*ompd_callback_memory_free_fn_t) (
  void     *ptr
);
```

C

Description

The OMPD plugin calls the **ompd_callback_memory_free_fn_t** callback function to deallocate memory obtained from a prior call to the **ompd_callback_memory_alloc_fn_t** callback function.

Description of Arguments

ptr is the address of the block to be deallocated.

Cross References

- **ompd_callback_memory_alloc_fn_t**, Section 4.2.3.1.1 on page 488
- **ompd_rc_t**, Section 4.2.2.6 on page 483
- **ompd_callbacks_t**, Section 4.2.3.8 on page 499

4.2.3.2 Context Management and Navigation

The tool provides the OMPD plugin with callbacks to manage and navigate context relationships.

4.2.3.2.1 ompd_callback_get_thread_context_for_thread_id_fn_t

Summary

The type signature of the callback routine provided by the third party tool the OMPD library can use to map a thread identifier to a tool *thread context*.

C

```
typedef ompd_rc_t
  (*ompd_callback_get_thread_context_for_thread_id_fn_t) (
    ompd_address_space_context_t* address_space_context,
    ompd_thread_id_kind_t kind,
    ompd_size_t sizeof_thread_id,
    const void* thread_id,
    ompd_thread_context_t** thread_context
);
```

C

Description

This callback maps a thread identifier within the address space identified by *address_space_context* to a tool thread context. The OMPD plugin library can use the thread context, for example, to access thread local storage (TLS).

Description of Arguments

The input argument *address_space_context* is an opaque handle provided by the tool to reference an address space. The input arguments *kind*, *sizeof_thread_id*, and *thread_id* represent a thread identifier. On return the output argument *thread_context* provides an opaque handle to the OMPD plugin library that maps a thread identifier to a tool thread context.

4.2.3.2.2 ompd_callback_get_address_space_context_for_thread_fn_t

Summary

The type signature of the callback routine provided by the tool the OMPD plugin can use to find the address space context for a thread identified by a thread context.

C

```
typedef ompd_rc_t
  (*ompd_callback_get_address_space_context_for_thread_fn_t) (
    ompd_thread_context_t          *thread_context,
    ompd_address_space_context_t  **address_space_context
);
```

C

Description

This callback maps a tool thread context to its address space context.

Description of Arguments

This callback finds the tool address space context for the thread identified by the tool thread context *thread_context*. The callback returns the address space context in **address_space_context*. If *thread_context* refers to a host device thread, this function returns the context for the host address space. If *thread_context* refers to a device thread, this function returns the device's address space.

Cross References

- **ompd_address_space_context_t**, Section 4.2.2.5 on page 482
- **ompd_thread_context_t**, Section 4.2.2.5 on page 482
- **ompd_rc_t**, Section 4.2.2.6 on page 483
- **ompd_callbacks_t**, Section 4.2.3.8 on page 499

4.2.3.2.3 ompd_callback_get_process_context_for_address_space_context_fn_t

Summary

The type signature of the callback routine provided by the third party tool the OMPD library can use to map a tool address space context to a tool address space context of the OpenMP process.

C

```
typedef ompd_rc_t
  (*ompd_callback_get_process_context_for_context_fn_t) (
    ompd_address_space_context_t*   target_device_address_space_context,
    ompd_address_space_context_t**  host_device_address_space_context,
);
```

C

Description

This callback maps a tool address space context for a target device to the address space context for the host device.

Description of Arguments

This callback maps the tool address space context for a target device provided in *target_device_address_space_context* to the address space context of its containing host device which it returns in **host_device_address_space_context*.

Cross References

- **ompd_address_space_context_t**, Section 4.2.2.5 on page 482
- **ompd_rc_t**, Section 4.2.2.6 on page 483
- **ompd_callbacks_t**, Section 4.2.3.8 on page 499

4.2.3.3 Sizes of Primitive Types

The tool provides the OMPD plugin with callbacks to discover information about the sizes of the basic primitive types in an address space. This information is important to an OMPD plugin because the architecture or programming model of the OpenMP runtime it is examining may be different from the its own and that of the tool which loaded it. On platforms with multiple programming models, an OMPD plugin may need to be able to handle OpenMP runtimes using different sizes for the basic types.

1 **Cross References**

2 • OMPD — Overivew, Section 4.2 on page 476

3 ### 4.2.3.3.1 ompd_callback_sizeof_fn_t

4 **Summary**

5 The type signature of the callback routine provided by the tool the OMPD plugin can use to find the
6 sizes of the primitive types in an address space.

C

```
typedef ompd_rc_t (*ompd_callback_sizeof_fn_t) (
    ompd_address_space_context_t  *address_space_context,
    ompd_device_type_sizes_t      *sizes
);
```

C

7 **Description**

8 This callback may be called by the OMPD library to obtain the sizes of the basic primitive types for
9 a given address space.

10 **Description of Arguments**

11 The callback returns the sizes of the basic primitive types used by the *address_space_context* in
12 **sizes*.

13 **Cross References**

14 • **ompd_address_space_context_t**, Section 4.2.2.5 on page 482

15 • **ompd_device_type_sizes_t**, Section 4.2.2.9 on page 485

16 • **ompd_rc_t**, Section 4.2.2.6 on page 483

17 • **ompd_callbacks_t**, Section 4.2.3.8 on page 499

4.2.3.4 ompd_callback_symbol_addr_fn_t

Summary

The type signature of the callback provided by the tool the OMPD plugin can use to look up the addresses of symbols in an OpenMP program.

C

```
typedef ompd_rc_t (*ompd_callback_symbol_addr_fn_t) (
    ompd_address_space_context_t*   address_space_context,
    ompd_thread_context_t*          thread_context,
    const char*                     symbol_name,
    ompd_address_t*                 symbol_addr
);
```

C

Description

This callback function may be called by the OMPD library to look up addresses of symbols within an specified address space of the tool.

Description of Arguments

This callback looks up the symbol provided in *symbol_name*. The *thread_context* is an optional input parameter which should be NULL for global memory access.

If *thread_context* is not NULL, it gives the thread specific context for the symbol lookup, for the purpose of calculating thread local storage (TLS) addresses.

The *symbol_name* supplied by the OMPD plugin is used verbatim by the tool, and in particular, no name mangling, demangling or other transformations are performed prior to the lookup.

The callback returns the address of the symbol in **symbol_addr*.

Cross References

- **ompd_address_space_context_t**, Section 4.2.2.5 on page 482
- **ompd_thread_context_t**, Section 4.2.2.5 on page 482
- **ompd_address_t**, Section 4.2.2.1.4 on page 480
- **ompd_rc_t**, Section 4.2.2.6 on page 483
- **ompd_callbacks_t**, Section 4.2.3.8 on page 499

4.2.3.5 Accessing Memory in the OpenMP Program or Runtime

The OMPD plugin may need to read from, or write to, the OpenMP program. It cannot do this directly, but instead must use the callbacks provided to it by the tool, which will perform the operation on its behalf.

4.2.3.5.1 ompd_callback_memory_read_fn_t

Summary

The type signature of the callback provided by the tool the OMPD plugin can use to read data out of an OpenMP program.

C

```
typedef ompd_rc_t (*ompd_callback_memory_read_fn_t) (
    ompd_address_space_context_t  *address_space_context,
    ompd_thread_context_t         *thread_context,
    const ompd_address_t          *addr,
    ompd_size_t                    nbytes,
    void                          *buffer
);
```

C

Description

The OMPD plugin may call this function callback to have the tool read a block of data from a location within an address space into a provided buffer.

Description of Arguments

The address from which the data are to be read out of the OpenMP program specified by *address_space_context* is given by *addr*. *nbytes* gives the number of bytes to be transfered. The *thread_context* argument is optional for global memory access, and in this case should be NULL. If it is not NULL, *thread_context* identifies the thread specific context for the memory access for the purpose of accessing thread local storage (TLS).

The data are returned through *buffer*, which is allocated and owned by the OMPD plugin. The contents of the buffer are unstructured, raw bytes. It is the responsibility of the OMPD plugin to arrange for any transformations such as byte-swapping that may be necessary (see Section 4.2.3.6.1 on page 498) to interpret the data returned.

Cross References

- `ompd_address_space_context_t`, Section 4.2.2.5 on page 482
- `ompd_thread_context_t`, Section 4.2.2.5 on page 482
- `ompd_address_t`, Section 4.2.2.1.4 on page 480
- `ompd_word_t`, Section 4.2.2.1.3 on page 480
- `ompd_rc_t`, Section 4.2.2.6 on page 483
- `ompd_callback_device_host_fn_t`, Section 4.2.3.6.1 on page 498
- `ompd_callbacks_t`, Section 4.2.3.8 on page 499

4.2.3.5.2 ompd_callback_memory_write_fn_t

Summary

The type signature of the callback provided by the tool the OMPD plugin can use to write data to an OpenMP program.

C

```
typedef ompd_rc_t (*ompd_callback_memory_write_fn_t) (
    ompd_address_space_context_t   *address_space_context,
    ompd_thread_context_t          *thread_context,
    const ompd_address_t           *addr,
    ompd_size_t                     nbytes,
    const void                     *buffer
);
```

C

Description

The OMPD plugin may call this function callback to have the tool write a block of data to a location within an address space from a provided buffer.

Description of Arguments

The address to which the data are to be written in the OpenMP program specified by *address_space_context* is given by *addr*. *nbytes* gives the number of bytes to be transfered. The *thread_context* argument is optional for global memory access, and in this case should be NULL. If it is not NULL, *thread_context* identifies the thread specific context for the memory access for the purpose of accessing thread local storage (TLS).

The data to be written are passed through *buffer*, which is allocated and owned by the OMPD plugin. The contents of the buffer are unstructured, raw bytes. It is the responsibility of the OMPD plugin to arrange for any transformations such as byte-swapping that may be necessary (see Section 4.2.3.6.1 on page 498) to render the data into a form compatible with the OpenMP runtime.

Cross References

- **ompd_address_space_context_t** Section 4.2.2.5 on page 482
- **ompd_thread_context_t**, Section 4.2.2.5 on page 482
- **ompd_address_t**, Section 4.2.2.1.4 on page 480
- **ompd_word_t**, Section 4.2.2.1.3 on page 480
- **ompd_rc_t**, Section 4.2.2.6 on page 483
- **ompd_callback_device_host_fn_t**, Section 4.2.3.6.1 on page 498
- **ompd_callbacks_t**, Section 4.2.3.8 on page 499

4.2.3.6 Data Format Conversion

The architecuture and/or programming-model of tool and OMPD plugin may be different from that of the OpenMP program being examined. Consequently, the conventions for representing data will differ. The callback interface includes operations for converting between the conventions, such as byte order (‘endianness’), used by the tool and OMPD plugin on the one hand, and the OpenMP program on the other.

4.2.3.6.1 ompd_callback_device_host_fn_t

Summary

The type signature of the callback provided by the tool the OMPD plugin can use to convert data between the formats used by the tool and OMPD plugin, and the OpenMP program.

C

```
typedef ompd_rc_t (*ompd_callback_device_host_fn_t) (
    ompd_address_space_context_t   *address_space_context,
    const void                      *input,
    ompd_size_t                      unit_size,
    ompd_size_t                      count,
    void                            *output
);
```

C

Description

This callback function may be called by the OMPD plugin to convert data between formats used by the tool and OMPD plugin, and the OpenMP program.

Description of Arguments

The OpenMP address space associated with the data is given by *address_space_context*. The source and destination buffers are given by *input* and *output*, respectively. *unit_size* gives the size of each of the elements to be converted. *count* is the number of elements to be transformed.

The input and output buffers are allocated and owned by the OMPD plugin, and it is its responsibility to ensure that the buffers are the correct size, and eventually deallocated when they are no longer needed.

Cross References

- **ompd_address_space_context_t**, Section 4.2.2.5 on page 482
- **ompd_rc_t**, Section 4.2.2.6 on page 483
- **ompd_callbacks_t**, Section 4.2.3.8 on page 499

4.2.3.7 ompd_callback_print_string_fn_t

Summary

The type signature of the callback provided by the tool the OMPD plugin can use to emit output.

C

```c
typedef ompd_rc_t (*ompd_callback_print_string_fn_t) (
    const char          *string
);
```

C

Description

The OMPD plugin emits output, such as logging or debug information, using a callback supplied to it by the tool. It should not emit output directly.

Description of Arguments

string is the null-terminated string to be printed. No conversion or formating is performed on the string.

Cross References

- **ompd_rc_t**, Section 4.2.2.6 on page 483
- **ompd_callbacks_t**, Section 4.2.3.8 on page 499

4.2.3.8 The Callback Interface

Summary

All the OMPD plugin's interactions with the OpenMP program must be through a set of callbacks provided to it by the tool which loaded it. These callbacks must also be used for allocating or releasing resources, such as memory, that the plugin needs.

C

```
typedef struct {
  ompd_callback_memory_alloc_fn_t alloc_memory;
  ompd_callback_memory_free_fn_t  free_memory;
  ompd_callback_print_string_fn_t print_string;
  ompd_callback_sizeof_fn_t       sizeof_type;
  ompd_callback_symbol_addr_fn_t  symbol_addr_lookup;
  ompd_callback_memory_read_fn_t  read_memory;
  ompd_callback_memory_write_fn_t write_memory;
  ompd_callback_device_host_fn_t  device_to_host;
  ompd_callback_device_host_fn_t  host_to_device;
  ompd_callback_get_thread_context_for_thread_id_fn_t
                        get_thread_context_for_thread_id;
  ompd_callback_get_address_space_context_for_thread_fn_t
                        get_address_space_context_for_thread;
  ompd_callback_get_process_context_for_address_space_context_fn_t
                        get_containing_process_context;
} ompd_callbacks_t;
```

C

Description

The set of callbacks the OMPD plugin should use is collected in the **ompd_callbacks_t** record structure. An instance of this type is passed to the OMPD plugin as a parameter to **ompd_initialize**. Each field points to a function that the OMPD plugin should use for interacting with the OpenMP program, or getting memory from the tool.

The *alloc_memory* and *free_memory* fields are pointers to functions the OMPD plugin uses to allocate and release dynamic memory.

print_string points to a function that prints a string.

The architectures or programming models of the OMPD plugin and party tool may be different from that of the OpenMP program being examined. *sizeof_type* points to function that allows the OMPD plugin to determine the sizes of the basic integer and pointer types used by the OpenMP program. Because of the differences in architecture or programming model, the conventions for representing data in the OMPD plugin and the OpenMP program may be different. The *device_to_host* field points to a function which translates data from the conventions used by the OpenMP program to that used by the tool and OMPD plugin. The reverse operation is performed by the function pointed to by the *host_to_device* field.

The OMPD plugin may need to access memory in the OpenMP program. The *symbol_addr_lookup* field points to a callback the OMPD plugin can use to find the address of a global or thread local

storage (TLS) symbol. The *read_memory* and *write_memory* fields are pointers to functions for reading from, and writing to, global or TLS memory in the OpenMP program, respectively.

get_thread_context_for_thread_id is a pointer to a function the OMPD plugin can use to obtain a thread context that corresponds to a thread identifier. *get_address_space_context_for_thread* points to a callback the OMPD plugin can use to get the tool context that 'owns' the thread represented by a tool thread context.

Cross References

- **ompd_callback_memory_alloc_fn_t**, Section 4.2.3.1.1 on page 488
- **ompd_callback_memory_free_fn_t**, Section 4.2.3.1.2 on page 489
- **ompd_callback_print_string_fn_t**, Section 4.2.3.7 on page 499
- **ompd_callback_sizeof_fn_t**, Section 4.2.3.3.1 on page 493
- **ompd_callback_symbol_addr_fn_t**, Section 4.2.3.4 on page 494
- **ompd_callback_memory_read_fn_t**, Section 4.2.3.5.1 on page 495
- **ompd_callback_memory_write_fn_t**, Section 4.2.3.5.2 on page 496
- **ompd_callback_device_host_fn_t**, Section 4.2.3.6.1 on page 498
- **ompd_callback_get_thread_context_for_thread_id_fn_t**, Section 4.2.3.2.1 on page 490
- **ompd_callback_get_address_space_context_for_thread_fn_t**, Section 4.2.3.2.2 on page 491
- **ompd_callback_get_process_context_for_address_space_context_fn_t**, Section 4.2.3.2.3 on page 492

4.2.4 OMPD Tool Interface Routines

4.2.4.1 Per OMPD Library Initialization and Finalization

The OMPD library must be initialized exactly once after it is loaded, and finalized exactly once before it is unloaded. Per OpenMP process or core file initialization and finalization are also required. A tool starts the initialization by calling **ompd_initialize**. Once loaded, the tool can determine the version of the OMPD API supported by the plugin by calling **ompd_get_version**.

4.2.4.1.1 `ompd_initialize`

Summary

The **`ompd_initialize`** function initializes the OMPD library.

Format

C

```
ompd_rc_t ompd_initialize(const ompd_callbacks_t *callbacks);
```

C

Description

The above initialization is performed for each OMPD library that is loaded by an OMPD using tool. There may be more than one library present in a thid-party tool, such as a debugger, because the tool may be controlling a number of devices that may be using different runtime systems which require different OMPD libraries. This initialization must be performed exactly once before the tool can begin operating on a OpenMP process or core file.

Description of Arguments

The tool provides the OMPD plugin with a set of callback functions in the *callbacks* input argument which enables the OMPD plugin to allocate and deallocate memory in the tool's address space, to lookup the sizes of basic primitive types in the device, to lookup symbols in the device, as well as to read and write memory in the device.

Cross References

- **`ompd_rc_t`**, Section 4.2.2.6 on page 483
- **`ompd_callbacks_t`**, Section 4.2.3.8 on page 499

4.2.4.1.2 `ompd_get_version`

Summary

The **`ompd_get_version`** function returns the OMPD API version.

Format

C

```
ompd_rc_t ompd_get_version(int *version);
```

C

Description

The tool may call this function to obtain the version number of the OMPD plugin.

Description of Arguments

The version number is returned in to the location pointed to by the *version* output argument

4.2.4.1.3 `ompd_get_version_string`

Summary

The **ompd_get_version_string** function returns a descriptive string for the OMPD API version.

Format

C

```
ompd_rc_t ompd_get_version_string(const char **string);
```

C

Description

The tool may call this function to obtain a pointer to a descriptive version string of the OMPD plugin.

Description of Arguments

A pointer to a descriptive version string will be placed into the **string* output argument. The string returned by the OMPD library is 'owned' by the library, and it must not be modified or released by the tool. It is guaranteed to remain valid for as long as the library is loaded.

ompd_get_version_string may be called before **ompd_initialize**. Accordingly, the OMPD library must not use heap or stack memory for the string it returns to the tool.

The signatures of **ompd_get_version** and **ompd_get_version_string** are guaranteed not to change in future versions of the API. In contrast, the type definitions and prototypes in the rest of the API do not carry the same guarantee. Therefore an OMPD using tool should check the version of the API of a loaded OMPD library before calling any other function of the API.

4.2.4.1.4 ompd_finalize

Summary

When the tool is finished with the OMPD library it should call **ompd_finalize** before unloading the library.

Format

C

```
ompd_rc_t ompd_finalize(void);
```

C

Description

This must be the last call the tool makes to the library before unloading it. The call to **ompd_finalize** gives the OMPD library a chance to free up any remaining resources it may be holding.

The OMPD library may implement a *finalizer* section. This will execute as the library is unloaded, and therefore after the tool's call to **ompd_finalize**. The OMPD library is allowed to use the callbacks (provided to it earlier by the tool after the call to **ompd_initialize**) during finalization.

Cross References

- **ompd_rc_t**, Section 4.2.2.6 on page 483

4.2.4.2 Per OpenMP Process Initialization and Finalization

4.2.4.2.1 `ompd_process_initialize`

Summary

A tool obtains an address space handle when it initializes a session on a live process or core file by calling **ompd_process_initialize**.

Format

C

```
ompd_rc_t ompd_process_initialize(
  ompd_address_space_context_t  *context,
  ompd_address_space_handle_t  **handle
);
```

C

Description

On return from **ompd_process_initialize** the address space handle is owned by the tool. This function must be called before any OMPD operations are performed on the OpenMP process. **ompd_process_initialize** gives the OMPD library an opportunity to confirm that it is capable of handling the OpenMP process or core file identified by the **context**. Incompatibility is signaled by a return value of **ompd_rc_incompatible**. On return, the handle is owned by the tool, which must release it using **ompd_release_address_space_handle**.

Description of Arguments

The input argument *context* is an opaque handle provided by the tool to address an address space. On return the output argument *handle* provides an opaque handle to the tool for this address space, which the tool is responsible for releasing when it is no longer needed

Cross References

- **ompd_address_space_context_t** type, see Section 4.2.2.5 on page 482.
- **ompd_address_space_handle_t** type, see Section 4.2.2.4 on page 482.

4.2.4.2.2 ompd_device_initialize

Summary

A tool obtains an address space handle for a device that has at least one active target region by calling **ompd_device_initialize**.

Format

C

```
ompd_rc_t ompd_device_initialize(
  ompd_address_space_handle_t*   process_handle,
  ompd_address_space_context_t* device_context,
  omp_device_kind_t   kind,
  ompd_size_t   sizeof_id,
  void*   id,
  ompd_address_space_handle_t** device_handle
);
```

C

Description

On return from **ompd_device_initialize** the address space handle is owned by the tool.

Description of Arguments

The input argument *process_handle* is an opaque handle provided by the tool to reference the address space of the OpenMP process. The input argument *device_context* is an opaque handle provided by the tool to reference a device address space. The input arguments *kind*, *sizeof_id*, and *id* represent a device identifier. On return the output argument *device_handle* provides an opaque handle to the tool for this address space.

Cross References

- **ompd_address_space_context_t** type, see Section 4.2.2.5 on page 482.
- **omp_device_kind_t** type, see Section 4.2.2.2 on page 481.
- **ompd_size_t** type, see Section 4.2.2.1.1 on page 479.
- **ompd_address_space_handle_t** type, see Section 4.2.2.4 on page 482.

4.2.4.2.3 `ompd_release_address_space_handle`

Summary

A tool calls **`ompd_release_address_space_handle`** to release an address space handle.

Format

C

```
ompd_rc_t ompd_release_address_space_handle(
  ompd_address_space_handle_t *handle
);
```

C

Description

When the tool is finished with the OpenMP process address space handle it should call **`ompd_release_address_space_handle`** to release the handle and give the OMPD library the opportunity to release any resources it may have related to the address space.

Description of Arguments

The input argument *handle* is an opaque handle for an address space to be released.

Cross References

- **`ompd_address_space_handle_t`** type, see Section 4.2.2.4 on page 482.

4.2.4.3 Thread and Signal Safety

The OMPD library does not need to be reentrant. It is the responsibility of the tool to ensure that only one thread enters the OMPD library at a time. The OMPD implementation must not install signal handlers or otherwise interfere with the tool's signal configuration.

4.2.4.4 Available Processors and Threads

4.2.4.4.1 ompd_get_num_procs

Summary

The **ompd_get_num_procs** routine returns the number of processors available on the entity identified by an address space handle.

Format

C

```
ompd_rc_t ompd_get_num_procs (
  ompd_address_space_handle_t *handle,
  ompd_word_t                 *num_processors
);
```

C

Description

The **ompd_get_num_procs** routine returns the number of processors available on the entity identified by an address space handle, and corresponds to the **omp_get_num_procs** OpenMP runtime library routine.

Description of Arguments

The input argument *handle* is an opaque handle for an address space; the number of processors on the entity identified by the address space handle is returned through the output argument *num_processors*. This operation corresponds to the OpenMP runtime function **omp_get_num_procs** that an OpenMP program can call.

Cross References

- **ompd_address_space_handle_t** type, see Section 4.2.2.4 on page 482.
- **omp_get_num_procs**, see Section 3.2.5 on page 305.

4.2.4.4.2 ompd_get_thread_limit

Summary

The **ompd_get_thread_limit** routine returns the maximum number of OpenMP threads available on the entity identified by an address space handle.

Format

C

```
ompd_rc_t ompd_get_thread_limit (
  ompd_address_space_handle_t *handle,
  ompd_word_t   *num_threads
);
```

C

Description

The **ompd_get_thread_limit** routine returns the maximum number of OpenMP threads available on the entity identified by an address space handle, and corresponds to the **omp_get_thread_limit** OpenMP runtime library routine.

Description of Arguments

The input argument *handle* is an opaque handle for an address space; the maximum number of OpenMP threads on the entity identified by the address space handle is returned through the output argument *num_threads*. This operation corresponds to the OpenMP runtime function **omp_get_thread_limit** that an OpenMP program can call.

Cross References

- **ompd_address_space_handle_t** type, see Section 4.2.2.4 on page 482.
- **omp_get_thread_limit**, see Section 3.2.14 on page 314.

1 #### 4.2.4.5 Thread Handles

2 ##### 4.2.4.5.1 `ompd_get_thread_in_parallel`

3 **Summary**

4 The **`ompd_get_thread_in_parallel`** operation enables a tool to obtain handles for
5 OpenMP threads associated with a parallel region.

6 **Format**

C

```
ompd_rc_t ompd_get_thread_in_parallel(
  ompd_parallel_handle_t *parallel_handle,
  int nth_handle,
  ompd_thread_handle_t **thread_handle
);
```

C

7 **Description**

8 A successful invocation of **`ompd_get_thread_in_parallel`** returns a pointer to a thread
9 handle in **`*thread_handle`**. This call yields meaningful results only if all OpenMP threads in
10 the parallel region are stopped.

11 **Description of Arguments**

12 The input argument *parallel_handle* is an opaque handle for a parallel region and selects the parallel
13 region to operate on. The input argument **`nth_handles`** selects the thread of the team to be
14 returned. On return the output argument *thread_handle* is an opaque handle for the selected thread.

15 **Restrictions**

16 The value of **`nth_handles`** must be a non-negative integer smaller than the team size returned by
17 a call to **`ompd_get_num_threads`**.

18 **Cross References**

19 • **`ompd_parallel_handle_t`** type, see Section 4.2.2.4 on page 482.

20 • **`ompd_thread_handle_t`** type, see Section 4.2.2.4 on page 482.

4.2.4.5.2 `ompd_get_thread_handle`

Summary

Mapping a native thread to an OMPD thread handle.

Format

C

```
ompd_rc_t ompd_get_thread_handle(
ompd_address_space_handle_t    *handle,
ompd_thread_id_kind_t            kind,
ompd_size_t                     sizeof_thread_id,
const void                      *thread_id,
ompd_thread_handle_t           **thread_handle
);
```

C

Description

OMPD provides the function **`ompd_get_thread_handle`** to inquire whether a native thread is an OpenMP thread or not. On success, the thread identifier is an OpenMP thread and **`*thread_handle`** is initialized to a pointer to the thread handle for the OpenMP thread.

Description of Arguments

The input argument *handle* is an opaque handle provided by the tool to reference to an address space. The input arguments *kind*, *sizeof_thread_id*, and *thread_id* represent a thread identifier. On return the output argument *thread_handle* provides an opaque handle to the tool for thread within the provided address space.

The thread identifier **thread_id* is guaranteed to be valid for the duration of the call. If the OMPD library needs to retain the thread identifier it must copy it.

Cross References

- **`ompd_address_space_handle_t`** type, see Section 4.2.2.4 on page 482.
- **`ompd_thread_id_kind_t`** type, see Section 4.2.2.3 on page 482.
- **`ompd_size_t`** type, see Section 4.2.2.1.1 on page 479.
- **`ompd_thread_handle_t`** type, see Section 4.2.2.4 on page 482.

1 ### 4.2.4.5.3 `ompd_release_thread_handle`

2 #### Summary

3 This operation releases a thread handle.

4 #### Format

C

```
ompd_rc_t ompd_release_thread_handle(
  ompd_thread_handle_t *thread_handle
);
```

C

5 #### Description

6 Thread handles are opaque to tools, which therefore cannot release them directly. Instead, when the
7 tool is finished with a thread handle it must pass it to the OMPD
8 **`ompd_release_thread_handle`** routine for disposal.

9 #### Description of Arguments

10 The input argument *thread_handle* is an opaque handle for a thread to be released.

11 #### Cross References

12 • **`ompd_thread_handle_t`** type, see Section 4.2.2.4 on page 482.

13 ### 4.2.4.5.4 `ompd_thread_handle_compare`

14 #### Summary

15 The **`ompd_thread_handle_compare`** operation allows tools to compare two thread handles.

Format

C

```
ompd_rc_t ompd_thread_handle_compare(
  ompd_thread_handle_t *thread_handle_1,
  ompd_thread_handle_t *thread_handle_2,
  int *cmp_value
);
```

C

Description

The internal structure of thread handles is opaque to a tool. While the tool can easily compare pointers to thread handles, it cannot determine whether handles of two different addresses refer to the same underlying thread. This function can be used to compare thread handles.

On success, **ompd_thread_handle_compare** returns in ***cmp_value** a signed integer value that indicates how the underlying threads compare: a value less than, equal to, or greater than 0 indicates that the thread corresponding to **thread_handle_1** is, respectively, less than, equal to, or greater than that corresponding to **thread_handle_2**.

Description of Arguments

The input arguments *thread_handle_1* and *thread_handle_2* are opaque handles for threads. On return the output argument *cmp_value* is set to a signed integer value.

Cross References

- **ompd_thread_handle_t** type, see Section 4.2.2.4 on page 482.

4.2.4.5.5 ompd_get_thread_handle_string_id

Summary

The **ompd_get_thread_handle_string_id** function returns a string that contains a unique printable value that identifies the thread.

Format

C

```
ompd_rc_t ompd_get_thread_handle_string_id(
  ompd_thread_handle_t *thread_handle,
  char **string_id
);
```

C

Description

The string that is returned by the function should be a single sequence of alphanumeric or underscore characters, and NULL terminated.

The OMPD library allocates the string returned in ***string_id** using the allocation routine in the callbacks passed to it during initialization. On return the string is owned by the tool, which is responsible for deallocating it. The contents of the strings returned for thread handles which compare as equal with **ompd_thread_handle_compare** must be the same.

Description of Arguments

The input argument *thread_handle* is an opaque handle for a thread and selects the thread to operate on. On return the output argument *string_id* is set to a string representation of the thread.

Cross References

- **ompd_thread_handle_t** type, see Section 4.2.2.4 on page 482.

4.2.4.5.6 ompd_get_thread_id

Summary

Mapping an OMPD thread handle to a native thread.

1 **Format**

C

```
ompd_rc_t ompd_get_thread_id(
ompd_thread_handle_t *thread_handle,
ompd_thread_id_kind_t kind,
ompd_size_t sizeof_thread_id,
void  *thread_id
);
```

C

2 **Description**

3 **ompd_get_thread_id** performs the mapping between an OMPD thread handle and a thread
4 identifier.

5 **Description of Arguments**

6 The input argument *thread_handle* is an opaque thread handle. The input argument *kind* represents
7 the thread identifier. The input argument *sizeof_thread_id* represents the size of the thread
8 identifier. The output argument *thread_id* is a buffer that represents a thread identifier.

9 **Cross References**

10 • **ompd_thread_handle_t** type, see Section 4.2.2.4 on page 482.

11 • **ompd_thread_id_kind_t** type, see Section 4.2.2.3 on page 482.

12 • **ompd_size_t** type, see Section 4.2.2.1.1 on page 479.

13 ## 4.2.4.6 Parallel Region Handles

14 ### 4.2.4.6.1 ompd_get_current_parallel_handle

15 **Summary**

16 The **ompd_get_current_parallel_handle** operation enables the tool to obtain a pointer
17 to the parallel handle for the current parallel region associated with an OpenMP thread.

Format

C

```c
ompd_rc_t ompd_get_current_parallel_handle(
  ompd_thread_handle_t *thread_handle,
  ompd_parallel_handle_t **parallel_handle
);
```

C

Description

This call is meaningful only if the thread whose handle is provided is stopped. The parallel handle must be released by calling **ompd_release_parallel_handle**.

Description of Arguments

The input argument *thread_handle* is an opaque handle for a thread and selects the thread to operate on. On return the output argument *parallel_handle* is set to a handle for the parallel region currently executing on this thread if there is any.

Cross References

- **ompd_thread_handle_t** type, see Section 4.2.2.4 on page 482.
- **ompd_parallel_handle_t** type, see Section 4.2.2.4 on page 482.

4.2.4.6.2 ompd_get_enclosing_parallel_handle

Summary

The **ompd_get_enclosing_parallel_handle** operation enables a tool to obtain a pointer to the parallel handle for the parallel region enclosing the parallel region specified by **parallel_handle**.

1 **Format**

C

```
ompd_rc_t ompd_get_enclosing_parallel_handle(
  ompd_parallel_handle_t *parallel_handle,
  ompd_parallel_handle_t **enclosing_parallel_handle
);
```

C

2 **Description**

3 This call is meaningful only if at least one thread in the parallel region is stopped. A pointer to the
4 parallel handle for the enclosing region is returned in **`*enclosing_parallel_handle`**. After
5 the call the handle is owned by the tool, which must release it when it is no longer required by
6 calling **`ompd_release_parallel_handle`**.

7 **Description of Arguments**

8 The input argument *parallel_handle* is an opaque handle for a parallel region and selects the
9 parallel region to operate on. On return the output argument *parallel_handle* is set to a handle for
10 the parallel region enclosing the selected parallel region.

11 **Cross References**

12 • **`ompd_parallel_handle_t`** type, see Section 4.2.2.4 on page 482.

13 **4.2.4.6.3 `ompd_get_task_parallel_handle`**

14 **Summary**

15 The **`ompd_get_task_parallel_handle`** operation enables a tool to obtain a pointer to the
16 parallel handle for the parallel region enclosing the task region specified by **`task_handle`**.

Format

C

```
ompd_rc_t ompd_get_task_parallel_handle(
  ompd_task_handle_t *task_handle,
  ompd_parallel_handle_t **task_parallel_handle
);
```

C

Description

This call is meaningful only if at least one thread in the parallel region is stopped. A pointer to the parallel regions handle is returned in **`*task_parallel_handle`**. The parallel handle is owned by the tool, which must release it by calling **`ompd_release_parallel_handle`**.

Description of Arguments

The input argument *task_handle* is an opaque handle for a task and selects the task to operate on. On return the output argument *parallel_handle* is set to a handle for the parallel region enclosing the selected task.

Cross References

- **`ompd_task_handle_t`** type, see Section 4.2.2.4 on page 482.
- **`ompd_parallel_handle_t`** type, see Section 4.2.2.4 on page 482.

4.2.4.6.4 `ompd_release_parallel_handle`

Summary

This operation allows releasing a parallel region handle.

Format

C

```
ompd_rc_t ompd_release_parallel_handle(
  ompd_parallel_handle_t *parallel_handle
);
```

C

Description

Parallel region handles are opaque to the tool, which therefore cannot release them directly. Instead, when the tool is finished with a parallel region handle it must must pass it to the OMPD **ompd_release_parallel_handle** routine for disposal.

Description of Arguments

The input argument *parallel_handle* is an opaque handle for a parallel region to be released.

Cross References

- **ompd_parallel_handle_t** type, see Section 4.2.2.4 on page 482.

4.2.4.6.5 ompd_parallel_handle_compare

Summary

The **ompd_parallel_handle_compare** operation allows a tool to compare two parallel region handles.

Format

C

```
ompd_rc_t ompd_parallel_handle_compare(
  ompd_parallel_handle_t *parallel_handle_1,
  ompd_parallel_handle_t *parallel_handle_2,
  int *cmp_value
);
```

C

Description

The internal structure of parallel region handles is opaque to the tool. While the tool can easily compare pointers to parallel region handles, it cannot determine whether handles at two different addresses refer to the same underlying parallel region.

On success, **ompd_parallel_handle_compare** returns in ***cmp_value** a signed integer value that indicates how the underlying parallel regions compare: a value less than, equal to, or

greater than 0 indicates that the region corresponding to **parallel_handle_1** is, respectively, less than, equal to, or greater than that corresponding to **parallel_handle_2**.

For OMPD libraries that always have a single, unique, underlying parallel region handle for a given parallel region, this operation reduces to a simple comparison of the pointers. However, other implementations may take a different approach, and therefore the only reliable way of determining whether two different pointers to parallel regions handles refer the same or distinct parallel regions is to use **ompd_parallel_handle_compare**.

Allowing parallel region handles to be compared allows the tool to hold them in ordered collections. The means by which parallel region handles are ordered is implementation-defined.

Description of Arguments

The input arguments *parallel_handle_1* and *parallel_handle_2* are opaque handles corresponding to parallel regions. On return the output argument *cmp_value* returns a signed integer value that indicates how the underlying parallel regions compare.

Cross References

- **ompd_parallel_handle_t** type, see Section 4.2.2.4 on page 482.

4.2.4.6.6 ompd_get_parallel_handle_string_id

Summary

The **ompd_get_parallel_handle_string_id** function returns a string that contains a unique printable value that identifies the parallel region.

Format

C

```
ompd_rc_t ompd_get_parallel_handle_string_id(
  ompd_parallel_handle_t *parallel_handle,
  char **string_id
);
```

C

Description

The returned string should be a single sequence of alphanumeric or underscore characters, and NULL terminated. The OMPD library allocates the string returned in **`*string_id`** using the allocation routine in the callbacks passed to it during initialization. On return the string is owned by the tool, which is responsible for deallocating it.

The contents of the strings returned for parallel regions handles which compare as equal with **`ompd_parallel_handle_compare`** must be the same.

Description of Arguments

The input argument *parallel_handle* is an opaque handle for a parallel region and selects the parallel region to operate on. On return the output argument *string_id* is set to a string representation of the parallel region.

Cross References

- **`ompd_parallel_handle_t`** type, see Section 4.2.2.4 on page 482.

4.2.4.7 Task Handles

4.2.4.7.1 `ompd_get_current_task_handle`

Summary

A tool uses the **`ompd_get_current_task_handle`** operation to obtain a pointer to the task handle for the current task region associated with an OpenMP thread.

Format

C

```
ompd_rc_t ompd_get_current_task_handle(
  ompd_thread_handle_t *thread_handle,
  ompd_task_handle_t **task_handle
);
```

C

Description

This call is meaningful only if the thread whose handle is provided is stopped. The task handle must be released by calling **`ompd_release_task_handle`**.

Description of Arguments

The input argument *thread_handle* is an opaque handle for a thread and selects the thread to operate on. On return the output argument *task_handle* is set to a handle for the task currently executing on the thread.

Cross References

- **`ompd_thread_handle_t`** type, see Section 4.2.2.4 on page 482.
- **`ompd_task_handle_t`** type, see Section 4.2.2.4 on page 482.

Retrieve the handle for an enclosing task.

The OMPD API includes operations to obtain the handle of the parent of a task represented by a given task handle. There are two notions of parenthood. The *scheduling* parent task is the task that was active when the child task was scheduled to run. The *generating* is the task that encountered the OpenMP that caused the child task to be created.

The generating and scheduling parents need not be the same. This might happen if the thread executing a task encounters an OpenMP construct. When this happens, the thread will enter the runtime. The runtime will set up the tasks to implement the OpenMP program construct, and then call its schedular to choose a task to execute. If the scheduler chooses a task other than one of these newly created tasks to run, the scheduling parent of the selected task will not be the same as its generating parent. The former is the task that the thread was executing most recently, and from which it entered the runtime. The later is the task which encountered the OpenMP construct it is executing.

4.2.4.7.2 `ompd_get_generating_ancestor_task_handle`

Summary

A tool uses **`ompd_get_generating_ancestor_task_handle`** to obtain a pointer to the task handle for the task that encountered the OpenMP construct which caused the task represented by **`task_handle`** to be created.

1 #### Format

C

```
ompd_rc_t ompd_get_generating_ancestor_task_handle(
  ompd_task_handle_t *task_handle,
  ompd_task_handle_t **parent_task_handle
);
```

C

2 #### Description

3 This call is meaningful only if the thread executing the task specified by **task_handle** is stopped.

4 #### Description of Arguments

5 The input argument *task_handle* is an opaque handle for a task and selects the task to operate on.
6 On return the output argument *task_handle* is set to a handle for the task that created the selected
7 task.

8 #### Cross References

9 • **ompd_task_handle_t** type, see Section 4.2.2.4 on page 482.

10 ### 4.2.4.7.3 `ompd_get_scheduling_ancestor_task_handle`

11 #### Summary

12 The **ompd_get_scheduling_ancestor_task_handle** returns the scheduling parent of
13 the task represented **task_handle**.

14 #### Format

C

```
ompd_rc_t ompd_get_scheduling_ancestor_task_handle(
  ompd_task_handle_t *task_handle,
  ompd_task_handle_t **parent_task_handle
);
```

C

1 ### Description

2 In this operation, the scheduling parent task is the OpenMP task that was active when the child task
3 was scheduled. This call is meaningful only if the thread executing the task specified by
4 **task_handle** is stopped. The parent task handle must be released by calling
5 **ompd_release_task_handle**.

6 ### Description of Arguments

7 The input argument *task_handle* is an opaque handle for a task and selects the task to operate on.
8 On return the output argument *parent_task_handle* is set to a handle for the task that is still on the
9 stack of execution on the same thread and was deferred in favor of executing the selected task.

10 ### Cross References

11 • **ompd_task_handle_t** type, see Section 4.2.2.4 on page 482.

12 ## 4.2.4.7.4 ompd_get_implicit_task_in_parallel

13 ### Summary

14 The codeompd_get_implicit_task_in_parallel operation enables a tool to obtain a vector of pointers
15 to task handles for all implicit tasks associated with a parallel region.

16 ### Format

C

```
ompd_rc_t ompd_get_implicit_task_in_parallel(
  ompd_parallel_handle_t *parallel_handle,
  ompd_task_handle_t ***task_handle_vector,
  int *num_handles
);
```

C

Description

This call is meaningful only if all threads associated with the parallel region are stopped. The OMPD library must use the memory allocation callback to obtain the memory for the vector of pointers to task handles returned by the operation. If the OMPD library needs to allocate heap memory for the task handles it returns, it must use the callbacks to acquire this memory. After the call the vector and the task handles are 'owned' by the tool, which is responsible for deallocating them. The task handles must be released calling **ompd_release_task_handle**. The vector was allocated by the OMPD library using the allocation routine passed to it during the call to **ompd_initialize**. The tool itself must deallocate the vector in a compatible manner.

Description of Arguments

The input argument *parallel_handle* is an opaque handle for a parallel region and selects the parallel region to operate on. On return the output argument *task_handle* is set to a handle for the implicit task executing on *nth* thread of the parallel region.

Cross References

- **ompd_parallel_handle_t** type, see Section 4.2.2.4 on page 482.
- **ompd_task_handle_t** type, see Section 4.2.2.4 on page 482.

4.2.4.7.5 ompd_release_task_handle

Summary

This operation releases a task handle.

Format

C

```
ompd_rc_t ompd_release_task_handle(
  ompd_task_handle_t *task_handle
);
```

C

Description

Task handles are opaque to the tool, which therefore cannot release them directly. Instead, when the tool is finished with a task handle it must pass it to the OMPD **ompd_release_task_handle** routine for disposal.

Description of Arguments

The input argument *task_handle* is an opaque handle for a task to be released.

Cross References

- **ompd_task_handle_t** type, see Section 4.2.2.4 on page 482.

4.2.4.7.6 ompd_task_handle_compare

Summary

The **ompd_task_handle_compare** operations allows a tool to compare task handles.

Format

C

```
ompd_rc_t ompd_task_handle_compare(
  ompd_task_handle_t *task_handle_1,
  ompd_task_handle_t *task_handle_2,
  int *cmp_value
);
```

C

Description

The internal structure of task handles is opaque to the tool. While the tool can easily compare pointers to task handles, it cannot determine whether handles at two different addresses refer to the same underlying task.

On success, **ompd_task_handle_compare** returns in ***cmp_value** a signed integer value that indicates how the underlying tasks compare: a value less than, equal to, or greater than 0 indicates that the task corresponding to *task_handle_1* is, respectively, less than, equal to, or greater than that corresponding to *task_handle_2*.

For OMPD libraries that always have a single, unique, underlying task handle for a given task, this operation reduces to a simple comparison of the pointers. However, other implementations may take a different approach, and therefore the only reliable way of determining whether two different pointers to task handles refer the same or distinct task is to use **ompd_task_handle_compare**.

Allowing task handles to be compared allows the tool to hold them in ordered collections. The means by which task handles are ordered is implementation-defined.

Description of Arguments

The input arguments *task_handle_1* and *task_handle_2* are opaque handles corresponding to tasks. On return the output argument *cmp_value* returns a signed integer value that indicates how the underlying tasks compare.

Cross References

- **ompd_task_handle_t** type, see Section 4.2.2.4 on page 482.

4.2.4.7.7 ompd_get_task_handle_string_id

Summary

The **ompd_get_task_handle_string_id** function returns a string that contains a unique printable value that identifies the task.

Format

C

```
ompd_rc_t ompd_get_task_handle_string_id (
  ompd_task_handle_t *task_handle,
  char **string_id
);
```

C

Description

The returned string should be a single sequence of alphanumeric or underscore characters, and NULL terminated. The OMPD library allocates the string returned in ***string_id** using the allocation routine in the callbacks passed to it during initialization. On return the string is owned by the tool, which is responsible for deallocating it.

The contents of the strings returned for task handles which compare as equal with **ompd_task_handle_compare** must be the same.

Description of Arguments

The input argument *thread_handle* is an opaque handle for a task and selects the task to operate on. On return the output argument *string_id* is set to a string representation of the task.

Cross References

- `ompd_task_handle_t` type, see Section 4.2.2.4 on page 482.

4.2.4.7.8 `ompd_get_num_threads`

Summary

Determine the number of threads associated with a parallel region.

Format

C

```
ompd_rc_t ompd_get_num_threads(
  ompd_parallel_handle_t *parallel_handle,
  ompd_word_t *val
);
```

C

Description

Description of Arguments

The input argument *parallel_handle* is an opaque handle for a parallel region and selects the parallel region to operate on. On return the output argument *val* is set to the value that `omp_get_num_threads` would return when executed in this parallel region.

Cross References

- `ompd_parallel_handle_t` type, see Section 4.2.2.4 on page 482.
- `ompd_word_t` type, see Section 4.2.2.1.3 on page 480.

4.2.4.7.9 `ompd_get_parallel_function`

Summary

Returns the entry point of a parallel region.

Format

C

```c
ompd_rc_t ompd_get_parallel_function (
  ompd_parallel_handle_t  *parallel_handle,
  ompd_address_t          *entry_point
);
```

C

Description

Description of Arguments

The input argument *parallel_handle* is an opaque handle for a parallel region, and selects the parallel region to operate on. On return, the output argument *entry_point* is set to the entry point of the code that corresponds to the body of the parallel construct.

Cross References

- `ompd_parallel_handle_t` type, see Section 4.2.2.4 on page 482.
- `ompd_address_t` type, see Section 4.2.2.1.4 on page 480.

4.2.4.7.10 `ompd_get_level`

Summary

Determine the nesting depth of a particular parallel region.

Format

C

```c
ompd_rc_t ompd_get_level (
  ompd_task_handle_t *task_handle,
  ompd_word_t *val
);
```

C

Description

ompd_get_level returns the number of nested parallel regions enclosing the parallel region specified by its handle.

Description of Arguments

The input argument *task_handle* is an opaque handle for a task and selects the task to operate on. On return the output argument *val* is set to the value that **omp_get_level** would return when executed in this task.

Cross References

- **ompd_task_handle_t** type, see Section 4.2.2.4 on page 482.
- **ompd_word_t** type, see Section 4.2.2.1.3 on page 480.

4.2.4.7.11 ompd_get_active_level

Summary

Determine the number of enclosing parallel regions.

Format

C

```c
ompd_rc_t ompd_get_active_level(
  ompd_task_handle_t *task_handle,
  ompd_word_t *val
);
```

C

Description

ompd_get_active_level returns the number of nested, active parallel regions enclosing the parallel region specified by its handle.

Description of Arguments

The input argument *task_handle* is an opaque handle for a task and selects the task to operate on. On return the output argument *val* is set to the value that **omp_get_active_level** would return when executed in this task.

Cross References

- **ompd_task_handle_t** type, see Section 4.2.2.4 on page 482.
- **ompd_word_t** type, see Section 4.2.2.1.3 on page 480.

4.2.4.7.12 ompd_get_task_function

Summary

Task Function Entry Point

Format

C

```
ompd_rc_t ompd_get_task_function (
  ompd_task_handle_t *task_handle,
  ompd_address_t *entry_point
);
```

C

Description

The **ompd_get_task_function** returns the entry point of the code that corresponds to the body of code executed by the task:

Description of Arguments

The input argument *task_handle* is an opaque handle for a task and selects the task to operate on. On return the output argument *entry_point* is set an address that describes the begin of application code which executes the task region.

Cross References

- **ompd_task_handle_t** type, see Section 4.2.2.4 on page 482.
- **ompd_address_t** type, see Section 4.2.2.1.4 on page 480.

4.2.4.7.13 ompd_get_max_threads

Summary

The **ompd_get_max_threads** function returns the first value of the *nthreads-var* ICV.

Format

C

```
ompd_rc_t ompd_get_max_threads (
  const ompd_task_handle_t  *task_handle,
  ompd_word_t               *val
);
```

C

Description

The **ompd_get_max_threads** function returns the first value of the *nthreads-var* ICV, and corresponds to the **omp_get_max_threads** function in the OpenMP runtime API. This returns an upper bound on the number threads that could be used to form a new team if a **parallel** construct without a **num_threads** clause were encountered.

Description of Arguments

The input argument *task_handle* is an opaque handle for a task and selects the task to operate on. On return the output argument *val* is set to the value that **omp_get_max_threads** would return when executed in this task.

Cross References

- **ompd_task_handle_t** type, see Section 4.2.2.4 on page 482.
- **ompd_word_t** type, see Section 4.2.2.1.3 on page 480.
- *nthreads-var* ICV, see Section 2.4 on page 49.

1 #### 4.2.4.7.14 `ompd_get_thread_num`

2 **Summary**

3 The **ompd_get_thread_num** function returns the logical thread number of the thread in the
4 team of the OpenMP process.

5 **Format**

C

```
ompd_rc_t ompd_get_thread_num (
  const ompd_thread_handle_t *task_handle,
  ompd_word_t *val
);
```

C

6 **Description**

7 The **ompd_get_thread_num** function returns the logical thread number of the thread in the
8 team of the OpenMP process, and corresponds to the **omp_get_thread_num** function in the
9 OpenMP runtime API.

10 **Description of Arguments**

11 The input argument *task_handle* is an opaque handle for a task and selects the task to operate on.
12 On return the output argument *val* is set to the value that **omp_get_thread_num** would return
13 when executed in this task.

14 **Cross References**

15 • **ompd_thread_handle_t** type, see Section 4.2.2.4 on page 482.

16 • **ompd_word_t** type, see Section 4.2.2.1.3 on page 480.

17 #### 4.2.4.7.15 `ompd_in_parallel`

18 **Summary**

19 The **ompd_in_parallel** function returns *true* if *active-levels-var* is greater than 0, and *false*
20 otherwise.

Format

C

```
ompd_rc_t ompd_in_parallel (
  const ompd_task_handle_t  *task_handle,
  ompd_word_t  *val
);
```

C

Description

ompd_in_parallel returns logical true (*i.e.*, ***val != 0**) if *active-levels-var* ICV is greater than 0, and false (0) otherwise. The routine corresponds to **omp_in_parallel** in the OpenMP runtime.

Description of Arguments

The input argument *task_handle* is an opaque handle for a task and selects the task to operate on. On return the output argument *val* is set to the value that **omp_in_parallel** would return when executed in this task.

Cross References

- **ompd_task_handle_t** type, see Section 4.2.2.4 on page 482.
- **ompd_word_t** type, see Section 4.2.2.1.3 on page 480.
- *active-levels-var* ICV, see Section 2.4 on page 49.

4.2.4.7.16 ompd_in_final

Summary

The **ompd_in_final** function returns *true* if the task is a final task.

Format

C

```
ompd_rc_t ompd_in_final (
  const ompd_task_handle_t  *task_handle,
  ompd_word_t    *val
);
```

C

Description

ompd_in_final corresponds to **omp_in_final** and returns logical true if the task is a final task.

Description of Arguments

The input argument *task_handle* is an opaque handle for a task and selects the task to operate on. On return the output argument *val* is set to the value that **omp_in_final** would return when executed in this task.

Cross References

- **ompd_task_handle_t** type, see Section 4.2.2.4 on page 482.
- **ompd_word_t** type, see Section 4.2.2.1.3 on page 480.

4.2.4.7.17 ompd_get_dynamic

Summary

The **ompd_get_dynamic** function returns the value of the *dyn-var* ICV.

Format

C

```
ompd_rc_t ompd_get_dynamic (
  const ompd_task_handle_t  *task_handle,
  ompd_word_t    *val
);
```

C

1 Description

2 **ompd_get_dynamic** returns the value of the *dyn-var* ICV, and corresponds to the
3 **omp_get_dynamic** member of the OpenMPI API.

4 Description of Arguments

5 The input argument *task_handle* is an opaque handle for a task and selects the task to operate on.
6 On return the output argument *val* is set to the value that **omp_get_dynamic** would return when
7 executed in this task.

8 Cross References

9 • **ompd_task_handle_t** type, see Section 4.2.2.4 on page 482.

10 • **ompd_word_t** type, see Section 4.2.2.1.3 on page 480.

11 • *dyn-var* ICV, see Section 2.4 on page 49.

12 4.2.4.7.18 **ompd_get_nested**

13 Summary

14 The **ompd_get_nested** function returns the value of the *nest-var* ICV.

15 Format

C

```
ompd_rc_t ompd_get_nested (
  const ompd_task_handle_t  *task_handle,
  ompd_word_t               *val
);
```

C

16 Description

17 **ompd_get_nested** corresponds to **omp_get_nested**, and returns the value of the *nest-var*
18 ICV.

Description of Arguments

The input argument *task_handle* is an opaque handle for a task and selects the task to operate on. On return the output argument *val* is set to the value that **omp_get_nested** would return when executed in this task.

Cross References

- **ompd_task_handle_t** type, see Section 4.2.2.4 on page 482.
- **ompd_word_t** type, see Section 4.2.2.1.3 on page 480.
- *nest-var* ICV, see Section 2.4 on page 49.

4.2.4.7.19 ompd_get_max_active_levels

Summary

The **ompd_get_max_active_levels** function returns the value of the *max-active-levels-var* ICV.

Format

C

```
ompd_rc_t ompd_get_max_active_levels (
  const ompd_thread_handle_t  *thread_handle,
  ompd_word_t *val
);
```

C

Description

The maximum number of nested levels parallelism is returned by **get_max_active_levels**. This operation corresponds to the OpenMP routine **omp_get_max_active_levels** and the ICV *max-active-levels-var*.

Description of Arguments

The input argument *task_handle* is an opaque handle for a task and selects the task to operate on. On return the output argument *val* is set to the value that **omp_get_max_active_levels** would return when executed in this task.

Cross References

- **ompd_thread_handle_t** type, see Section 4.2.2.4 on page 482.
- **ompd_word_t** type, see Section 4.2.2.1.3 on page 480.
- *max-active-levels-var* ICV, see Section 2.4 on page 49.

4.2.4.7.20 ompd_get_schedule

Summary

The **ompd_get_schedule** function returns information about the schedule when **runtime** scheduling is used.

Format

C

```
ompd_rc_t ompd_get_schedule (
  ompd_task_handle_t        *task_handle,
  ompd_sched_t              *kind,
  ompd_word_t              *modifier
);
```

C

Description

ompd_get_schedule returns information about the schedule that is applied when **runtime** scheduling is used. This information is represented in the device by the *run-sched-var*.

Description of Arguments

The input argument *task_handle* is an opaque handle for a task and selects the task to operate on. On return the output argument *kind* is set to the schedule that **omp_get_schedule** would return when executed in this task; the output argument *modifier* is set to the *chunk_size* that **omp_get_schedule** would return.

Cross References

- **ompd_task_handle_t** type, see Section 4.2.2.4 on page 482.
- **ompd_sched_t** type, see Section 4.2.2.7 on page 484.
- **ompd_word_t** type, see Section 4.2.2.1.3 on page 480.
- *run-sched-var* ICV, see Section 2.4 on page 49.

4.2.4.7.21 ompd_get_proc_bind

Summary

The **ompd_get_proc_bind** function returns the value of the task's *bind-var* ICV.

Format

C

```
ompd_rc_t ompd_get_proc_bind (
  ompd_task_handle_t        *task_handle,
  ompd_proc_bind_t          *bind
);
```

C

Description

ompd_get_proc_bind returns the value of the task's *bind-var* ICV (Section 2.4 on page 49), which "controls the binding of the OpenMP threads to places," or "default thread affinity policies." The OMPD API defines **ompd_proc_bind_t**, which contains the corresponding OpenMP enumeration values.

Description of Arguments

The input argument *task_handle* is an opaque handle for a task and selects the task to operate on. On return the output argument *bind* is set to the value that **omp_get_proc_bind** would return when executed in this task.

Cross References

- **ompd_task_handle_t** type, see Section 4.2.2.4 on page 482.
- **ompd_proc_bind_t** type, see Section 4.2.2.8 on page 485.
- *bind-var* ICV, see Section 2.4 on page 49.

4.2.4.7.22 ompd_is_implicit

Summary

The **ompd_is_implicit** function returns *true* if a task is implicit, and *false* otherwise.

Format

C

```
ompd_rc_t ompd_is_implicit (
  ompd_task_handle_t   *task_handle,
  ompd_word_t          *val
);
```

C

Description

ompd_is_implicit returns logical true (*i.e.*, ***val != 0**) if a task is implicit, and false (0) otherwise. The routine has no corresponding call in the OpenMP runtime.

Description of Arguments

The input argument *task_handle* is an opaque handle for a task and selects the task to operate on. On return the output argument *val* is set to the value that **omp_get_dynamic** would return when executed in this task.

Cross References

- **ompd_task_handle_t** type, see Section 4.2.2.4 on page 482.
- **ompd_word_t** type, see Section 4.2.2.1.3 on page 480.

4.2.4.7.23 `ompd_get_task_frame`

Summary

For the specified task, extract the task's frame pointers maintained by an OpenMP implementation.

Format

C

```
ompd_rc_t ompd_get_task_frame (
  ompd_task_handle_t   *task_handle,
  ompd_address_t   *exit_frame,
  ompd_address_t *enter_frame
);
```

C

Description

An OpenMP implementation maintains an **omp_frame_t** object for every implicit or explicit task. For the task identified by *task_handle*, **ompd_get_task_frame** extracts the *enter_frame* and *exit_frame* fields of the task's **omp_frame_t** object.

Description of Arguments

The argument *task_handle* specifies an OpenMP task.

The argument *exit_frame* is a pointer to an **ompd_address_t** object that the OMPD plugin will modify to return the segment and address that represent the value of the *exit_frame* field of the **omp_frame_t** object associated with the specified task.

The argument *enter_frame* is a pointer to an **ompd_address_t** object that the OMPD plugin will modify to return the segment and address that represent the value of the *enter_frame* field of the **omp_frame_t** object associated with the specified task.

Cross References

- **ompd_task_handle_t** type, see Section 4.2.2.4 on page 482.
- **ompd_address_t** type, see Section 4.2.2.1.4 on page 480.
- **omp_frame_t** type, see Section 4.3.1.2 on page 556.

4.2.4.7.24 `ompd_get_state`

Summary

This function allows a third party tool to interrogate the OMPD plugin about the state of a thread.

Format

C

```
ompd_rc_t ompd_get_state (
  ompd_thread_handle_t *thread_handle,
  omp_state_t *state,
  omp_wait_id_t     *wait_id
);
```

C

Description

The function **`ompd_get_state`** is the OMPD version of **`ompt_get_state`**. The only difference between the OMPD and OMPT counterparts is that the OMPD version must supply a thread handle to provide a context for this inquiry.

Description of Arguments

The input argument *thread_handle* is a thread handle. The output argument *state* represents the state of the thread that is represented by the thread handle. The thread states are represented by values of the enumeration type **`omp_state_t`**.

The output argument *wait_id* is a pointer to an opaque handle available to receive the value of the thread's wait identifier. If the *wait_id* pointer is not **`NULL`**, it will contain the value of the thread's wait identifier **wait_id*. If the thread state is not one of the specified wait states, the value of **wait_id* is undefined.

Cross References

- **`ompd_thread_handle_t`** type, see Section 4.2.2.4 on page 482.
- **`omp_state_t`** type, see Section 4.3.1.1 on page 551.
- **`ompd_wait_id_t`** type, see Section 4.2.2.1.2 on page 480.

4.2.4.8 Display Control variables

4.2.4.8.1 ompd_get_display_control_vars

Summary

Returns a list of name/value pairs for the OpenMP control variables that are user-controllable and important to the operation or performance of OpenMP programs.

Format

C

```
ompd_rc_t ompd_get_display_control_vars (
  ompd_address_space_handle_t *address_space_handle,
  const char * const *  *control_vars
);
```

C

Description

The function **ompd_get_display_control_vars** returns a NULL-terminated vactor of strings of name/value pairs of control variables whose settings are (a) user controllable, and (b) importannt to the operation or performance of an OpenMP runtime system. The control variables exposed through this interface include all of the OpenMP environment variables, settings that may come from vendor or platform-specific environment variables, and other settings that affect the operation or functioning of an OpenMP runtime.

The format of the strings is:

name=a string

The third-party tool must not modify the vector or the strings (i.e., they are both **const**). The strings are NULL terminated. The vector is NULL terminated.

After returning from the call, the vector and strings are 'owned' by the third third-party tool. Providing the termination constraints are satisfied, the OMPD implementation is free to use static or dynamic memory for the vector and/or the strings, and to arrange them in memory as it pleases. If dynamic memory is used, then the OMPD implementation must use the allocate callback it received in the call to **ompd_initialize**. As the third-party tool cannot make any assumptions about the origin or layout of the memory used for the vector or strings, it cannot release the display control variables directly when they are no longer needed; instead it must use **ompd_release_display_control_vars ()**.

Description of Arguments

The address space is identified by the input argument *address_space_handle*. The vector of display control variables is returned through the output argument *control_vars*.

Cross References

- **`ompd_address_space_handle_t`** type, see Section 4.2.2.4 on page 482.
- **`ompd_release_display_control_vars`** type, see Section 4.2.4.8.2 on page 544.

4.2.4.8.2 `ompd_release_display_control_vars`

Summary

Releases a list of name/value pairs of OpenMP control variables previously acquired using **`ompd_get_display_control_vars`**.

Format

C

```
ompd_rc_t ompd_release_display_control_vars (
  const char * const **control_vars
);
```

C

Description

The vector and strings returned from **`ompd_get_display_control_vars`** are 'owned' by the thrid-party, but allocated by the OMPD. Because the third-party tool doesn't know how the memory for the vector and strings was alloacted, it cannot deallocate the memory itself. Instead, the third-party tool must call **`ompd_release_display_control_vars`** to release the vector and strings.

Description of Arguments

The input parameter *control_vars* is the vector of display control variables to be released.

Cross References

- **ompd_get_display_control_vars** type, see Section 4.2.4.8.1 on page 543.

4.2.5 Runtime Entry Points for OMPD

Most of the tool's OpenMP-related activity on an OpenMP program will be performed through the OMPD interface. However, supporting OMPD introduced some requirements of the OpenMP runtime. These fall into three categories: entrypoints the user's code in OpenMP program can call; locations in the OpenMP runtime through which control must pass when specific events occur; and data that must be accessible to the tool.

4.2.5.1 Event notification

Neither a tool nor an OpenMP runtime system know what application code a program will launch as parallel regions or tasks until the program invokes the runtime system and provides a code address as an argument. To help a tool control the execution of an OpenMP program launching parallel regions or tasks, the OpenMP runtime must define a number of symbols through which execution must pass when particular events occur *and* data collection for OMPD is enabled. These locations may, but do not have to, be subroutines, They may, for example, be labeled locations. The symbols must all have external, **C**, linkage.

A tool can gain notification of the event by planting a breakpoint at the corresponding named location.

4.2.5.1.1 Beginning Parallel Regions

Summary

The OpenMP runtime must execute through **ompd_bp_parallel_begin** when a new parallel region is launched.

C

```
void ompd_bp_parallel_begin ( void );
```

C

Description

When starting a new parallel region, the runtime must allow execution to flow through **ompd_bp_parallel_begin**. This should occur after a task encounters a parallel construct, but before any implicit task starts to execute the parallel region's work.

Control passes through **ompd_bp_parallel_begin** once per region, and not once for each thread per region.

At the point where the runtime reaches **ompd_bp_parallel_begin**, a tool can map the encountering native thread to an OpenMP thread handle using **ompd_get_thread_handle**. At this point the handle returned by **ompd_get_current_parallel_handle** is that of the new parallel region. The tool can find the entry point of the user code that the new parallel region will execute by passing the parallel handle region to **ompd_get_parallel_function**.

The actual number of threads, rather than the requested number of threads, in the team is returned by **ompd_get_num_threads**.

The task handle returned by **ompd_get_current_task_handle** will be that of the task encountering the parallel construct.

The 'reenter runtime' address in the information returned by **ompd_get_task_frame** will be that of the stack frame where the thread called the OpenMP runtime to handle the parallel construct. The 'exit runtime' address will be for the stack frame where the thread left the OpenMP runtime to execute the user code that encountered the parallel construct.

Restrictions

ompd_bp_parallel_begin has external **C** linkage, and no demangling or other transformations are required by a tool to look up its address in the OpenMP program.

Conceptually **ompd_bp_parallel_begin** has the type signature given above. However, it does not need to be a function, but can be a labeled location in the runtime code.

Cross References

- **ompd_get_thread_handle**, Section 4.2.4.5.2 on page 511
- **ompd_get_current_parallel_handle**, Section 4.2.4.6.1 on page 515
- **ompd_get_parallel_function**, Section 4.2.4.7.9 on page 528
- **ompd_get_num_threads**, Section 4.2.4.7.8 on page 528
- **ompd_get_current_task_handle**, Section 4.2.4.7.1 on page 521
- **ompd_get_task_frame**, Section 4.2.4.7.23 on page 541

4.2.5.1.2 Ending Parallel Regions

The OpenMP runtime must execute through **ompd_bp_parallel_end** when a parallel region ends.

C

```
void ompd_bp_parallel_end ( void );
```

C

Description

When a parallel region finishes, the OpenMP runtime must allow execution to flow through **ompd_bp_parallel_end**.

Control passes through **ompd_bp_parallel_end** once per region, and not once for each thread per region.

At the point the runtime reaches **ompd_bp_parallel_end** the tool can map the encountering native thread to an OpenMP thread handle using **ompd_get_thread_handle**. **ompd_get_current_parallel_handle** returns the handle of the terminating parallel region.

ompd_get_current_task_handle returns the handle of the task that encountered the parallel construct that initiated the parallel region just terminating. The 'reenter runtime' address in the frame information returned by **ompd_get_task_frame** will be that for the stack frame in which the thread called the OpenMP runtime to start the parallel construct just terminating. The 'exit runtime' address will refer to the stack frame where the thread left the OpenMP runtime to execute the user code that invoked the parallel construct just terminating.

Restrictions

ompd_bp_parallel_end has external **C** linkage, and no demangling or other transformations are required by a tool to look up its address in the OpenMP program.

Conceptually **ompd_bp_parallel_end** has the type signature given above. However, it does not need to be a function, but can be a labeled location in the runtime code.

Cross References

- **ompd_get_thread_handle**, Section 4.2.4.5.2 on page 511
- **ompd_get_current_parallel_handle**, Section 4.2.4.6.1 on page 515
- **ompd_get_current_task_handle**, Section 4.2.4.7.1 on page 521
- **ompd_get_task_frame**, Section 4.2.4.7.23 on page 541

4.2.5.1.3 Beginning Task Regions

The OpenMP runtime must execute through **ompd_bp_task_begin** when a new task is started.

C

```
void ompd_bp_task_begin ( void );
```

C

Description

When starting a new task region, the OpenMP runtime system must allow control to pass through **ompd_bp_task_begin**.

The OpenMP runtime system will execute through this location after the task construct is encountered, but before the new explicit task starts. At the point where the runtime reaches **ompd_bp_task_begin** the tool can map the native thread to an OpenMP handle using **ompd_get_thread_handle**.

ompd_get_current_task_handle returns the handle of the new task region. The entry point of the user code to be executed by the new task is returned from **ompd_get_task_function**.

Restrictions

ompd_bp_task_begin has external **C** linkage, and no demangling or other transformations are required by a tool to look up its address in the OpenMP program.

Conceptually **ompd_bp_task_begin** has the type signature given above. However, it does not need to be a function, but can be a labeled location in the runtime code.

Cross References

- **ompd_get_thread_handle**, Section 4.2.4.5.2 on page 511
- **ompd_get_current_task_handle**, Section 4.2.4.7.1 on page 521
- **ompd_get_task_function**, Section 4.2.4.7.12 on page 531

4.2.5.1.4 Ending Task Regions

Summary

The OpenMP runtime must execute through **ompd_bp_task_end** when a task region ends.

C

```
void ompd_bp_task_end ( void );
```

C

Description

When a task region completes, the OpenMP runtime system must allow execution to flow through the location **ompd_bp_task_end**.

At the point where the runtime reaches **ompd_bp_task_end** the tool can use **ompd_get_thread_handle** to map the encountering native thread to the corresponding OpenMP thread handle. At this point **ompd_get_current_task_handle** returns the handle for the terminating task.

Restrictions

ompd_bp_task_end has external **C** linkage, and no demangling or other transformations are required by a tool to look up its address in the OpenMP program.

Conceptually **ompd_bp_task_end** has the type signature given above. However, it does not need to be a function, but can be a labeled location in the runtime code.

Cross References

- **ompd_get_thread_handle**, Section 4.2.4.5.2 on page 511
- **ompd_get_current_task_handle**, Section 4.2.4.7.1 on page 521

4.2.6 Entry Points for OMPD defined in the OpenMP program

There is a small group of OpenMP entrypoints that are defined in OpenMP program, rather than in the OpenMP runtime. Unless otherwise stated, these entrypoints need not be defined in a program, in which case default behavior will apply.

4.2.6.1 Enabling Support for OMPD at Runtime

Summary

Instructs the OpenMP runtime to collect whatever information is necessary for supporting access by tools using OMPD.

C

```
int ompd_enable ( void );
```

C

Description

In some cases it may not be possible to control an OpenMP program's environment to set the **OMPD_ENABLED** variable so as to enable data collection by the runtime for OMPD. **ompd_enable** allows an OpenMP process itself to turn on data collection. Upon starting, the OpenMP runtime will check to see if the function **ompd_enable** is defined in the OpenMP program or any of its dynamically-linked libraries. If it is defined, the OpenMP runtime will call the function, and if logical true (non-zero) is returned, it will enable data collection to support external tools. The function may be positioned in an otherwise empty DLL that the programmer can link with the OpenMP program, thereby leaving the program code unmodified.

Cross References

- **OMPD_ENABLED**, Section 5.20 on page 580
- Enabling the Runtime for OMPD, Section 4.2.1.1 on page 478

4.3 Tool Foundation

4.3.1 Data Types

4.3.1.1 Thread States

To enable a tool to understand the behavior of an executing program, an OpenMP implementation maintains a state for each thread. The state maintained for a thread is an approximation of the thread's instantaneous state.

C / C++

A thread's state will be one of the values of the enumeration type **omp_state_t** or an implementation-defined state value of 512 or higher. Thread states in the enumeration fall into several classes: work, barrier wait, task wait, mutex wait, target wait, and miscellaneous.

```
typedef enum omp_state_t {
  omp_state_work_serial                       = 0x000,
  omp_state_work_parallel                     = 0x001,
  omp_state_work_reduction                    = 0x002,

  omp_state_wait_barrier                      = 0x010,
  omp_state_wait_barrier_implicit_parallel    = 0x011,
  omp_state_wait_barrier_implicit_workshare   = 0x012,
  omp_state_wait_barrier_implicit             = 0x013,
  omp_state_wait_barrier_explicit             = 0x014,

  omp_state_wait_taskwait                     = 0x020,
  omp_state_wait_taskgroup                    = 0x021,

  omp_state_wait_mutex                        = 0x040,
  omp_state_wait_lock                         = 0x041,
  omp_state_wait_critical                     = 0x042,
  omp_state_wait_atomic                       = 0x043,
  omp_state_wait_ordered                      = 0x044,

  omp_state_wait_target                       = 0x080,
  omp_state_wait_target_map                   = 0x081,
  omp_state_wait_target_update                = 0x082,

  omp_state_idle                              = 0x100,
  omp_state_overhead                          = 0x101,
  omp_state_undefined                         = 0x102
} omp_state_t;
```

C / C++

A tool can query the OpenMP state of a thread at any time. If a tool queries the state of a thread that is not associated with OpenMP, the implementation reports the state as **omp_state_undefined**.

Some values of the enumeration type **omp_state_t** are used by all OpenMP implementations, e.g., **omp_state_work_serial**, which indicates that a thread is executing in a serial region, and **omp_state_work_parallel**, which indicates that a thread is executing in a parallel

region. Other values of the enumeration type describe a thread's state at different levels of specificity. For instance, an OpenMP implementation may use the state **omp_state_wait_barrier** to represent all waiting at barriers. It may differentiate between waiting at implicit or explicit barriers using **omp_state_wait_barrier_implicit** and **omp_state_wait_barrier_explicit**. To provide full detail about the type of an implicit barrier, a runtime may report **omp_state_wait_barrier_implicit_parallel** or **omp_state_wait_barrier_implicit_workshare** as appropriate.

For states that represent waiting, an OpenMP implementation has the choice of transitioning a thread to such states early or late. For instance, when an OpenMP thread is trying to acquire a lock, there are several points at which an OpenMP implementation transition the thread to the **omp_state_wait_lock** state. One implementation may transition the thread to the state early before the thread attempts to acquire a lock. Another implementation may transition the thread to the state late, only if the thread begins to spin or block to wait for an unavailable lock. A third implementation may transition the thread to the state even later, e.g., only after the thread waits for a significant amount of time.

The following sections describe the classes of states and the states in each class.

4.3.1.1.1 Work States

An OpenMP implementation reports a thread in a work state when the thread is performing serial work, parallel work, or a reduction.

omp_state_work_serial

The thread is executing code outside all parallel regions.

omp_state_work_parallel

The thread is executing code within the scope of a parallel region construct.

omp_state_work_reduction

The thread is combining partial reduction results from threads in its team. An OpenMP implementation might never report a thread in this state; a thread combining partial reduction results may have its state reported as **omp_state_work_parallel** or **omp_state_overhead**.

4.3.1.1.2 Barrier Wait States

An OpenMP implementation reports that a thread is in a barrier wait state when the thread is awaiting completion of a barrier.

omp_state_wait_barrier

The thread is waiting at either an implicit or explicit barrier. A thread may enter this state early, when the thread encounters a barrier, or late, when the thread begins to wait at the barrier. An implementation may never report a thread in this state; instead, a thread may have its state reported as **omp_state_wait_barrier_implicit** or **omp_state_wait_barrier_explicit**, as appropriate.

omp_state_wait_barrier_implicit

The thread is waiting at an implicit barrier in a parallel region. A thread may enter this state early, when the thread encounters a barrier, or late, when the thread begins to wait at the barrier. An OpenMP implementation may report **omp_state_wait_barrier** for implicit barriers.

omp_state_wait_barrier_implicit_parallel

The description of when a thread reports a state associated with an implicit barrier is described for state **omp_state_wait_barrier_implicit**. An OpenMP implementation may report **omp_state_wait_barrier_implicit_parallel** for an implicit barrier that occurs at the end of a parallel region. As explained in Section 4.1.4.2.12 on page 418, reporting the state **omp_state_wait_barrier_implicit_parallel** permits a weaker contract between a runtime and a tool that enables a simpler and faster implementation of parallel regions.

omp_state_wait_barrier_implicit_workshare

The description of when a thread reports a state associated with an implicit barrier is described for state **omp_state_wait_barrier_implicit**. An OpenMP implementation may report **omp_state_wait_barrier_implicit_parallel** for an implicit barrier that occurs at the end of a worksharing construct.

omp_state_wait_barrier_explicit

The thread is waiting at an explicit barrier in a parallel region. A thread may enter this state early, when the thread encounters a barrier, or late, when the thread begins to wait at the barrier. An implementation may report **omp_state_wait_barrier** for explicit barriers.

4.3.1.1.3 Task Wait States

omp_state_wait_taskwait

The thread is waiting at a taskwait construct. A thread may enter this state early, when the thread encounters a taskwait construct, or late, when the thread begins to wait for an uncompleted task.

omp_state_wait_taskgroup

The thread is waiting at the end of a taskgroup construct. A thread may enter this state early, when the thread encounters the end of a taskgroup construct, or late, when the thread begins to wait for an uncompleted task.

4.3.1.1.4 Mutex Wait States

OpenMP provides several mechanisms that enforce mutual exclusion: locks as well as critical, atomic, and ordered sections. This grouping contains all states used to indicate that a thread is awaiting exclusive access to a lock, critical section, variable, or ordered section.

An OpenMP implementation may report a thread waiting for any type of mutual exclusion using either a state that precisely identifies the type of mutual exclusion, or a more generic state such as **omp_state_wait_mutex** or **omp_state_wait_lock**. This flexibility may significantly simplify the maintenance of states associated with mutual exclusion in the runtime when various mechanisms for mutual exclusion rely on a common implementation, e.g., locks.

omp_state_wait_mutex

The thread is waiting for a mutex of an unspecified type. A thread may enter this state early, when a thread encounters a lock acquisition or a region that requires mutual exclusion, or late, when the thread begins to wait.

omp_state_wait_lock

The thread is waiting for a lock or nest lock. A thread may enter this state early, when a thread encounters a lock **set** routine, or late, when the thread begins to wait for a lock.

omp_state_wait_critical

The thread is waiting to enter a critical region. A thread may enter this state early, when the thread encounters a critical construct, or late, when the thread begins to wait to enter the critical region.

omp_state_wait_atomic

The thread is waiting to enter an atomic region. A thread may enter this state early, when the thread encounters an atomic construct, or late, when the thread begins to wait to enter the atomic region. An implementation may opt not to report this state when using atomic hardware instructions that support non-blocking atomic implementations.

omp_state_wait_ordered

The thread is waiting to enter an ordered region. A thread may enter this state early, when the thread encounters an ordered construct, or late, when the thread begins to wait to enter the ordered region.

4.3.1.1.5 Target Wait States

omp_state_wait_target

The thread is waiting for a target region to complete.

omp_state_wait_target_map

The thread is waiting for a target data mapping operation to complete. An implementation may report **omp_state_wait_target** for target data constructs.

omp_state_wait_target_update

The thread is waiting for a target update operation to complete. An implementation may report **omp_state_wait_target** for target update constructs.

4.3.1.1.6 Miscellaneous States

omp_state_idle

The thread is idle, waiting for work.

omp_state_overhead

A thread may be reported as being in the overhead state at any point while executing within an OpenMP runtime, except while waiting indefinitely at a synchronization point. An OpenMP implementation report a thread's state as a work state for some or all of the time the thread spends in executing in the OpenMP runtime.

omp_state_undefined

This state is reserved for threads that are not user threads, initial threads, threads currently in an OpenMP team, or threads waiting to become part of an OpenMP team.

4.3.1.2 Frames

C / C++

```
typedef struct omp_frame_t {
  void *exit_frame;
  void *enter_frame;
} omp_frame_t;
```

C / C++

Description

When executing an OpenMP program, at times, one or more procedure frames associated with the OpenMP runtime may appear on a thread's stack between frames associated with tasks. To help a tool determine whether a procedure frame on the call stack belongs to a task or not, for each task whose frames appear on the stack, the runtime maintains an **omp_frame_t** object that indicates a contiguous sequence of procedure frames associated with the task. Each **omp_frame_t** object is associated with the task to which the procedure frames belong. Each non-merged initial, implicit, explicit, or target task with one or more frames on a thread's stack will have an associated **omp_frame_t** object.

An **omp_frame_t** object associated with a task contains a pair of pointers: *exit_frame* and *enter_frame*. The field names were chosen, respectively, to reflect that they typically contain a pointer to a procedure frame on the stack when *exiting* the OpenMP runtime into code for a task or *entering* the OpenMP runtime from a task.

The *exit_frame* field of a task's **omp_frame_t** object contains the canonical frame address for the procedure frame that transfers control to the structured block for the task. The value of *exit_frame* is **NULL** until just prior to beginning execution of the structured block for the task. A task's *exit_frame* may point to a procedure frame that belongs to the OpenMP runtime or one that belongs to another task. The *exit_frame* for the **omp_frame_t** object associated with an *initial task* is **NULL**.

The *enter_frame* field of a task's **omp_frame_t** object contains the canonical frame address of a task procedure frame that invoked the OpenMP runtime causing the current task to suspend and another task to execute. If a task with frames on the stack has not suspended, the value of *enter_frame* for the **omp_frame_t** object associated with the task may contain **NULL**. The value of *enter_frame* in a task's **omp_frame_t** is reset to **NULL** just before a suspended task resumes execution.

An **omp_frame_t**'s lifetime begins when a task is created and ends when the task is destroyed. Tools should not assume that a frame structure remains at a constant location in memory throughout a task's lifetime. A pointer to a task's **omp_frame_t** object is passed to some callbacks; a pointer to a task's **omp_frame_t** object can also be retrieved by a tool at any time, including in a signal handler, by invoking the **ompt_get_task_info** runtime entry point (described in Section 4.1.5.1.14).

Table 4.6 describes various states in which an **omp_frame_t** object may be observed and their meaning. In the presence of nested parallelism, a tool may observe a sequence of **omp_frame_t** objects for a thread. Appendix D illustrates use of **omp_frame_t** objects with nested parallelism.

Note – A monitoring tool using asynchronous sampling can observe values of *exit_frame* and *enter_frame* at inconvenient times. Tools must be prepared to observe and handle **omp_frame_t** objects observed just prior to when their field values will be set or cleared.

TABLE 4.6: Meaning of various states of an **omp_frame_t** object.

exit_frame / *enter_frame* state	*enter_frame* is **NULL**	*enter_frame* is non-**NULL**
exit_frame is **NULL**	case 1) initial task during execution case 2) task that is created but not yet scheduled or already finished	initial task suspended while another task executes
exit_frame is non-**NULL**	non-initial task that has been scheduled	non-initial task suspended while another task executes

4.3.1.3 Wait Identifiers

Each thread instance maintains a *wait identifier* of type **omp_wait_id_t**. When a task executing on a thread is waiting for mutual exclusion, the thread's wait identifer indicates what the thread is awaiting. A wait identifier may represent a critical section *name*, a lock, a program variable accessed in an atomic region, or a synchronization object internal to an OpenMP implementation.

C / C++

```
typedef uint64_t omp_wait_id_t;
static const ompt_wait_id_t ompt_wait_id_none = 0;
```

C / C++

When a thread is not in a wait state, the value of the thread's wait identifier is undefined.

4.3.2 Global Symbols

Many of the interfaces between tools and an OpenMP implementation are invisible to users. This section describes a few global symbols used by OMPT and OMPD tools to coordinate with an OpenMP implementation.

4.3.2.1 ompt_start_tool

Summary

If a tool wants to use the OMPT interface provided by an OpenMP implementation, the tool must implement the function **ompt_start_tool** to announce its interest.

1 **Format**

C

```
ompt_start_tool_result_t *ompt_start_tool(
  unsigned int omp_version,
  const char *runtime_version
);
```

C

2 ### Description

3 For a tool to use the OMPT interface provided by an OpenMP implementation, the tool must define
4 a globally-visible implementation of the function **ompt_start_tool**.

5 A tool may indicate its intent to use the OMPT interface provided by an OpenMP implementation
6 by having **ompt_start_tool** return a non-**NULL** pointer to an
7 **ompt_start_tool_result_t** structure, which contains pointers to tool initialization and
8 finalization callbacks along with a tool data word that an OpenMP implementation must pass by
9 reference to these callbacks.

10 A tool may use its argument *omp_version* to determine whether it is compatible with the OMPT
11 interface provided by an OpenMP implementation.

12 If a tool implements **ompt_start_tool** but has no interest in using the OMPT interface in a
13 particular execution, **ompt_start_tool** should return **NULL**.

14 ### Description of Arguments

15 The argument *omp_version* is the value of the **_OPENMP** version macro associated with the
16 OpenMP API implementation. This value identifies the OpenMP API version supported by an
17 OpenMP implementation, which specifies the version of the OMPT interface that it supports.

18 The argument *runtime_version* is a version string that unambiguously identifies the OpenMP
19 implementation.

20 ### Constraints on Arguments

21 The argument *runtime_version* must be an immutable string that is defined for the lifetime of a
22 program execution.

Effect

If a tool returns a non-**NULL** pointer to an **ompt_start_tool_result_t** structure, an OpenMP implementation will call the tool initializer specified by the *initialize* field in this structure before beginning execution of any OpenMP construct or completing execution of any environment routine invocation; the OpenMP implementation will call the tool finalizer specified by the *finalize* field in this structure when the OpenMP implementation shuts down.

Cross References

- **ompt_start_tool_result_t**, see Section 4.1.3.1 on page 388.

4.3.2.2 ompd_dll_locations

Summary

The global variable **ompd_dll_locations** indicates where a tool should look for OMPD plugin(s) that are compatible with the OpenMP implementation.

C

```
const char **ompd_dll_locations;
```

C

Description

ompd_dll_locations is an **argv**-style vector of filename strings that provide the names of any OMPD plugin implementations that are compatible with the OpenMP runtime. The vector is NULL-terminated.

The programming model or architecture of the third-party tool, and hence that of the required OMPD plugin, might not match that of the OpenMP program to be examined. On platforms that support multiple programming models (*e.g.*, 32- v. 64-bit), or in heterogenous environments where the architectures of the OpenMP program and third-party tool may be be different, OpenMP implementors are encourgaed to provide OMPD plugins for all models. The vector, therefore, may name plugins that are not compatible with the third-party tool. This is legal, and it is up to the third-party tool to check that a plugin is compatible. (Typically, a tool might iterate over the vector until a compatible plugin is found.)

Restrictions

ompd_dll_locations has external **C** linkage, no demangling or other transformations are required by a third-party tool before looking up its address in the OpenMP program.

The vector and its members must be fully initialized before **ompd_dll_locations** is set to a non-NULL value. That is, if **ompd_dll_locations** is not NULL, the vector and its contents are valid.

Cross References

- **ompd_dll_locations_valid**, Section 4.3.2.3 on page 561
- Finding the OMPD plugin, Section 4.2.1.2 on page 478

4.3.2.3 ompd_dll_locations_valid

Summary

The OpenMP runtime notifies third-party tools that **ompd_dll_locations** is valid by allowing execution to pass through a location identified by the symbol **ompd_dll_locations_valid**.

C

```
void ompd_dll_locations_valid(void);
```

C

Description

Depending on how the OpenMP runtime is implemented, **ompd_dll_locations** might not be a static variable, and therefore needs to be initialized at runtime. The OpenMP runtime notifies third-party tools that **ompd_dll_locations** is valid by having execution pass through a location identified by the symbol **ompd_dll_locations_valid**. If **ompd_dll_locations** is NULL, a third-party tool, e.g., a debugger can place a breakpoint at **ompd_dll_locations_valid** to be notified when **ompd_dll_locations** has been initialized. In practice, the symbol **ompd_dll_locations_valid** need not be a function; instead, it may be a labeled machine instruction through which execution passes once the vector is valid.

This page intentionally left blank

1 CHAPTER 5

2 Environment Variables

3 This chapter describes the OpenMP environment variables that specify the settings of the ICVs that
4 affect the execution of OpenMP programs (see Section 2.4 on page 49). The names of the
5 environment variables must be upper case. The values assigned to the environment variables are
6 case insensitive and may have leading and trailing white space. Modifications to the environment
7 variables after the program has started, even if modified by the program itself, are ignored by the
8 OpenMP implementation. However, the settings of some of the ICVs can be modified during the
9 execution of the OpenMP program by the use of the appropriate directive clauses or OpenMP API
10 routines.

11 The following examples demonstrate how the OpenMP environment variables can be set in
12 different environments:

13 • csh-like shells:

```
setenv OMP_SCHEDULE "dynamic"
```

14 • bash-like shells:

```
export OMP_SCHEDULE="dynamic"
```

15 • Windows Command Line:

```
set OMP_SCHEDULE=dynamic
```

5.1 OMP_SCHEDULE

The **OMP_SCHEDULE** environment variable controls the schedule type and chunk size of all loop directives that have the schedule type **runtime**, by setting the value of the *run-sched-var* ICV.

The value of this environment variable takes the form:

type[, *chunk*]

where

- *type* is one of **static**, **dynamic**, **guided**, or **auto**
- *chunk* is an optional positive integer that specifies the chunk size

If *chunk* is present, there may be white space on either side of the "**,**". See Section 2.10.1 on page 78 for a detailed description of the schedule types.

The behavior of the program is implementation defined if the value of **OMP_SCHEDULE** does not conform to the above format.

Implementation specific schedules cannot be specified in **OMP_SCHEDULE**. They can only be specified by calling **omp_set_schedule**, described in Section 3.2.12 on page 311.

Examples:

```
setenv OMP_SCHEDULE "guided,4"
setenv OMP_SCHEDULE "dynamic"
```

Cross References

- *run-sched-var* ICV, see Section 2.4 on page 49.
- Loop construct, see Section 2.10.1 on page 78.
- Parallel loop construct, see Section 2.16.1 on page 169.
- **omp_set_schedule** routine, see Section 3.2.12 on page 311.
- **omp_get_schedule** routine, see Section 3.2.13 on page 313.

5.2 OMP_NUM_THREADS

The **OMP_NUM_THREADS** environment variable sets the number of threads to use for **parallel** regions by setting the initial value of the *nthreads-var* ICV. See Section 2.4 on page 49 for a comprehensive set of rules about the interaction between the **OMP_NUM_THREADS** environment variable, the **num_threads** clause, the **omp_set_num_threads** library routine and dynamic adjustment of threads, and Section 2.8.1 on page 71 for a complete algorithm that describes how the number of threads for a **parallel** region is determined.

The value of this environment variable must be a list of positive integer values. The values of the list set the number of threads to use for **parallel** regions at the corresponding nested levels.

The behavior of the program is implementation defined if any value of the list specified in the **OMP_NUM_THREADS** environment variable leads to a number of threads which is greater than an implementation can support, or if any value is not a positive integer.

Example:

```
setenv OMP_NUM_THREADS 4,3,2
```

Cross References

- *nthreads-var* ICV, see Section 2.4 on page 49.
- **num_threads** clause, Section 2.8 on page 66.
- **omp_set_num_threads** routine, see Section 3.2.1 on page 300.
- **omp_get_num_threads** routine, see Section 3.2.2 on page 301.
- **omp_get_max_threads** routine, see Section 3.2.3 on page 302.
- **omp_get_team_size** routine, see Section 3.2.19 on page 320.

5.3 OMP_DYNAMIC

The **OMP_DYNAMIC** environment variable controls dynamic adjustment of the number of threads to use for executing **parallel** regions by setting the initial value of the *dyn-var* ICV. The value of this environment variable must be **true** or **false**. If the environment variable is set to **true**, the OpenMP implementation may adjust the number of threads to use for executing **parallel** regions in order to optimize the use of system resources. If the environment variable is set to **false**, the dynamic adjustment of the number of threads is disabled. The behavior of the program is implementation defined if the value of **OMP_DYNAMIC** is neither **true** nor **false**.

Example:

```
setenv OMP_DYNAMIC true
```

Cross References

- *dyn-var* ICV, see Section 2.4 on page 49.
- **omp_set_dynamic** routine, see Section 3.2.7 on page 306.
- **omp_get_dynamic** routine, see Section 3.2.8 on page 308.

5.4 OMP_PROC_BIND

The **OMP_PROC_BIND** environment variable sets the initial value of the *bind-var* ICV. The value of this environment variable is either **true**, **false**, or a comma separated list of **master**, **close**, or **spread**. The values of the list set the thread affinity policy to be used for parallel regions at the corresponding nested level.

If the environment variable is set to **false**, the execution environment may move OpenMP threads between OpenMP places, thread affinity is disabled, and **proc_bind** clauses on **parallel** constructs are ignored.

Otherwise, the execution environment should not move OpenMP threads between OpenMP places, thread affinity is enabled, and the initial thread is bound to the first place in the OpenMP place list prior to the first active parallel region.

The behavior of the program is implementation defined if the value in the **OMP_PROC_BIND** environment variable is not **true**, **false**, or a comma separated list of **master**, **close**, or **spread**. The behavior is also implementation defined if an initial thread cannot be bound to the first place in the OpenMP place list.

Examples:

```
setenv OMP_PROC_BIND false
setenv OMP_PROC_BIND "spread, spread, close"
```

Cross References

- *bind-var* ICV, see Section 2.4 on page 49.
- **proc_bind** clause, see Section 2.8.2 on page 73.
- **omp_get_proc_bind** routine, see Section 3.2.22 on page 323.

5.5 OMP_PLACES

A list of places can be specified in the **OMP_PLACES** environment variable. The *place-partition-var* ICV obtains its initial value from the **OMP_PLACES** value, and makes the list available to the execution environment. The value of **OMP_PLACES** can be one of two types of values: either an abstract name describing a set of places or an explicit list of places described by non-negative numbers.

The **OMP_PLACES** environment variable can be defined using an explicit ordered list of comma-separated places. A place is defined by an unordered set of comma-separated non-negative numbers enclosed by braces. The meaning of the numbers and how the numbering is done are implementation defined. Generally, the numbers represent the smallest unit of execution exposed by the execution environment, typically a hardware thread.

Intervals may also be used to define places. Intervals can be specified using the <*lower-bound*> : <*length*> : <*stride*> notation to represent the following list of numbers: "<*lower-bound*>, <*lower-bound*> + <*stride*>, ..., <*lower-bound*> + (<*length*>- 1)*<*stride*>." When <*stride*> is omitted, a unit stride is assumed. Intervals can specify numbers within a place as well as sequences of places.

An exclusion operator "**!**" can also be used to exclude the number or place immediately following the operator.

Alternatively, the abstract names listed in Table 5.1 should be understood by the execution and runtime environment. The precise definitions of the abstract names are implementation defined. An implementation may also add abstract names as appropriate for the target platform.

The abstract name may be appended by a positive number in parentheses to denote the length of the place list to be created, that is *abstract_name(num-places)*. When requesting fewer places than

available on the system, the determination of which resources of type *abstract_name* are to be included in the place list is implementation defined. When requesting more resources than available, the length of the place list is implementation defined.

TABLE 5.1: Defined Abstract Names for **OMP_PLACES**

Abstract Name	Meaning
threads	Each place corresponds to a single hardware thread on the target machine.
cores	Each place corresponds to a single core (having one or more hardware threads) on the target machine.
sockets	Each place corresponds to a single socket (consisting of one or more cores) on the target machine.

The behavior of the program is implementation defined when the execution environment cannot map a numerical value (either explicitly defined or implicitly derived from an interval) within the **OMP_PLACES** list to a processor on the target platform, or if it maps to an unavailable processor. The behavior is also implementation defined when the **OMP_PLACES** environment variable is defined using an abstract name.

The following grammar describes the values accepted for the **OMP_PLACES** environment variable.

⟨list⟩ ⊨ ⟨p-list⟩ | ⟨aname⟩
⟨p-list⟩ ⊨ ⟨p-interval⟩ | ⟨p-list⟩,⟨p-interval⟩
⟨p-interval⟩ ⊨ ⟨place⟩:⟨len⟩:⟨stride⟩ | ⟨place⟩:⟨len⟩ | ⟨place⟩ | !⟨place⟩
⟨place⟩ ⊨ {⟨res-list⟩}
⟨res-list⟩ ⊨ ⟨res-interval⟩ | ⟨res-list⟩,⟨res-interval⟩
⟨res-interval⟩ ⊨ ⟨res⟩:⟨num-places⟩:⟨stride⟩ | ⟨res⟩:⟨num-places⟩ | ⟨res⟩ | !⟨res⟩
⟨aname⟩ ⊨ ⟨word⟩(⟨num-places⟩) | ⟨word⟩
⟨word⟩ ⊨ sockets | cores | threads | <implementation-defined abstract name>
⟨res⟩ ⊨ *non-negative integer*
⟨num-places⟩ ⊨ *positive integer*
⟨stride⟩ ⊨ *integer*
⟨len⟩ ⊨ *positive integer*

Examples:

```
setenv OMP_PLACES threads
setenv OMP_PLACES "threads(4)"
setenv OMP_PLACES "{0,1,2,3},{4,5,6,7},{8,9,10,11},{12,13,14,15}"
setenv OMP_PLACES "{0:4},{4:4},{8:4},{12:4}"
setenv OMP_PLACES "{0:4}:4:4"
```

where each of the last three definitions corresponds to the same 4 places including the smallest units of execution exposed by the execution environment numbered, in turn, 0 to 3, 4 to 7, 8 to 11, and 12 to 15.

Cross References

- *place-partition-var*, Section 2.4 on page 49.
- Controlling OpenMP thread affinity, Section 2.8.2 on page 73.
- **omp_get_num_places** routine, see Section 3.2.23 on page 325.
- **omp_get_place_num_procs** routine, see Section 3.2.24 on page 326.
- **omp_get_place_proc_ids** routine, see Section 3.2.25 on page 327.
- **omp_get_place_num** routine, see Section 3.2.26 on page 328.
- **omp_get_partition_num_places** routine, see Section 3.2.27 on page 329.
- **omp_get_partition_place_nums** routine, see Section 3.2.28 on page 330.

5.6 OMP_NESTED

The deprecated **OMP_NESTED** environment variable controls nested parallelism by setting the initial value of the *nest-var* ICV. The value of this environment variable must be **true** or **false**. If the environment variable is set to **true**, nested parallelism is enabled; if set to **false**, nested parallelism is disabled. The behavior of the program is implementation defined if the value of **OMP_NESTED** is neither **true** nor **false**.

Example:

```
setenv OMP_NESTED false
```

Cross References

- *nest-var* ICV, see Section 2.4 on page 49.
- **omp_set_nested** routine, see Section 3.2.10 on page 309.
- **omp_get_team_size** routine, see Section 3.2.19 on page 320.

5.7 OMP_STACKSIZE

The **OMP_STACKSIZE** environment variable controls the size of the stack for threads created by the OpenMP implementation, by setting the value of the *stacksize-var* ICV. The environment variable does not control the size of the stack for an initial thread.

The value of this environment variable takes the form:

size | *size***B** | *size***K** | *size***M** | *size***G**

where:

- *size* is a positive integer that specifies the size of the stack for threads that are created by the OpenMP implementation.
- **B**, **K**, **M**, and **G** are letters that specify whether the given size is in Bytes, Kilobytes (1024 Bytes), Megabytes (1024 Kilobytes), or Gigabytes (1024 Megabytes), respectively. If one of these letters is present, there may be white space between *size* and the letter.

If only *size* is specified and none of **B**, **K**, **M**, or **G** is specified, then *size* is assumed to be in Kilobytes.

The behavior of the program is implementation defined if **OMP_STACKSIZE** does not conform to the above format, or if the implementation cannot provide a stack with the requested size.

Examples:

```
setenv OMP_STACKSIZE 2000500B
setenv OMP_STACKSIZE "3000 k "
setenv OMP_STACKSIZE 10M
setenv OMP_STACKSIZE " 10 M "
setenv OMP_STACKSIZE "20 m "
setenv OMP_STACKSIZE " 1G"
setenv OMP_STACKSIZE 20000
```

Cross References

- *stacksize-var* ICV, see Section 2.4 on page 49.

5.8 OMP_WAIT_POLICY

The **OMP_WAIT_POLICY** environment variable provides a hint to an OpenMP implementation about the desired behavior of waiting threads by setting the *wait-policy-var* ICV. A compliant OpenMP implementation may or may not abide by the setting of the environment variable.

The value of this environment variable takes the form:

ACTIVE | PASSIVE

The **ACTIVE** value specifies that waiting threads should mostly be active, consuming processor cycles, while waiting. An OpenMP implementation may, for example, make waiting threads spin.

The **PASSIVE** value specifies that waiting threads should mostly be passive, not consuming processor cycles, while waiting. For example, an OpenMP implementation may make waiting threads yield the processor to other threads or go to sleep.

The details of the **ACTIVE** and **PASSIVE** behaviors are implementation defined.

Examples:

```
setenv OMP_WAIT_POLICY ACTIVE
setenv OMP_WAIT_POLICY active
setenv OMP_WAIT_POLICY PASSIVE
setenv OMP_WAIT_POLICY passive
```

Cross References

- *wait-policy-var* ICV, see Section 2.4 on page 49.

5.9 OMP_MAX_ACTIVE_LEVELS

The **OMP_MAX_ACTIVE_LEVELS** environment variable controls the maximum number of nested active **parallel** regions by setting the initial value of the *max-active-levels-var* ICV.

The value of this environment variable must be a non-negative integer. The behavior of the program is implementation defined if the requested value of **OMP_MAX_ACTIVE_LEVELS** is greater than the maximum number of nested active parallel levels an implementation can support, or if the value is not a non-negative integer.

Cross References

- *max-active-levels-var* ICV, see Section 2.4 on page 49.
- **omp_set_max_active_levels** routine, see Section 3.2.15 on page 315.
- **omp_get_max_active_levels** routine, see Section 3.2.16 on page 317.

5.10 OMP_THREAD_LIMIT

The **OMP_THREAD_LIMIT** environment variable sets the maximum number of OpenMP threads to use in a contention group by setting the *thread-limit-var* ICV.

The value of this environment variable must be a positive integer. The behavior of the program is implementation defined if the requested value of **OMP_THREAD_LIMIT** is greater than the number of threads an implementation can support, or if the value is not a positive integer.

Cross References

- *thread-limit-var* ICV, see Section 2.4 on page 49.
- **omp_get_thread_limit** routine, see Section 3.2.14 on page 314.

5.11 OMP_CANCELLATION

The **OMP_CANCELLATION** environment variable sets the initial value of the *cancel-var* ICV.

The value of this environment variable must be **true** or **false**. If set to **true**, the effects of the **cancel** construct and of cancellation points are enabled and cancellation is activated. If set to **false**, cancellation is disabled and the **cancel** construct and cancellation points are effectively ignored.

Cross References

- *cancel-var*, see Section 2.4.1 on page 49.
- **cancel** construct, see Section 2.19.1 on page 232.
- **cancellation point** construct, see Section 2.19.2 on page 237.
- **omp_get_cancellation** routine, see Section 3.2.9 on page 308.

5.12 OMP_DISPLAY_ENV

The **OMP_DISPLAY_ENV** environment variable instructs the runtime to display the OpenMP version number and the value of the ICVs associated with the environment variables described in Chapter 5, as *name* = *value* pairs. The runtime displays this information once, after processing the environment variables and before any user calls to change the ICV values by runtime routines defined in Chapter 3.

The value of the **OMP_DISPLAY_ENV** environment variable may be set to one of these values:

TRUE | FALSE | VERBOSE

The **TRUE** value instructs the runtime to display the OpenMP version number defined by the **_OPENMP** version macro (or the **openmp_version** Fortran parameter) value and the initial ICV values for the environment variables listed in Chapter 5. The **VERBOSE** value indicates that the runtime may also display the values of runtime variables that may be modified by vendor-specific environment variables. The runtime does not display any information when the **OMP_DISPLAY_ENV** environment variable is **FALSE** or undefined. For all values of the environment variable other than **TRUE**, **FALSE**, and **VERBOSE**, the displayed information is unspecified.

The display begins with `"OPENMP DISPLAY ENVIRONMENT BEGIN"`, followed by the **_OPENMP** version macro (or the **openmp_version** Fortran parameter) value and ICV values, in the format *NAME* '=' *VALUE*. *NAME* corresponds to the macro or environment variable name, optionally prepended by a bracketed *device-type*. *VALUE* corresponds to the value of the macro or ICV associated with this environment variable. Values should be enclosed in single quotes. The display is terminated with `"OPENMP DISPLAY ENVIRONMENT END"`.

Example:

```
% setenv OMP_DISPLAY_ENV TRUE
```

The above example causes an OpenMP implementation to generate output of the following form:

```
OPENMP DISPLAY ENVIRONMENT BEGIN
  _OPENMP='201711'
  [host] OMP_SCHEDULE='GUIDED,4'
  [host] OMP_NUM_THREADS='4,3,2'
  [device] OMP_NUM_THREADS='2'
  [host,device] OMP_DYNAMIC='TRUE'
  [host] OMP_PLACES='0:4,4:4,8:4,12:4'
  ...
OPENMP DISPLAY ENVIRONMENT END
```

5.13 OMP_DISPLAY_AFFINITY

The **OMP_DISPLAY_AFFINITY** environment variable instructs the runtime to display formatted affinity information for OpenMP threads upon entering the first parallel region and when there is any change in thread affinity thereafter. If there is a change of affinity of any thread in a parallel region, thread affinity information for all threads in that region will be displayed. There is no specific order in displaying thread affinity information for all threads in the same parallel region.

The value of the **OMP_DISPLAY_AFFINITY** environment variable may be set to one of these values:

TRUE | FALSE

The **TRUE** value instructs the runtime to display the OpenMP thread affinity information, and uses the format setting defined in the *affinity-format-var* ICV.

The runtime does not display the OpenMP thread affinity information when the value of the **OMP_DISPLAY_AFFINITY** environment variable is **FALSE** or undefined. For all values of the environment variable other than **TRUE** or **FALSE**, the display action is implementation defined.

Example:

```
setenv OMP_DISPLAY_AFFINITY TRUE
```

The above example causes an OpenMP implementation to display OpenMP thread affinity information during execution of the program, in a format given by the *affinity-format-var* ICV. The following is a sample output:

```
thread_level=   1,   thread_id=   0,   thread_affinity=    0,1
thread_level=   1,   thread_id=   1,   thread_affinity=    2,3
```

Cross References

- Controlling OpenMP thread affinity, see Section 2.8.2 on page 73.
- **OMP_AFFINITY_FORMAT** environment variable, see Section 5.14 on page 575.
- **omp_set_affinity_format** routine, see Section 3.2.29 on page 331.
- **omp_get_affinity_format** routine, see Section 3.2.30 on page 332.
- **omp_display_affinity** routine, see Section 3.2.31 on page 333.
- **omp_capture_affinity** routine, see Section 3.2.32 on page 334.

5.14 OMP_AFFINITY_FORMAT

The **OMP_AFFINITY_FORMAT** environment variable sets the initial value of the *affinity-format-var* ICV which defines the format when displaying OpenMP thread affinity information.

The value of this environment variable is a character string that may contain as substrings one or more field specifiers, in addition to other characters. The format of each field specifier is "%[0[.]*size*]*type*", where an individual field specifier must contain the percent symbol (%) and a type. The type can be a single character short name or its corresponding long name delimited with curly braces, such as "%n" or "%{thread_num}". A literal percent is specified as "%%". Field specifiers can be provided in any order.

The "0" indicates whether or not to add leading zeros to the output. The "." indicates the output should be right justified when *size* is specified. By default, output is left justified. The minimum field length is *size*. If no *size* is specified, the actual length needed to print the field will be used.

Any other characters in the format string that are not part of a field specifier will be included literally in the output.

Available field types are:

TABLE 5.2: Available Field Types for Formatting OpenMP Thread Affinity Information

Short Name	Long Name	Meaning
L	**thread_level**	The value returned by **omp_get_level()**.
n	**thread_num**	The value returned by **omp_get_thread_num()**.
h	**host**	The name for the host machine on which the OpenMP program is running.
P	**process_id**	The process identifier used by the implementation.
T	**thread_identifier**	The thread identifier for a native thread defined by the implementation.
N	**num_threads**	The value returned by **omp_get_num_threads()**.
A	**ancestor_tnum**	The value returned by **omp_get_ancestor_thread_num(***level***)**, where *level* is **omp_get_level()** minus 1.
a	**thread_affinity**	The list of numerical identifiers, in the format of a comma-separated list of integers or integer ranges, representing processors on which a thread may execute, subject to OpenMP thread affinity control and/or other external affinity mechanisms.

1 Implementations may define additional field types. If an implementation does not have information
2 for a field type, "undefined" is printed for this field when displaying the OpenMP thread affinity
3 information.

4 Example:

```
setenv OMP_AFFINITY_FORMAT
       "Thread Affinity: %0.3L %.8n %.15{thread_affinity} %.12h"
```

5 The above example causes an OpenMP implementation to display OpenMP thread affinity
6 information in the following form:

```
Thread Affinity: 001        0       0-1,16-17      nid003
Thread Affinity: 001        1       2-3,18-19      nid003
```

Cross References

- Controlling OpenMP thread affinity, see Section 2.8.2 on page 73.
- **OMP_DISPLAY_AFFINITY** environment variable, see Section 5.13 on page 574.
- **omp_set_affinity_format** routine, see Section 3.2.29 on page 331.
- **omp_get_affinity_format** routine, see Section 3.2.30 on page 332.
- **omp_display_affinity** routine, see Section 3.2.31 on page 333.
- **omp_capture_affinity** routine, see Section 3.2.32 on page 334.

5.15 OMP_DEFAULT_DEVICE

The **OMP_DEFAULT_DEVICE** environment variable sets the device number to use in device constructs by setting the initial value of the *default-device-var* ICV.

The value of this environment variable must be a non-negative integer value.

Cross References

- *default-device-var* ICV, see Section 2.4 on page 49.
- device constructs, Section 2.15 on page 131.

5.16 OMP_MAX_TASK_PRIORITY

The **OMP_MAX_TASK_PRIORITY** environment variable controls the use of task priorities by setting the initial value of the *max-task-priority-var* ICV. The value of this environment variable must be a non-negative integer.

Example:

```
% setenv OMP_MAX_TASK_PRIORITY 20
```

Cross References

- *max-task-priority-var* ICV, see Section 2.4 on page 49.
- Tasking Constructs, see Section 2.13 on page 110.
- **omp_get_max_task_priority** routine, see Section 3.2.41 on page 343.

5.17 OMP_TARGET_OFFLOAD

The **OMP_TARGET_OFFLOAD** environment variable sets the initial value of the *target-offload-var* ICV. The value of the **OMP_TARGET_OFFLOAD** environment variable may be set to one of these values:

MANDATORY | DISABLED | DEFAULT

The **MANDATORY** value specifies that a device construct or a device memory routine must execute on a target device. If the device construct cannot execute on its target device, or if a device memory routine fails to execute, a warning is issued and the program execution aborts. Device constructs are exempt from this behavior when an if-clause is present and the if-clause expression evaluates to false.

The support of **DISABLED** is implementation defined. If an implementation supports it, the behavior should be that a device construct must execute on the host. The behavior with this environment value is equivalent to an if clause present on all device constructs, where each of these if clause expressions evaluate to false. Device memory routines behave as if all device number parameters are set to the value returned by **omp_get_initial_device()**. The **omp_get_initial_device()** routine returns that no target device is available

The **DEFAULT** value specifies that when one or more target devices are available, the runtime behaves as if this environment variable is set to **MANDATORY**; otherwise, the runtime behaves as if this environment variable is set to **DISABLED**.

Example:

```
% setenv OMP_TARGET_OFFLOAD MANDATORY
```

Cross References

- *target-offload-icv* ICV, see Section 2.4 on page 49.
- device constructs, Section 2.15 on page 131.

5.18 OMP_TOOL

The **OMP_TOOL** environment variable sets the *tool-var* ICV which controls whether an OpenMP runtime will try to register a first party tool. The value of this environment variable must be **enabled** or **disabled**. If **OMP_TOOL** is set to any value other than **enabled** or **disabled**, the behavior is unspecified. If **OMP_TOOL** is not defined, the default value for *tool-var* is **enabled**.

Example:

```
% setenv OMP_TOOL enabled
```

Cross References

- *tool-var* ICV, see Section 2.4 on page 49.
- Tool Interface, see Section 4 on page 377.

5.19 OMP_TOOL_LIBRARIES

The **OMP_TOOL_LIBRARIES** environment variable sets the *tool-libraries-var* ICV to a list of tool libraries that will be considered for use on a device where an OpenMP implementation is being initialized. The value of this environment variable must be a colon-separated list of dynamically-linked libraries, each specified by an absolute path.

If the *tool-var* ICV is not enabled, the value of *tool-libraries-var* will be ignored. Otherwise, if **ompt_start_tool**, a global function symbol for a tool initializer, isn't visible in the address space on a device where OpenMP is being initialized or if **ompt_start_tool** returns **NULL**, an OpenMP implementation will consider libraries in the *tool-libraries-var* list in a left to right order. The OpenMP implementation will search the list for a library that meets two criteria: it can be dynamically loaded on the current device and it defines the symbol **ompt_start_tool**. If an OpenMP implementation finds a suitable library, no further libraries in the list will be considered.

Cross References

- *tool-libraries-var* ICV, see Section 2.4 on page 49.
- Tool Interface, see Section 4 on page 377.
- **ompt_start_tool** routine, see Section 4.3.2.1 on page 558.

1
5.20 OMPD_ENABLED

2 The **OMPD_ENABLED** environment variable sets the *ompd-tool-var* ICV which controls whether an
3 OpenMP runtime will collect information that an OMPD plugin may need to support a tool. The
4 value of this environment variable must be **enabled** or **disabled**. If **OMPD_ENABLED** is set to
5 any value other than **enabled** or **disabled**, the behavior is unspecified. If **OMPD_ENABLED** is
6 not defined, the default value for *ompd-tool-var* is **disabled**.

7 Example:

```
% setenv OMPD_ENABLED enabled
```

8
Cross References

9 • *ompd-tool-var* ICV, see Section 2.4 on page 49.

10 • **ompd_enable**, see Section 4.2.6.1 on page 550

11 • Tool Interface, see Section 4 on page 377

12 • Enabling the Runtime for OMPD, see Section 4.2.1.1 on page 478.

13
5.21 OMP_ALLOCATOR

14 **OMP_ALLOCATOR** sets the *def-allocator-var* ICV that specifies the default allocator for allocation
15 calls, directives and clauses that do not specify an allocator. The value of this environment variable
16 is a predefined allocator from Table 2.5 on page 64. The value of this environment variable is not
17 case sensitive.

18
Cross References

19 • *def-allocator-var* ICV, see Section 2.4 on page 49.

20 • Memory Allocators, see Section 2.7 on page 64.

21 • **omp_set_default_allocator** routine, see Section 3.6.2 on page 369.

22 • **omp_get_default_allocator** routine, see Section 3.6.3 on page 370.

APPENDIX A

Stubs for Runtime Library Routines

This section provides stubs for the runtime library routines defined in the OpenMP API. The stubs are provided to enable portability to platforms that do not support the OpenMP API. On these platforms, OpenMP programs must be linked with a library containing these stub routines. The stub routines assume that the directives in the OpenMP program are ignored. As such, they emulate serial semantics executing on the host.

Note that the lock variable that appears in the lock routines must be accessed exclusively through these routines. It should not be initialized or otherwise modified in the user program.

In an actual implementation the lock variable might be used to hold the address of an allocated memory block, but here it is used to hold an integer value. Users should not make assumptions about mechanisms used by OpenMP implementations to implement locks based on the scheme used by the stub procedures.

Fortran

In order to be able to compile the Fortran stubs file, the include file **omp_lib.h** was split into two files: **omp_lib_kinds.h** and **omp_lib.h** and the **omp_lib_kinds.h** file included where needed. There is no requirement for the implementation to provide separate files.

Fortran

A.1 C/C++ Stub Routines

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

void omp_set_num_threads(int num_threads)
{
}

int omp_get_num_threads(void)
{
    return 1;
}

int omp_get_max_threads(void)
{
    return 1;
}

int omp_get_thread_num(void)
{
    return 0;
}

int omp_get_num_procs(void)
{
    return 1;
}

int omp_in_parallel(void)
{
    return 0;
}

void omp_set_dynamic(int dynamic_threads)
{
}

int omp_get_dynamic(void)
{
    return 0;
}

int omp_get_cancellation(void)
{
    return 0;
```

```
}

void omp_set_nested(int nested)
{
}

int omp_get_nested(void)
{
    return 0;
}

void omp_set_schedule(omp_sched_t kind, int chunk_size)
{
}

void omp_get_schedule(omp_sched_t *kind, int *chunk_size)
{
    *kind = omp_sched_static;
    *chunk_size = 0;
}

int omp_get_thread_limit(void)
{
    return 1;
}

void omp_set_max_active_levels(int max_active_levels)
{
}

int omp_get_max_active_levels(void)
{
    return 0;
}

int omp_get_level(void)
{
    return 0;
}

int omp_get_ancestor_thread_num(int level)
{
    if (level == 0)
    {
        return 0;
    }
    else
```

```
        {
            return -1;
        }
    }

    int omp_get_team_size(int level)
    {
        if (level == 0)
        {
            return 1;
        }
        else
        {
            return -1;
        }
    }

    int omp_get_active_level(void)
    {
        return 0;
    }

    int omp_in_final(void)
    {
        return 1;
    }

    omp_proc_bind_t omp_get_proc_bind(void)
    {
        return omp_proc_bind_false;
    }

    int omp_get_num_places(void)
    {
        return 0;
    }

    int omp_get_place_num_procs(int place_num)
    {
        return 0;
    }

    void omp_get_place_proc_ids(int place_num, int *ids)
    {
    }

    int omp_get_place_num(void)
```

```
{
    return -1;
}

int omp_get_partition_num_places(void)
{
    return 0;
}

void omp_get_partition_place_nums(int *place_nums)
{
}

void omp_set_affinity_format(char const *format)
{
}

size_t omp_get_affinity_format(char* buffer, size_t size)
{
    return 0;
}

void omp_display_affinity(char const *format)
{
}

size_t omp_capture_affinity(char *buffer, size_t size, char const *format)
{
    return 0;
}

void omp_set_default_device(int device_num)
{
}

int omp_get_default_device(void)
{
    return 0;
}

int omp_get_num_devices(void)
{
    return 0;
}

int omp_get_device_num(void)
{
```

```
        return -10;
    }

    int omp_get_num_teams(void)
    {
        return 1;
    }

    int omp_get_team_num(void)
    {
        return 0;
    }

    int omp_is_initial_device(void)
    {
        return 1;
    }

    int omp_get_initial_device(void)
    {
        return -10;
    }

    int omp_get_max_task_priority(void)
    {
        return 0;
    }

    struct __omp_lock
    {
        int lock;
    };

    enum { UNLOCKED = -1, INIT, LOCKED };

    void omp_init_lock(omp_lock_t *arg)
    {
        struct __omp_lock *lock = (struct __omp_lock *)arg;
        lock->lock = UNLOCKED;
    }

    void omp_init_lock_with_hint(omp_lock_t *arg, omp_lock_hint_t hint)
    {
        omp_init_lock(arg);
    }

    void omp_destroy_lock(omp_lock_t *arg)
```

```
{
    struct __omp_lock *lock = (struct __omp_lock *)arg;
    lock->lock = INIT;
}

void omp_set_lock(omp_lock_t *arg)
{
    struct __omp_lock *lock = (struct __omp_lock *)arg;
    if (lock->lock == UNLOCKED)
    {
        lock->lock = LOCKED;
    }
    else if (lock->lock == LOCKED)
    {
        fprintf(stderr, "error: deadlock in using lock variable\n");
        exit(1);
    }

    else
    {
        fprintf(stderr, "error: lock not initialized\n");
        exit(1);
    }
}

void omp_unset_lock(omp_lock_t *arg)
{
    struct __omp_lock *lock = (struct __omp_lock *)arg;
    if (lock->lock == LOCKED)
    {
        lock->lock = UNLOCKED;
    }
    else if (lock->lock == UNLOCKED)
    {
        fprintf(stderr, "error: lock not set\n");
        exit(1);
    }
    else
    {
        fprintf(stderr, "error: lock not initialized\n");
        exit(1);
    }
}

int omp_test_lock(omp_lock_t *arg)
{
    struct __omp_lock *lock = (struct __omp_lock *)arg;
```

```
        if (lock->lock == UNLOCKED)
        {
            lock->lock = LOCKED;
            return 1;
        }
        else if (lock->lock == LOCKED)
        {
            return 0;
        }
        else
        {
            fprintf(stderr, "error: lock not initialized\n");
            exit(1);
        }
    }

    struct __omp_nest_lock
    {
        short owner;
        short count;
    };

    enum { NOOWNER = -1, MASTER = 0 };

    void omp_init_nest_lock(omp_nest_lock_t *arg)
    {
        struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
        nlock->owner = NOOWNER;
        nlock->count = 0;
    }

    void omp_init_nest_lock_with_hint(omp_nest_lock_t *arg,
                                      omp_lock_hint_t hint)
    {
        omp_init_nest_lock(arg);
    }

    void omp_destroy_nest_lock(omp_nest_lock_t *arg)
    {
        struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
        nlock->owner = NOOWNER;
        nlock->count = UNLOCKED;
    }

    void omp_set_nest_lock(omp_nest_lock_t *arg)
    {
        struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
```

```
1         if (nlock->owner == MASTER && nlock->count >= 1)
2         {
3             nlock->count++;
4         }
5         else if (nlock->owner == NOOWNER && nlock->count == 0)
6         {
7             nlock->owner = MASTER;
8             nlock->count = 1;
9         }
10        else
11        {
12            fprintf(stderr, "error: lock corrupted or not initialized\n");
13            exit(1);
14        }
15    }
16
17    void omp_unset_nest_lock(omp_nest_lock_t *arg)
18    {
19        struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
20        if (nlock->owner == MASTER && nlock->count >= 1)
21        {
22            nlock->count--;
23            if (nlock->count == 0)
24            {
25                nlock->owner = NOOWNER;
26            }
27        }
28        else if (nlock->owner == NOOWNER && nlock->count == 0)
29        {
30            fprintf(stderr, "error: lock not set\n");
31            exit(1);
32        }
33        else
34        {
35            fprintf(stderr, "error: lock corrupted or not initialized\n");
36            exit(1);
37        }
38    }
39
40    int omp_test_nest_lock(omp_nest_lock_t *arg)
41    {
42        struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
43        omp_set_nest_lock(arg);
44        return nlock->count;
45    }
46
47    double omp_get_wtime(void)
```

```
{
/* This function does not provide a working
 * wallclock timer. Replace it with a version
 * customized for the target machine.
 */
    return 0.0;
}

double omp_get_wtick(void)
{
/* This function does not provide a working
 * clock tick function. Replace it with
 * a version customized for the target machine.
 */
    return 365. * 86400.;
}

void * omp_target_alloc(size_t size, int device_num)
{
    if (device_num != -10)
      return NULL;
    return malloc(size)
}

void omp_target_free(void *device_ptr, int device_num)
{
    free(device_ptr);
}

int omp_target_is_present(void *ptr, int device_num)
{
    return 1;
}

int omp_target_memcpy(void *dst, void *src, size_t length,
                      size_t dst_offset, size_t src_offset,
                      int dst_device, int src_device)
{
    // only the default device is valid in a stub
    if (dst_device != -10 || src_device != -10
            || ! dst || ! src )
        return EINVAL;
    memcpy((char *)dst + dst_offset,
           (char *)src + src_offset,
           length);
    return 0;
}
```

```
int omp_target_memcpy_rect(
    void *dst, void *src,
    size_t element_size,
    int num_dims,
    const size_t *volume,
    const size_t *dst_offsets,
    const size_t *src_offsets,
    const size_t *dst_dimensions,
    const size_t *src_dimensions,
    int dst_device_num, int src_device_num)
{
    int ret=0;
    // Both null, return number of dimensions supported,
    // this stub supports an arbitrary number
    if (dst == NULL && src == NULL) return INT_MAX;

    if (!volume || !dst_offsets || !src_offsets
            || !dst_dimensions || !src_dimensions
            || num_dims < 1 ) {
        ret = EINVAL;
        goto done;
    }
    if (num_dims == 1) {
        ret = omp_target_memcpy(dst, src,
                                element_size * volume[0],
                                dst_offsets[0] * element_size,
                                src_offsets[0] * element_size,
                                dst_device_num, src_device_num);
        if(ret) goto done;
    } else {
        size_t dst_slice_size = element_size;
        size_t src_slice_size = element_size;
        for (int i=1; i < num_dims; i++) {
            dst_slice_size *= dst_dimensions[i];
            src_slice_size *= src_dimensions[i];
        }
        size_t dst_off = dst_offsets[0] * dst_slice_size;
        size_t src_off = src_offsets[0] * src_slice_size;
        for (size_t i=0; i < volume[0]; i++) {
            ret = omp_target_memcpy_rect(
                        (char *)dst + dst_off + dst_slice_size*i,
                        (char *)src + src_off + src_slice_size*i,
                        element_size,
                        num_dims - 1,
                        volume + 1,
                        dst_offsets + 1,
```

```
                              src_offsets + 1,
                              dst_dimensions + 1,
                              src_dimensions + 1,
                              dst_device_num,
                              src_device_num);
                if (ret) goto done;
            }
        }
    done:
        return ret;
    }

    int omp_target_associate_ptr(void *host_ptr, void *device_ptr,
                                 size_t size, size_t device_offset,
                                 int device_num)
    {
        // No association is possible because all host pointers
        // are considered present
        return EINVAL;
    }

    int omp_target_disassociate_ptr(void *ptr, int device_num)
    {
        return EINVAL;
    }


    int omp_control_tool(int command, int modifier, void *arg)
    {
        return omp_control_tool_notool;
    }

    static omp_allocator_t * omp_allocator = OMP_NULL_ALLOCATOR;

    omp_allocator_t * omp_default_mem_alloc;
    omp_allocator_t * omp_large_cap_mem_alloc;
    omp_allocator_t * omp_const_mem_alloc;
    omp_allocator_t * omp_high_bw_mem_alloc;
    omp_allocator_t * omp_low_lat_mem_alloc;
    omp_allocator_t * omp_cgroup_mem_alloc;
    omp_allocator_t * omp_pteam_mem_alloc;
    omp_allocator_t * omp_thread_mem_alloc;

    void omp_set_default allocator(omp_allocator_t *allocator)
    {
         omp_allocator = allocator;
    }
```

```
omp_allocator_t * omp_get_default_allocator (void)
{
    return omp_allocator;
}

#ifdef __cplusplus
void * omp_alloc (size_t size, omp_allocator_t *allocator = OMP_NULL_ALLOCATOR)
#else
void * omp_alloc (size_t size, omp_allocator_t *allocator)
#endif
{
    return malloc(size);
}

#ifdef __cplusplus
void omp_free (void * ptr, omp_allocator_t *allocator = OMP_NULL_ALLOCATOR)
#else
void omp_free (void * ptr, omp_allocator_t *allocator)
#endif
{
    free(ptr);
}
```

A.2 Fortran Stub Routines

```
subroutine omp_set_num_threads(num_threads)
  integer num_threads
  return
end subroutine

integer function omp_get_num_threads()
  omp_get_num_threads = 1
  return
end function

integer function omp_get_max_threads()
  omp_get_max_threads = 1
  return
end function

integer function omp_get_thread_num()
  omp_get_thread_num = 0
  return
end function

integer function omp_get_num_procs()
  omp_get_num_procs = 1
  return
end function

logical function omp_in_parallel()
  omp_in_parallel = .false.
  return
end function

subroutine omp_set_dynamic(dynamic_threads)
  logical dynamic_threads
  return
end subroutine

logical function omp_get_dynamic()
  omp_get_dynamic = .false.
  return
end function

logical function omp_get_cancellation()
  omp_get_cancellation = .false.
  return
end function
```

```
subroutine omp_set_nested(nested)
  logical nested
  return
end subroutine

logical function omp_get_nested()
  omp_get_nested = .false.
  return
end function

subroutine omp_set_schedule(kind, chunk_size)
  include 'omp_lib_kinds.h'
  integer (kind=omp_sched_kind) kind
  integer chunk_size
  return
end subroutine

subroutine omp_get_schedule(kind, chunk_size)
  include 'omp_lib_kinds.h'
  integer (kind=omp_sched_kind) kind
  integer chunk_size
  kind = omp_sched_static
  chunk_size = 0
  return
end subroutine

integer function omp_get_thread_limit()
  omp_get_thread_limit = 1
  return
end function

subroutine omp_set_max_active_levels(max_level)
  integer max_level
end subroutine

integer function omp_get_max_active_levels()
  omp_get_max_active_levels = 0
  return
end function

integer function omp_get_level()
  omp_get_level = 0
  return
end function

integer function omp_get_ancestor_thread_num(level)
  integer level
```

```
1           if ( level .eq. 0 ) then
2              omp_get_ancestor_thread_num = 0
3           else
4              omp_get_ancestor_thread_num = -1
5           end if
6           return
7         end function
8
9         integer function omp_get_team_size(level)
10          integer level
11          if ( level .eq. 0 ) then
12             omp_get_team_size = 1
13          else
14             omp_get_team_size = -1
15          end if
16          return
17        end function
18
19        integer function omp_get_active_level()
20          omp_get_active_level = 0
21          return
22        end function
23
24        logical function omp_in_final()
25          omp_in_final = .true.
26          return
27        end function
28
29        function omp_get_proc_bind()
30          include 'omp_lib_kinds.h'
31          integer (kind=omp_proc_bind_kind) omp_get_proc_bind
32          omp_get_proc_bind = omp_proc_bind_false
33        end function
34
35        integer function omp_get_num_places()
36          return 0
37        end function
38
39        integer function omp_get_place_num_procs(place_num)
40          integer place_num
41          return 0
42        end function
43
44        subroutine omp_get_place_proc_ids(place_num, ids)
45          integer place_num
46          integer ids(*)
47          return
```

```
end subroutine

integer function omp_get_place_num()
  return -1
end function

integer function omp_get_partition_num_places()
  return 0
end function

subroutine omp_get_partition_place_nums(place_nums)
  integer place_nums(*)
  return
end subroutine


subroutine omp_set_affinity_format(format)
   character(len=*),intent(in)::format
   return
end subroutine

integer function omp_get_affinity_format(buffer)
   character(len=*),intent(out)::buffer
   return 0
end function

subroutine omp_display_affinity(format)
   character(len=*),intent(in)::format
   return
end subroutine

integer function omp_capture_affinity(buffer,format)
   character(len=*),intent(out)::buffer
   character(len=*),intent(in)::format
   return 0
end function

subroutine omp_set_default_device(device_num)
  integer device_num
  return
end subroutine

integer function omp_get_default_device()
  omp_get_default_device = 0
  return
end function

```

```
integer function omp_get_num_devices()
  omp_get_num_devices = 0
  return
end function

integer function omp_get_device_num()
  omp_get_device_num = -10
  return
end function

integer function omp_get_num_teams()
  omp_get_num_teams = 1
  return
end function

integer function omp_get_team_num()
  omp_get_team_num = 0
  return
end function

logical function omp_is_initial_device()
  omp_is_initial_device = .true.
  return
end function

integer function omp_get_initial_device()
  omp_get_initial_device = -10
  return
end function

integer function omp_get_max_task_priority()
  omp_get_max_task_priority = 0
  return
end function

subroutine omp_init_lock(lock)
  ! lock is 0 if the simple lock is not initialized
  !        -1 if the simple lock is initialized but not set
  !         1 if the simple lock is set
  include 'omp_lib_kinds.h'
  integer(kind=omp_lock_kind) lock

  lock = -1
  return
end subroutine

subroutine omp_init_lock_with_hint(lock, hint)
```

```
      include 'omp_lib_kinds.h'
      integer(kind=omp_lock_kind) lock
      integer(kind=omp_lock_hint_kind) hint

      call omp_init_lock(lock)
      return
    end subroutine

    subroutine omp_destroy_lock(lock)
      include 'omp_lib_kinds.h'
      integer(kind=omp_lock_kind) lock

      lock = 0
      return
    end subroutine

    subroutine omp_set_lock(lock)
      include 'omp_lib_kinds.h'
      integer(kind=omp_lock_kind) lock

      if (lock .eq. -1) then
        lock = 1
      elseif (lock .eq. 1) then
        print *, 'error: deadlock in using lock variable'
        stop
      else
        print *, 'error: lock not initialized'
        stop
      endif
      return
    end subroutine

    subroutine omp_unset_lock(lock)
      include 'omp_lib_kinds.h'
      integer(kind=omp_lock_kind) lock

      if (lock .eq. 1) then
        lock = -1
      elseif (lock .eq. -1) then
        print *, 'error: lock not set'
        stop
      else
        print *, 'error: lock not initialized'
        stop
      endif
      return
    end subroutine
```

```
logical function omp_test_lock(lock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_lock_kind) lock

  if (lock .eq. -1) then
    lock = 1
    omp_test_lock = .true.
  elseif (lock .eq. 1) then
    omp_test_lock = .false.
  else
    print *, 'error: lock not initialized'
    stop
  endif

  return
end function

subroutine omp_init_nest_lock(nlock)
  ! nlock is
  ! 0 if the nestable lock is not initialized
  ! -1 if the nestable lock is initialized but not set
  ! 1 if the nestable lock is set
  ! no use count is maintained
  include 'omp_lib_kinds.h'
  integer(kind=omp_nest_lock_kind) nlock

  nlock = -1

  return
end subroutine

subroutine omp_init_nest_lock_with_hint(nlock, hint)
  include 'omp_lib_kinds.h'
  integer(kind=omp_nest_lock_kind) nlock
  integer(kind=omp_lock_hint_kind) hint

  call omp_init_nest_lock(nlock)
  return
end subroutine

subroutine omp_destroy_nest_lock(nlock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_nest_lock_kind) nlock

  nlock = 0
```

```
    return
  end subroutine

  subroutine omp_set_nest_lock(nlock)
    include 'omp_lib_kinds.h'
    integer(kind=omp_nest_lock_kind) nlock

    if (nlock .eq. -1) then
      nlock = 1
    elseif (nlock .eq. 0) then
      print *, 'error: nested lock not initialized'
      stop
    else
      print *, 'error: deadlock using nested lock variable'
      stop
    endif

    return
  end subroutine

  subroutine omp_unset_nest_lock(nlock)
    include 'omp_lib_kinds.h'
    integer(kind=omp_nest_lock_kind) nlock

    if (nlock .eq. 1) then
      nlock = -1
    elseif (nlock .eq. 0) then
      print *, 'error: nested lock not initialized'
      stop
    else
      print *, 'error: nested lock not set'
      stop
    endif

    return
  end subroutine

  integer function omp_test_nest_lock(nlock)
    include 'omp_lib_kinds.h'
    integer(kind=omp_nest_lock_kind) nlock

    if (nlock .eq. -1) then
      nlock = 1
      omp_test_nest_lock = 1
    elseif (nlock .eq. 1) then
      omp_test_nest_lock = 0
    else
```

```
          print *, 'error: nested lock not initialized'
          stop
        endif

        return
      end function

      double precision function omp_get_wtime()
        ! this function does not provide a working
        ! wall clock timer. replace it with a version
        ! customized for the target machine.

        omp_get_wtime = 0.0d0

        return
      end function

      double precision function omp_get_wtick()
        ! this function does not provide a working
        ! clock tick function. replace it with
        ! a version customized for the target machine.
        double precision one_year
        parameter (one_year=365.d0*86400.d0)

        omp_get_wtick = one_year

        return
      end function

      int function omp_control_tool(command, modifier)
        include 'omp_lib_kinds.h'
        integer (kind=omp_control_tool_kind) command
        integer (kind=omp_control_tool_kind) modifier

        return omp_control_tool_notool
      end function

      subroutine omp_set_default_allocator(allocator)
        include 'omp_lib_kinds.h'
        integer (kind=omp_allocator_kind) allocator
        return
      end subroutine

      function omp_get_default_allocator
        include 'omp_lib_kinds.h'
        integer (kind=omp_allocator_kind) omp_get_default_allocator
        omp_get_default_allocator = omp_null_allocator
```

```
1          end function
2
```

This page intentionally left blank

1 **APPENDIX B**

2
Interface Declarations

3 This appendix gives examples of the C/C++ header file, the Fortran **`include`** file and Fortran
4 **`module`** that shall be provided by implementations as specified in Chapter 3. It also includes an
5 example of a Fortran 90 generic interface for a library routine. This is a non-normative section,
6 implementation files may differ.

B.1 Example of the omp.h Header File

```
#ifndef _OMP_H_DEF
#define _OMP_H_DEF

/*
 * define the lock data types
 */
typedef void *omp_lock_t;

typedef void *omp_nest_lock_t;

/*
 * define the synchronization hints
 */
typedef enum omp_sync_hint_t {
  omp_sync_hint_none = 0,
  omp_lock_hint_none = omp_sync_hint_none,
  omp_sync_hint_uncontended = 1,
  omp_lock_hint_uncontended = omp_sync_hint_uncontended,
  omp_sync_hint_contended = 2,
  omp_lock_hint_contended = omp_sync_hint_contended,
  omp_sync_hint_nonspeculative = 4,
  omp_lock_hint_nonspeculative = omp_sync_hint_nonspeculative,
  omp_sync_hint_speculative = 8,
  omp_lock_hint_speculative = omp_sync_hint_speculative
/* , Add vendor specific constants for lock hints here,
   starting from the most-significant bit. */
} omp_sync_hint_t;

/* omp_lock_hint_t has been deprecated */
typedef omp_sync_hint_t omp_lock_hint_t;

/*
 * define the schedule kinds
 */
typedef enum omp_sched_t
{
 omp_sched_static = 1,
 omp_sched_dynamic = 2,
 omp_sched_guided = 3,
 omp_sched_auto = 4
/* , Add vendor specific schedule constants here */
} omp_sched_t;

/*
* define the proc bind values
```

```
*/
typedef enum omp_proc_bind_t
{
 omp_proc_bind_false = 0,
 omp_proc_bind_true = 1,
 omp_proc_bind_master = 2,
 omp_proc_bind_close = 3,
 omp_proc_bind_spread = 4
} omp_proc_bind_t;

/*
 * define the tool control commands
 */
typedef omp_control_tool_t
{
  omp_control_tool_start = 1,
  omp_control_tool_pause = 2,
  omp_control_tool_flush = 3,
  omp_control_tool_end = 4,
} omp_control_tool_t;


/*
 * define memory management types
 */
typedef void *omp_allocator_t;
enum { OMP_NULL_ALLOCATOR = NULL };

/*
 * exported OpenMP functions
 */
#ifdef __cplusplus
extern "C"
{
#endif

extern void omp_set_num_threads(int num_threads);
extern int omp_get_num_threads(void);
extern int omp_get_max_threads(void);
extern int omp_get_thread_num(void);
extern int omp_get_num_procs(void);
extern int omp_in_parallel(void);
extern void omp_set_dynamic(int dynamic_threads);
extern int omp_get_dynamic(void);
extern int omp_get_cancellation(void);
extern void omp_set_nested(int nested);
extern int omp_get_nested(void);
```

```
extern void omp_set_schedule(omp_sched_t kind, int chunk_size);
extern void omp_get_schedule(omp_sched_t *kind, int *chunk_size);
extern int omp_get_thread_limit(void);
extern void omp_set_max_active_levels(int max_active_levels);
extern int omp_get_max_active_levels(void);
extern int omp_get_level(void);
extern int omp_get_ancestor_thread_num(int level);
extern int omp_get_team_size(int level);
extern int omp_get_active_level(void);
extern int omp_in_final(void);
extern omp_proc_bind_t omp_get_proc_bind(void);
extern int omp_get_num_places(void);
extern int omp_get_place_num_procs(int place_num);
extern void omp_get_place_proc_ids(int place_num, int *ids);
extern int omp_get_place_num(void);
extern int omp_get_partition_num_places(void);
extern void omp_get_partition_place_nums(int *place_nums);

extern void omp_set_affinity_format(char const *format);
extern size_t omp_get_affinity_format(char* buffer, size_t size);
extern void omp_display_affinity(char const *format);
extern size_t omp_capture_affinity(char *buffer, size_t size, char const *format);

extern void omp_set_default_device(int device_num);
extern int omp_get_default_device(void);

extern int omp_get_num_devices(void);
extern int omp_get_device_num(void);
extern int omp_get_num_teams(void);
extern int omp_get_team_num(void);
extern int omp_is_initial_device(void);
extern int omp_get_initial_device(void);
extern int omp_get_max_task_priority(void);

extern void omp_init_lock(omp_lock_t *lock);
extern void omp_init_lock_with_hint(omp_lock_t *lock,
                                    omp_lock_hint_t hint);
extern void omp_destroy_lock(omp_lock_t *lock);
extern void omp_set_lock(omp_lock_t *lock);
extern void omp_unset_lock(omp_lock_t *lock);
extern int omp_test_lock(omp_lock_t *lock);

extern void omp_init_nest_lock(omp_nest_lock_t *lock);
extern void omp_init_nest_lock_with_hint(omp_nest_lock_t *lock,
                                         omp_lock_hint_t hint);
extern void omp_destroy_nest_lock(omp_nest_lock_t *lock);
extern void omp_set_nest_lock(omp_nest_lock_t *lock);
```

```
extern void omp_unset_nest_lock(omp_nest_lock_t *lock);
extern int omp_test_nest_lock(omp_nest_lock_t *lock);

extern double omp_get_wtime(void);
extern double omp_get_wtick(void);

extern void * omp_target_alloc(size_t size, int device_num);
extern void omp_target_free(void * device_ptr, int device_num);
extern int omp_target_is_present(void * ptr, int device_num);
extern int omp_target_memcpy(void *dst, void *src, size_t length,
                              size_t dst_offset, size_t src_offset,
                              int dst_device_num, int src_device_num);
extern int omp_target_memcpy_rect(
    void *dst, void *src,
    size_t element_size,
    int num_dims,
    const size_t *volume,
    const size_t *dst_offsets,
    const size_t *src_offsets,
    const size_t *dst_dimensions,
    const size_t *src_dimensions,
    int dst_device_num, int src_device_num);
extern int omp_target_associate_ptr(void * host_ptr,
                                     void * device_ptr,
                                     size_t size,
                                     size_t device_offset,
                                     int device_num);
extern int omp_target_disassociate_ptr(void * ptr,
                                        int device_num);

extern void omp_control_tool(int command, int modifier, void *arg);

extern void omp_set_default_allocator(const omp_allocator_t *allocator);
extern const omp_allocator_t * omp_get_default_allocator (void);


extern const omp_allocator_t * omp_default_mem_alloc;
extern const omp_allocator_t * omp_large_cap_mem_alloc;
extern const omp_allocator_t * omp_const_mem_alloc;
extern const omp_allocator_t * omp_high_bw_mem_alloc;
extern const omp_allocator_t * omp_low_lat_mem_alloc;
extern const omp_allocator_t * omp_cgroup_mem_alloc;
extern const omp_allocator_t * omp_pteam_mem_alloc;
extern const omp_allocator_t * omp_thread_mem_alloc;


#ifdef __cplusplus
```

```
extern void * omp_alloc (size_t size,
                          const omp_allocator_t *allocator = OMP_NULL_ALLOCATOR);
extern void omp_free (void * ptr,
                      const omp_allocator_t *allocator = OMP_NULL_ALLOCATOR);
#else
extern void * omp_alloc (size_t size, const omp_allocator_t *allocator);
extern void omp_free (void * ptr, const omp_allocator_t *allocator);
#endif


#ifdef __cplusplus
}
#endif

#endif
```

B.2 Example of an Interface Declaration `include` File

omp_lib_kinds.h:

```
    integer omp_lock_kind
    integer omp_nest_lock_kind
    integer omp_control_tool_kind
    integer omp_control_tool_result_kind
! this selects an integer that is large enough to hold a 64 bit integer
    parameter ( omp_lock_kind = selected_int_kind( 10 ) )
    parameter ( omp_nest_lock_kind = selected_int_kind( 10 ) )

    integer omp_sched_kind
! this selects an integer that is large enough to hold a 32 bit integer
    parameter ( omp_sched_kind = selected_int_kind( 8 ) )
    integer ( omp_sched_kind ) omp_sched_static
    parameter ( omp_sched_static = 1 )
    integer ( omp_sched_kind ) omp_sched_dynamic
    parameter ( omp_sched_dynamic = 2 )
    integer ( omp_sched_kind ) omp_sched_guided
    parameter ( omp_sched_guided = 3 )
    integer ( omp_sched_kind ) omp_sched_auto
    parameter ( omp_sched_auto = 4 )

    integer omp_proc_bind_kind
    parameter ( omp_proc_bind_kind = selected_int_kind( 8 ) )
    integer ( omp_proc_bind_kind ) omp_proc_bind_false
    parameter ( omp_proc_bind_false = 0 )
    integer ( omp_proc_bind_kind ) omp_proc_bind_true
    parameter ( omp_proc_bind_true = 1 )
    integer ( omp_proc_bind_kind ) omp_proc_bind_master
    parameter ( omp_proc_bind_master = 2 )
    integer ( omp_proc_bind_kind ) omp_proc_bind_close
    parameter ( omp_proc_bind_close = 3 )
    integer ( omp_proc_bind_kind ) omp_proc_bind_spread
    parameter ( omp_proc_bind_spread = 4 )
```

```
integer omp_sync_hint_kind
parameter ( omp_sync_hint_kind = selected_int_kind( 10 ) )
integer omp_lock_hint_kind
parameter ( omp_lock_hint_kind = omp_sync_hint_kind )
integer ( omp_sync_hint_kind) omp_sync_hint_none
parameter ( omp_sync_hint_none = 0 )
integer ( omp_lock_hint_kind) omp_lock_hint_none
parameter ( omp_lock_hint_none = omp_sync_hint_none )
integer ( omp_sync_hint_kind) omp_sync_hint_uncontended
parameter ( omp_sync_hint_uncontended = 1 )
integer ( omp_lock_hint_kind) omp_lock_hint_uncontended
parameter ( omp_lock_hint_uncontended = omp_sync_hint_uncontended )
integer ( omp_sync_hint_kind) omp_sync_hint_contended
parameter ( omp_sync_hint_contended = 2 )
integer ( omp_lock_hint_kind) omp_lock_hint_contended
parameter ( omp_lock_hint_contended = omp_sync_hint_contended )
integer ( omp_sync_hint_kind) omp_sync_hint_nonspeculative
parameter ( omp_sync_hint_nonspeculative = 4 )
integer ( omp_lock_hint_kind) omp_lock_hint_nonspeculative
parameter ( omp_lock_hint_nonspeculative = omp_sync_hint_nonspeculative )
integer ( omp_sync_hint_kind) omp_sync_hint_speculative
parameter ( omp_sync_hint_speculative = 8 )
integer ( omp_lock_hint_kind) omp_lock_hint_speculative
parameter ( omp_lock_hint_speculative = omp_sync_hint_speculative )


parameter ( omp_control_tool_kind = selected_int_kind( 8 ) )
integer ( omp_control_tool_kind ) omp_control_tool_start
parameter ( omp_control_tool_start = 1 )
integer ( omp_control_tool_kind ) omp_control_tool_pause
parameter ( omp_control_tool_pause = 2 )
integer ( omp_control_tool_kind ) omp_control_tool_flush
parameter ( omp_control_tool_flush = 3 )
integer ( omp_control_tool_kind ) omp_control_tool_end
parameter ( omp_control_tool_end = 4 )
```

```
      parameter ( omp_control_tool_result_kind = selected_int_kind( 8 ) )
      integer ( omp_control_tool_result_kind ) omp_control_tool_notool
      parameter ( omp_control_tool_notool = -2 )
      integer ( omp_control_tool_result_kind ) omp_control_tool_nocallback
      parameter ( omp_control_tool_nocallback = -1 )
      integer ( omp_control_tool_result_kind ) omp_control_tool_success
      parameter ( omp_control_tool_success = 0 )
      integer ( omp_control_tool_result_kind ) omp_control_tool_ignored
      parameter ( omp_control_tool_ignored = 1 )


      integer omp_allocator_kind
      parameter ( omp_allocator_kind = selected_int_kind( 8 ) )
      integer ( omp_allocator_kind ) omp_null_allocator
      parameter ( omp_null_allocator = 0 )
      integer ( omp_allocator_kind ) omp_default_mem_alloc
      parameter ( omp_default_mem_alloc = 1 )
      integer ( omp_allocator_kind ) omp_large_cap_mem_alloc
      parameter ( omp_large_cap_mem_alloc = 2 )
      integer ( omp_allocator_kind ) omp_const_mem_alloc
      parameter ( omp_const_mem_alloc = 3 )
      integer ( omp_allocator_kind ) omp_high_bw_mem_alloc
      parameter ( omp_high_bw_mem_alloc = 4 )
      integer ( omp_allocator_kind ) omp_low_lat_mem_alloc
      parameter ( omp_low_lat_mem_alloc = 5 )
      integer ( omp_allocator_kind ) omp_cgroup_mem_alloc
      parameter ( omp_cgroup_mem_alloc = 6 )
      integer ( omp_allocator_kind ) omp_pteam_mem_alloc
      parameter ( omp_pteam_mem_alloc = 7 )
      integer ( omp_allocator_kind ) omp_thread_mem_alloc
      parameter ( omp_thread_mem_alloc = 8 )
```

omp_lib.h:

```
! default integer type assumed below
! default logical type assumed below
! OpenMP API v5.0 Preview 2 (TR6)

      include 'omp_lib_kinds.h'
      integer openmp_version
      parameter ( openmp_version = 201711 )

      external omp_set_num_threads
      external omp_get_num_threads
      integer omp_get_num_threads
      external omp_get_max_threads
```

```
1          integer omp_get_max_threads
2          external omp_get_thread_num
3          integer omp_get_thread_num
4          external omp_get_num_procs
5          integer omp_get_num_procs
6          external omp_in_parallel
7          logical omp_in_parallel
8          external omp_set_dynamic
9          external omp_get_dynamic
10         logical omp_get_dynamic
11         external omp_get_cancellation
12         logical omp_get_cancellation
13         external omp_set_nested
14         external omp_get_nested
15         logical omp_get_nested
16         external omp_set_schedule
17         external omp_get_schedule
18         external omp_get_thread_limit
19         integer omp_get_thread_limit
20         external omp_set_max_active_levels
21         external omp_get_max_active_levels
22         integer omp_get_max_active_levels
23         external omp_get_level
24         integer omp_get_level
25         external omp_get_ancestor_thread_num
26         integer omp_get_ancestor_thread_num
27         external omp_get_team_size
28         integer omp_get_team_size
29         external omp_get_active_level
30         integer omp_get_active_level
31         external omp_set_affinity_format
32         external omp_get_affinity_format
33         integer omp_get_affinity_format
34         external omp_display_affinity
35         external omp_capture_affinity
36         integer omp_capture_affinity
37         external omp_set_default_device
38         external omp_get_default_device
39         integer omp_get_default_device
40         external omp_get_num_devices
41         integer omp_get_num_devices
42         external omp_get_device_num
43         integer omp_get_device_num
44         external omp_get_num_teams
45         integer omp_get_num_teams
46         external omp_get_team_num
47         integer omp_get_team_num
```

```
external omp_is_initial_device
logical omp_is_initial_device
external omp_get_initial_device
integer omp_get_initial_device
external omp_get_max_task_priority
integer omp_get_max_task_priority

external omp_in_final
logical omp_in_final

integer ( omp_proc_bind_kind ) omp_get_proc_bind
external omp_get_proc_bind
integer omp_get_num_places
external omp_get_num_places
integer omp_get_place_num_procs
external omp_get_place_num_procs
external omp_get_place_proc_ids
integer omp_get_place_num
external omp_get_place_num
integer omp_get_partition_num_places
external omp_get_partition_num_places
external omp_get_partition_place_nums

external omp_init_lock
external omp_init_lock_with_hint
external omp_destroy_lock
external omp_set_lock
external omp_unset_lock
external omp_test_lock
logical omp_test_lock

external omp_init_nest_lock
external omp_init_nest_lock_with_hint
external omp_destroy_nest_lock
external omp_set_nest_lock
external omp_unset_nest_lock
external omp_test_nest_lock
integer omp_test_nest_lock

external omp_get_wtick
double precision omp_get_wtick
external omp_get_wtime
double precision omp_get_wtime

integer  omp_control_tool
external omp_control_tool

```

```
external omp_set_default_allocator
external omp_get_default_allocator
integer ( omp_allocator_kind ) omp_get_default_allocator
```

B.3 Example of a Fortran Interface Declaration module

```
!      the "!" of this comment starts in column 1
!23456

       module omp_lib_kinds
       integer, parameter :: omp_lock_kind = selected_int_kind( 10 )
       integer, parameter :: omp_nest_lock_kind = selected_int_kind( 10 )
       integer, parameter :: omp_lock_hint_kind = selected_int_kind( 10 )
       integer (kind=omp_lock_hint_kind), parameter ::
     &   omp_lock_hint_none = 0
       integer (kind=omp_lock_hint_kind), parameter ::
     &   omp_lock_hint_uncontended = 1
       integer (kind=omp_lock_hint_kind), parameter ::
     &   omp_lock_hint_contended = 2
       integer (kind=omp_lock_hint_kind), parameter ::
     &   omp_lock_hint_nonspeculative = 4
       integer (kind=omp_lock_hint_kind), parameter ::
     &   omp_lock_hint_speculative = 8

       integer, parameter :: omp_sched_kind = selected_int_kind( 8 )
       integer(kind=omp_sched_kind), parameter ::
     &   omp_sched_static = 1
       integer(kind=omp_sched_kind), parameter ::
     &   omp_sched_dynamic = 2
       integer(kind=omp_sched_kind), parameter ::
     &   omp_sched_guided = 3
       integer(kind=omp_sched_kind), parameter ::
     &   omp_sched_auto = 4

       integer, parameter :: omp_proc_bind_kind = selected_int_kind( 8 )
       integer (kind=omp_proc_bind_kind), parameter ::
     &   omp_proc_bind_false = 0
       integer (kind=omp_proc_bind_kind), parameter ::
     &   omp_proc_bind_true = 1
       integer (kind=omp_proc_bind_kind), parameter ::
     &   omp_proc_bind_master = 2
       integer (kind=omp_proc_bind_kind), parameter ::
     &   omp_proc_bind_close = 3
       integer (kind=omp_proc_bind_kind), parameter ::
     &   omp_proc_bind_spread = 4
```

```
      integer, parameter :: omp_control_tool_kind = selected_int_kind( 8 )
      integer (kind=omp_control_tool_kind), parameter ::
     &   omp_control_tool_start = 1
      integer (kind=omp_control_tool_kind), parameter ::
     &   omp_control_tool_pause = 2
      integer (kind=omp_control_tool_kind), parameter ::
     &   omp_control_tool_flush = 3
      integer (kind=omp_control_tool_kind), parameter ::
     &   omp_control_tool_end = 4
      end module omp_lib_kinds


      integer, parameter :: omp_control_tool_result_kind =
     &   selected_int_kind( 8 )
      integer ( omp_control_tool_result_kind ), parameter ::
     &   omp_control_tool_notool = -2
      integer ( omp_control_tool_result_kind ), parameter ::
     &   omp_control_tool_nocallback = -1
      integer ( omp_control_tool_result_kind ), parameter ::
     &   omp_control_tool_success = 0
      integer ( omp_control_tool_result_kind ), parameter ::
     &   omp_control_tool_ignored = 1


      integer, pamameter :: omp_allocator_kind =
     &  selected_int_kind( 8 )
      integer ( omp_allocator_kind ), parameter ::
     & omp_null_allocator = 0
      integer ( omp_allocator_kind ), parameter ::
     & omp_default_mem_alloc = 1
      integer ( omp_allocator_kind ), parameter ::
     & omp_large_cap_mem_alloc = 2
      integer ( omp_allocator_kind ), parameter ::
     & omp_const_mem_alloc = 3
      integer ( omp_allocator_kind ), parameter ::
     & omp_high_bw_mem_alloc = 4
      integer ( omp_allocator_kind ), parameter ::
     & omp_low_lat_mem_alloc = 5
      integer ( omp_allocator_kind ), parameter ::
     & omp_cgroup_mem_alloc = 6
      integer ( omp_allocator_kind ), parameter ::
     & omp_pteam_mem_alloc = 7
      integer ( omp_allocator_kind ), parameter ::
     & omp_thread_mem_alloc = 8
```

```
      module omp_lib

        use omp_lib_kinds

!                             OpenMP API v5.0 Preview 2 (TR6)
        integer, parameter :: openmp_version = 201711

       interface

        subroutine omp_set_num_threads (num_threads)
         integer, intent(in) :: num_threads
        end subroutine omp_set_num_threads

        function omp_get_num_threads ()
         integer :: omp_get_num_threads
        end function omp_get_num_threads

        function omp_get_max_threads ()
         integer :: omp_get_max_threads
        end function omp_get_max_threads

        function omp_get_thread_num ()
         integer :: omp_get_thread_num
        end function omp_get_thread_num

        function omp_get_num_procs ()
         integer :: omp_get_num_procs
        end function omp_get_num_procs

        function omp_in_parallel ()
         logical :: omp_in_parallel
        end function omp_in_parallel

        subroutine omp_set_dynamic (dynamic_threads)
         logical, intent(in) ::dynamic_threads
        end subroutine omp_set_dynamic

        function omp_get_dynamic ()
         logical :: omp_get_dynamic
        end function omp_get_dynamic

        function omp_get_cancellation ()
         logical :: omp_get_cancellation
        end function omp_get_cancellation

        subroutine omp_set_nested (nested)
         logical, intent(in) :: nested
```

```
end subroutine omp_set_nested

function omp_get_nested ()
 logical :: omp_get_nested
end function omp_get_nested

subroutine omp_set_schedule (kind, chunk_size)
 use omp_lib_kinds
 integer(kind=omp_sched_kind), intent(in) :: kind
 integer, intent(in) :: chunk_size
end subroutine omp_set_schedule

subroutine omp_get_schedule (kind, chunk_size)
 use omp_lib_kinds
 integer(kind=omp_sched_kind), intent(out) :: kind
 integer, intent(out)::chunk_size
end subroutine omp_get_schedule

function omp_get_thread_limit ()
 integer :: omp_get_thread_limit
end function omp_get_thread_limit

subroutine omp_set_max_active_levels (max_levels)
 integer, intent(in) :: max_levels
end subroutine omp_set_max_active_levels

function omp_get_max_active_levels ()
 integer :: omp_get_max_active_levels
end function omp_get_max_active_levels

function omp_get_level()
 integer :: omp_get_level
end function omp_get_level

function omp_get_ancestor_thread_num (level)
 integer, intent(in) :: level
 integer :: omp_get_ancestor_thread_num
end function omp_get_ancestor_thread_num

function omp_get_team_size (level)
 integer, intent(in) :: level
 integer :: omp_get_team_size
end function omp_get_team_size

function omp_get_active_level ()
 integer :: omp_get_active_level
end function omp_get_active_level
```

```
function omp_in_final ()
 logical :: omp_in_final
end function omp_in_final

function omp_get_proc_bind ()
 use omp_lib_kinds
 integer(kind=omp_proc_bind_kind) :: omp_get_proc_bind
 omp_get_proc_bind = omp_proc_bind_false
end function omp_get_proc_bind

function omp_get_num_places ()
 integer :: omp_get_num_places
end function omp_get_num_places

function omp_get_place_num_procs (place_num)
 integer, intent(in) :: place_num
 integer :: omp_get_place_num_procs
end function omp_get_place_num_procs

subroutine omp_get_place_proc_ids (place_num, ids)
 integer, intent(in) :: place_num
 integer, intent(out) :: ids(*)
end subroutine omp_get_place_proc_ids

function omp_get_place_num ()
 integer :: omp_get_place_num
end function omp_get_place_num

function omp_get_partition_num_places ()
 integer :: omp_get_partition_num_places
end function omp_get_partition_num_places

subroutine omp_get_partition_place_nums (place_nums)
 integer, intent(out) :: place_nums(*)
end subroutine omp_get_partition_place_nums

subroutine omp_set_affinity_format(format)
 character(len=*),intent(in)::format
end subroutine omp_set_affinity_format

function omp_get_affinity_format(buffer)
 character(len=*),intent(out)::buffer
 integer :: omp_get_affinity_format
end function omp_get_affinity_format

subroutine omp_display_affinity(format)
```

```
   character(len=*),intent(in)::format
  end subroutine omp_display_affinity

  function omp_capture_affinity(buffer,format)
   character(len=*),intent(out)::buffer
   character(len=*),intent(in)::format
   integer :: omp_capture_affinity
  end function omp_capture_affinity

  subroutine omp_set_default_device (device_num)
   integer :: device_num
  end subroutine omp_set_default_device

  function omp_get_default_device ()
   integer :: omp_get_default_device
  end function omp_get_default_device

  function omp_get_num_devices ()
   integer :: omp_get_num_devices
  end function omp_get_num_devices

  function omp_get_device_num ()
   integer :: omp_get_device_num
  end function omp_get_device_num

  function omp_get_num_teams ()
   integer :: omp_get_num_teams
  end function omp_get_num_teams

  function omp_get_team_num ()
   integer :: omp_get_team_num
  end function omp_get_team_num

  function omp_is_initial_device ()
   logical :: omp_is_initial_device
  end function omp_is_initial_device

  function omp_get_initial_device ()
   integer :: omp_get_initial_device
  end function omp_get_initial_device

  function omp_get_max_task_priority ()
   integer :: omp_get_max_task_priority
  end function omp_get_max_task_priority

  subroutine omp_init_lock (svar)
   use omp_lib_kinds
```

```
   integer(kind=omp_lock_kind), intent(out) :: svar
  end subroutine omp_init_lock

  subroutine omp_init_lock_with_hint (svar, hint)
   use omp_lib_kinds
   integer(kind=omp_lock_kind), intent(out) :: svar
   integer(kind=omp_lock_hint_kind), intent(in) :: hint
  end subroutine omp_init_lock_with_hint

  subroutine omp_destroy_lock (svar)
   use omp_lib_kinds
   integer(kind=omp_lock_kind), intent(inout) :: svar
  end subroutine omp_destroy_lock

  subroutine omp_set_lock (svar)
   use omp_lib_kinds
   integer(kind=omp_lock_kind), intent(inout) :: svar
  end subroutine omp_set_lock

  subroutine omp_unset_lock (svar)
   use omp_lib_kinds
   integer(kind=omp_lock_kind), intent(inout) :: svar
  end subroutine omp_unset_lock

  function omp_test_lock (svar)
   use omp_lib_kinds
   logical :: omp_test_lock
   integer(kind=omp_lock_kind), intent(inout) :: svar
  end function omp_test_lock

  subroutine omp_init_nest_lock (nvar)
   use omp_lib_kinds
   integer(kind=omp_nest_lock_kind), intent(out) :: nvar
  end subroutine omp_init_nest_lock

  subroutine omp_init_nest_lock_with_hint (nvar, hint)
   use omp_lib_kinds
   integer(kind=omp_nest_lock_kind), intent(out) :: nvar
   integer(kind=omp_lock_hint_kind), intent(in) :: hint
  end subroutine omp_init_nest_lock_with_hint

  subroutine omp_destroy_nest_lock (nvar)
   use omp_lib_kinds
   integer(kind=omp_nest_lock_kind), intent(inout) :: nvar
  end subroutine omp_destroy_nest_lock

  subroutine omp_set_nest_lock (nvar)
```

```
      use omp_lib_kinds
      integer(kind=omp_nest_lock_kind), intent(inout) :: nvar
     end subroutine omp_set_nest_lock

     subroutine omp_unset_nest_lock (nvar)
      use omp_lib_kinds
      integer(kind=omp_nest_lock_kind), intent(inout) :: nvar
     end subroutine omp_unset_nest_lock

     function omp_test_nest_lock (nvar)
      use omp_lib_kinds
      integer :: omp_test_nest_lock
      integer(kind=omp_nest_lock_kind), intent(inout) :: nvar
     end function omp_test_nest_lock

     function omp_get_wtick ()
      double precision :: omp_get_wtick
     end function omp_get_wtick

     function omp_get_wtime ()
      double precision :: omp_get_wtime
     end function omp_get_wtime

     function omp_control_tool (command, modifier)
      use omp_lib_kinds
      integer :: omp_control_tool
      integer(kind=omp_control_tool_kind), intent(in) :: command
      integer(kind=omp_control_tool_kind), intent(in) :: modifier
     end function omp_control_tool


     subroutine omp_set_default_allocator (svar)
      use omp_lib_kinds
      integer(kind=omp_allocator_kind), intent(in) :: svar
     end subroutine omp_set_default_allocator

     function omp_get_default_allocator ()
      use omp_lib_kinds
      integer(kind=omp_allocator_kind) :: omp_get_default_allocator
     end function omp_get_default_allocator

     end interface

   end module omp_lib
```

B.4 Example of a Generic Interface for a Library Routine

Any of the OpenMP runtime library routines that take an argument may be extended with a generic interface so arguments of different **KIND** type can be accommodated.

The **OMP_SET_NUM_THREADS** interface could be specified in the **omp_lib** module as follows:

```
interface omp_set_num_threads

    subroutine omp_set_num_threads_4(num_threads)
      use omp_lib_kinds
      integer(4), intent(in) :: num_threads
    end subroutine omp_set_num_threads_4

    subroutine omp_set_num_threads_8(num_threads)
      use omp_lib_kinds
      integer(8), intent(in) :: num_threads
    end subroutine omp_set_num_threads_8

end interface omp_set_num_threads
```

APPENDIX C

OpenMP Implementation-Defined Behaviors

This appendix summarizes the behaviors that are described as implementation defined in this API. Each behavior is cross-referenced back to its description in the main specification. An implementation is required to define and document its behavior in these cases.

- **Processor**: a hardware unit that is implementation defined (see Section 1.2.1 on page 2).
- **Device**: an implementation defined logical execution engine (see Section 1.2.1 on page 2).
- **Device address**: an address in a *device data environment* (see Section 1.2.6 on page 12).
- **Memory model**: the minimum size at which a memory update may also read and write back adjacent variables that are part of another variable (as array or structure elements) is implementation defined but is no larger than required by the base language (see Section 1.4.1 on page 20).
- **Memory model**: Implementations are allowed to relax the ordering imposed by implicit flush operations when the result is only visible to programs using non-sequentially consistent atomic directives (see Section 1.4.6 on page 25).
- **Internal control variables**: the initial values of *dyn-var*, *nest-var*, *nthreads-var*, *run-sched-var*, *def-sched-var*, *bind-var*, *stacksize-var*, *wait-policy-var*, *thread-limit-var*, *max-active-levels-var*, *place-partition-var*, *affinity-format-var*, *default-device-var* and *def-allocator-var* are implementation defined. The method for initializing a target device's internal control variable is implementation defined (see Section 2.4.2 on page 51).
- **Dynamic adjustment of threads**: providing the ability to dynamically adjust the number of threads is implementation defined . Implementations are allowed to deliver fewer threads (but at least one) than indicated in Algorithm 2-1 even if dynamic adjustment is disabled (see Section 2.8.1 on page 71).

- **Thread affinity**: For the **close** thread affinity policy, if $T > P$ and P does not divide T evenly, the exact number of threads in a particular place is implementation defined. For the **spread** thread affinity, if $T > P$ and P does not divide T evenly, the exact number of threads in a particular subpartition is implementation defined. The determination of whether the affinity request can be fulfilled is implementation defined. If not, the number of threads in the team and their mapping to places become implementation defined (see Section 2.8.2 on page 73).

- **Loop directive**: the integer type (or kind, for Fortran) used to compute the iteration count of a collapsed loop is implementation defined. The effect of the **schedule(runtime)** clause when the *run-sched-var* ICV is set to **auto** is implementation defined. The *simd_width* used when a **simd** schedule modifier is specified is implementation defined (see Section 2.10.1 on page 78).

- **sections construct**: the method of scheduling the structured blocks among threads in the team is implementation defined (see Section 2.10.2 on page 86).

- **single construct**: the method of choosing a thread to execute the structured block is implementation defined (see Section 2.10.3 on page 90)

- **simd construct**: the integer type (or kind, for Fortran) used to compute the iteration count for the collapsed loop is implementation defined. The number of iterations that are executed concurrently at any given time is implementation defined. If the *alignment* parameter is not specified in the **aligned** clause, the default alignments for the SIMD instructions are implementation defined (see Section 2.11.1 on page 96).

- **declare simd directive**: if the parameter of the **simdlen** clause is not a constant positive integer expression, the number of concurrent arguments for the function is implementation defined. If the *alignment* parameter of the **aligned** clause is not specified, the default alignments for SIMD instructions are implementation defined (see Section 2.11.2 on page 100).

- **taskloop construct**: The number of loop iterations assigned to a task created from a **taskloop** construct is implementation defined, unless the **grainsize** or **num_tasks** clauses are specified. The integer type (or kind, for Fortran) used to compute the iteration count for the collapsed loop is implementation defined (see Section 2.13.2 on page 114).

- **is_device_ptr clause**: Support for pointers created outside of the OpenMP device data management routines is implementation defined (see Section 2.15.5 on page 141).

- **target construct**: the effect of invoking a virtual member function of an object on a device other than the device on which the object was constructed is implementation defined (see Section 2.15.5 on page 141).

- **teams construct**: the number of teams that are created is implementation defined but less than or equal to the value of the **num_teams** clause if specified. The maximum number of threads participating in the contention group that each team initiates is implementation defined but less than or equal to the value of the **thread_limit** clause if specified (see Section 2.15.9 on page 157).

- **`distribute` construct**: the integer type (or kind, for Fortran) used to compute the iteration count for the collapsed loop is implementation defined (see Section 2.15.10 on page 160).
- **`distribute` construct**: If no **`dist_schedule`** clause is specified then the schedule for the **`distribute`** construct is implementation defined (see Section 2.15.10 on page 160).
- **`critical` construct**: the effect of using a **`hint`** clause is implementation defined (see Section 2.18.2 on page 195 and Section 2.18.11 on page 229).
- **`atomic` construct**: a compliant implementation may enforce exclusive access between **`atomic`** regions that update different storage locations. The circumstances under which this occurs are implementation defined. If the storage location designated by x is not size-aligned (that is, if the byte alignment of x is not a multiple of the size of x), then the behavior of the atomic region is implementation defined (see Section 2.18.7 on page 206). The effect of using a **`hint`** clause is implementation defined (see Section 2.18.7 on page 206 and Section 2.18.11 on page 229).

Fortran

- **Data-sharing attributes**: The data-sharing attributes of dummy arguments without the **`VALUE`** attribute are implementation-defined if the associated actual argument is shared, except for the conditions specified (see Section 2.20.1.2 on page 243).
- **`threadprivate` directive**: if the conditions for values of data in the threadprivate objects of threads (other than an initial thread) to persist between two consecutive active parallel regions do not all hold, the allocation status of an allocatable variable in the second region is implementation defined (see Section 2.20.2 on page 244).
- **Runtime library definitions**: it is implementation defined whether the include file **`omp_lib.h`** or the module **`omp_lib`** (or both) is provided. It is implementation defined whether any of the OpenMP runtime library routines that take an argument are extended with a generic interface so arguments of different **`KIND`** type can be accommodated (see Section 3.1 on page 298).

Fortran

- **`omp_set_num_threads` routine**: if the argument is not a positive integer the behavior is implementation defined (see Section 3.2.1 on page 300).
- **`omp_set_schedule` routine**: for implementation specific schedule types, the values and associated meanings of the second argument are implementation defined. (see Section 3.2.12 on page 311).
- **`omp_set_max_active_levels` routine**: when called from within any explicit **`parallel`** region the binding thread set (and binding region, if required) for the **`omp_set_max_active_levels`** region is implementation defined and the behavior is implementation defined. If the argument is not a non-negative integer then the behavior is implementation defined (see Section 3.2.15 on page 315).

- **omp_get_max_active_levels routine**: when called from within any explicit **parallel** region the binding thread set (and binding region, if required) for the **omp_get_max_active_levels** region is implementation defined (see Section 3.2.16 on page 317).
- **omp_get_place_proc_ids routine**: the meaning of the nonnegative numerical identifiers returned by the **omp_get_place_proc_ids** routine is implementation defined (see Section 3.2.25 on page 327).
- **omp_set_affinity_format routine**: when called from within any explicit **parallel** region the binding thread set (and binding region, if required) for the **omp_set_affinity_format** region is implementation defined and the behavior is implementation defined. If the argument does not conform to the specified format then the result is implementation defined (see Section 3.2.29 on page 331).
- **omp_get_affinity_format routine**: when called from within any explicit **parallel** region the binding thread set (and binding region, if required) for the **omp_get_affinity_format** region is implementation defined (see Section 3.2.30 on page 332).
- **omp_display_affinity routine**: if the argument does not conform to the specified format then the result is implementation defined (see Section 3.2.31 on page 333).
- **omp_capture_affinity routine**: if the *format* argument does not conform to the specified format then the result is implementation defined (see Section 3.2.32 on page 334).
- **omp_get_initial_device routine**: the value of the device number is implementation defined (see Section 3.2.40 on page 342).
- **omp_init_lock_with_hint** and **omp_init_nest_lock_with_hint routines**: if hints are stored with a lock variable, the effect of the hints on the locks are implementation defined (see Section 3.3.2 on page 347).
- **omp_target_memcpy_rect routine**: the maximum number of dimensions supported is implementation defined, but must be at least three (see Section 3.5.5 on page 363).
- **OMP_SCHEDULE environment variable**: if the value does not conform to the specified format then the result is implementation defined (see Section 5.1 on page 564).
- **OMP_NUM_THREADS environment variable**: if any value of the list specified in the **OMP_NUM_THREADS** environment variable leads to a number of threads that is greater than the implementation can support, or if any value is not a positive integer, then the result is implementation defined (see Section 5.2 on page 565).
- **OMP_PROC_BIND environment variable**: if the value is not **true**, **false**, or a comma separated list of **master**, **close**, or **spread**, the behavior is implementation defined. The behavior is also implementation defined if an initial thread cannot be bound to the first place in the OpenMP place list (see Section 5.4 on page 566).

- **`OMP_DYNAMIC` environment variable**: if the value is neither **`true`** nor **`false`** the behavior is implementation defined (see Section 5.3 on page 566).
- **`OMP_NESTED` environment variable**: if the value is neither **`true`** nor **`false`** the behavior is implementation defined (see Section 5.6 on page 569).
- **`OMP_STACKSIZE` environment variable**: if the value does not conform to the specified format or the implementation cannot provide a stack of the specified size then the behavior is implementation defined (see Section 5.7 on page 570).
- **`OMP_WAIT_POLICY` environment variable**: the details of the **`ACTIVE`** and **`PASSIVE`** behaviors are implementation defined (see Section 5.8 on page 571).
- **`OMP_MAX_ACTIVE_LEVELS` environment variable**: if the value is not a non-negative integer or is greater than the number of parallel levels an implementation can support then the behavior is implementation defined (see Section 5.9 on page 572).
- **`OMP_THREAD_LIMIT` environment variable**: if the requested value is greater than the number of threads an implementation can support, or if the value is not a positive integer, the behavior of the program is implementation defined (see Section 5.10 on page 572).
- **`OMP_PLACES` environment variable**: the meaning of the numbers specified in the environment variable and how the numbering is done are implementation defined. The precise definitions of the abstract names are implementation defined. An implementation may add implementation-defined abstract names as appropriate for the target platform. When creating a place list of n elements by appending the number n to an abstract name, the determination of which resources to include in the place list is implementation defined. When requesting more resources than available, the length of the place list is also implementation defined. The behavior of the program is implementation defined when the execution environment cannot map a numerical value (either explicitly defined or implicitly derived from an interval) within the **`OMP_PLACES`** list to a processor on the target platform, or if it maps to an unavailable processor. The behavior is also implementation defined when the **`OMP_PLACES`** environment variable is defined using an abstract name (see Section 5.5 on page 567).
- **`OMP_AFFINITY_FORMAT` environment variable**: if the value does not conform to the specified format then the result is implementation defined (see Section 5.14 on page 575).
- **OMPT thread states**: The set of OMPT thread states supported is implementation defined (see Section 4.3.1.1 on page 551).
- **`ompt_callback_idle` tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **`ompt_set_callback`**, it is implementation defined whether the registered callback may never or sometimes invoke this callback for the associated events (see Table 4.2 on page 384)
- **`ompt_callback_sync_region_wait` tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **`ompt_set_callback`**, it is

implementation defined whether the registered callback may never or sometimes invoke this callback for the associated events (see Table 4.2 on page 384)

- **ompt_callback_mutex_released tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **ompt_set_callback**, it is implementation defined whether the registered callback may never or sometimes invoke this callback for the associated events (see Table 4.2 on page 384)
- **ompt_callback_task_dependences tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **ompt_set_callback**, it is implementation defined whether the registered callback may never or sometimes invoke this callback for the associated events (see Table 4.2 on page 384)
- **ompt_callback_task_dependence tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **ompt_set_callback**, it is implementation defined whether the registered callback may never or sometimes invoke this callback for the associated events (see Table 4.2 on page 384)
- **ompt_callback_work tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **ompt_set_callback**, it is implementation defined whether the registered callback may never or sometimes invoke this callback for the associated events (see Table 4.2 on page 384)
- **ompt_callback_master tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **ompt_set_callback**, it is implementation defined whether the registered callback may never or sometimes invoke this callback for the associated events (see Table 4.2 on page 384)
- **ompt_callback_target_map tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **ompt_set_callback**, it is implementation defined whether the registered callback may never or sometimes invoke this callback for the associated events (see Table 4.2 on page 384)
- **ompt_callback_sync_region tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **ompt_set_callback**, it is implementation defined whether the registered callback may never or sometimes invoke this callback for the associated events (see Table 4.2 on page 384)
- **ompt_callback_lock_init tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **ompt_set_callback**, it is implementation defined whether the registered callback may never or sometimes invoke this callback for the associated events (see Table 4.2 on page 384)
- **ompt_callback_lock_destroy tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **ompt_set_callback**, it is implementation defined whether the registered callback may never or sometimes invoke this callback for the associated events (see Table 4.2 on page 384)

- **ompt_callback_mutex_acquire tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **ompt_set_callback**, it is implementation defined whether the registered callback may never or sometimes invoke this callback for the associated events (see Table 4.2 on page 384)
- **ompt_callback_mutex_acquired tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **ompt_set_callback**, it is implementation defined whether the registered callback may never or sometimes invoke this callback for the associated events (see Table 4.2 on page 384)
- **ompt_callback_nest_lock tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **ompt_set_callback**, it is implementation defined whether the registered callback may never or sometimes invoke this callback for the associated events (see Table 4.2 on page 384)
- **ompt_callback_flush tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **ompt_set_callback**, it is implementation defined whether the registered callback may never or sometimes invoke this callback for the associated events (see Table 4.2 on page 384)
- **ompt_callback_cancel tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **ompt_set_callback**, it is implementation defined whether the registered callback may never or sometimes invoke this callback for the associated events (see Table 4.2 on page 384)
- **Device tracing**: Whether a target device supports tracing or not is implementation defined; if a target device does not support tracing, a **NULL** may be supplied for the *lookup* function to a tool's device initializer (see Section 4.1.1.4 on page 383).
- **ompt_set_trace_ompt runtime entry point**: it is implementation defined whether a device-specific tracing interface will define this runtime entry point, indicating that it can collect traces in OMPT format (see Section 4.1.1.4 on page 383).
- **ompt_buffer_get_record_ompt runtime entry point**: it is implementation defined whether a device-specific tracing interface will define this runtime entry point, indicating that it can collect traces in OMPT format (see Section 4.1.1.4 on page 383).
- **Memory allocators**: The storage resource that will be used by each memory allocator defined in Table 2.5 on page 64 is implementation defined.
- **allocate directive**: The effect of not being able to fulfill an allocation request specified in **allocate** directive is implementation defined.
- **allocate clause**: The effect of not being able to fulfill an allocation request specified in **allocate** clause is implementation defined.

This page intentionally left blank

1 **APPENDIX D**

2 Task Frame Management for the
3 Tool Interface

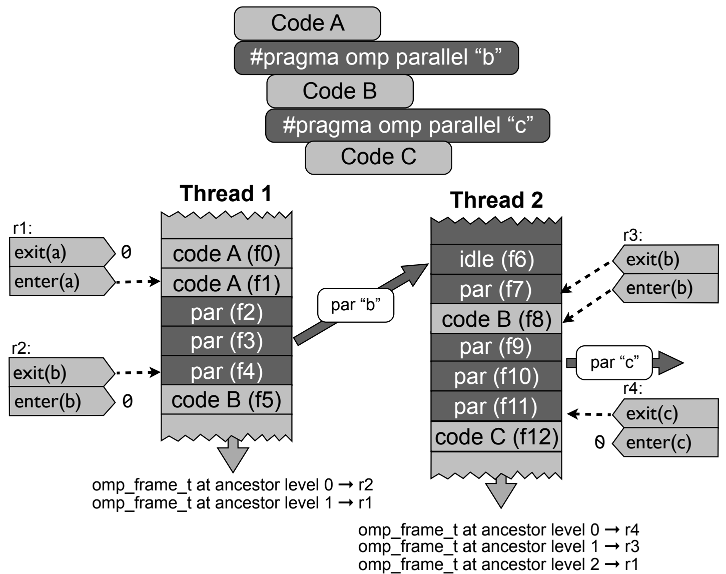


FIGURE D.1: Thread call stacks implementing nested parallelism annotated with frame information for the OMPT tool interface.

4 The top half of Figure D.1 illustrates a conceptualization of a program executing a nested parallel
5 region, where code A, B, and C represent, respectively, one or more procedure frames of code
6 associated with an initial task, an outer parallel region, and an inner parallel region. The bottom
7 half of Figure D.1 illustrates the stacks of two threads executing the nested parallel region. In the
8 illustration, stacks grow downward—a call to a function adds a new frame to the stack below the

frame of its caller. When thread 1 encounters the outer-parallel region "b", it calls a routine in the OpenMP runtime to create a new parallel region. The OpenMP runtime sets the *enter_frame* field in the **omp_frame_t** for the initial task executing code A to the canonical frame address of frame f1—the user frame in the initial task that calls the runtime. The **omp_frame_t** for the initial task is labeled *r1* in Figure D.1. In this figure, three consecutive runtime system frames, labeled "par" with frame identifiers f2–f4, are on the stack. Before starting the implicit task for parallel region "b" in thread 1, the runtime sets the *exit_frame* in the implicit task's **omp_frame_t** (labeled *r2*) to the canonical frame address of frame f4. Execution of application code for parallel region "b" begins on thread 1 when the runtime system invokes application code B (frame f5) from frame f4.

Let us focus now on thread 2, an OpenMP thread. Figure D.1 shows this worker executing work for the outer-parallel region "b." On the OpenMP thread's stack is a runtime frame labeled "idle," where the OpenMP thread waits for work. When work becomes available, the runtime system invokes a function to dispatch it. While dispatching parallel work might involve a chain of several calls, here we assume that the length of this chain is 1 (frame f7). Before thread 2 exits the runtime to execute an implicit task for parallel region "b," the runtime sets the *exit_frame* field of the implicit task's **omp_frame_t** (labeled *r3*) to the canonical frame address of frame f7. When thread 2 later encounters the inner-parallel region "c," as execution returns to the runtime, the runtime fills in the *enter_frame* field of the current task's **omp_frame_t** (labeled *r3*) to the canonical frame address of frame f8—the frame that invoked the runtime. Before the task for parallel region "c" is invoked on thread 2, the runtime system sets the *exit_frame* field of the **omp_frame_t** (labeled *r4*) for the implicit task for "c" to the canonical frame address of frame f11. Execution of application code for parallel region "c" begins on thread 2 when the runtime system invokes application code C (frame f12) from frame f11.

Below the stack for each thread in Figure D.1, the figure shows the **omp_frame_t** information obtained by calls to **ompt_get_task_info** made on each thread for the stack state shown. We show the ID of the **omp_frame_t** object returned at each ancestor level. Note that thread 2 has task frame information for three levels of tasks, whereas thread 1 has only two.

Cross References

- **omp_frame_t**, see Section 4.3.1.2 on page 556.
- **ompt_get_task_info_t**, see Section 4.1.5.1.14 on page 457.

1 **APPENDIX E**

2 # Interaction Diagram of OMPD Components

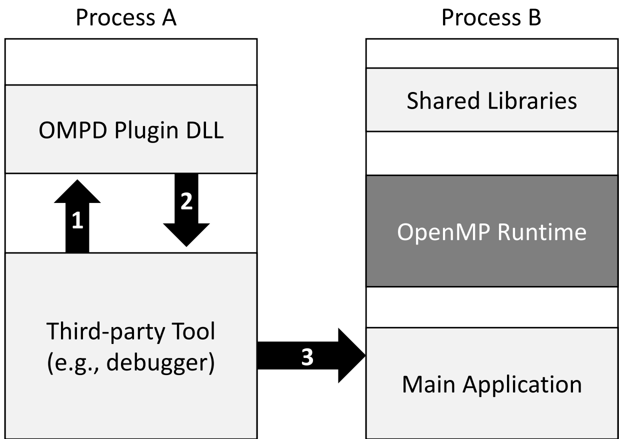


FIGURE E.1: Interaction Diagram of OMPD Components

3 The figure shows how the different components of OMPD fit together. The third-party tool loads
4 the OMPD plugin that matches the OpenMP runtime being used by the OpenMP program. The
5 plugin exports the API defined in Section 4.2 on page 476, which the tool uses to get OpenMP
6 information about the OpenMP program. The OMPD plugin will need to look up the symbols, or

read data out of the OpenMP program. It does not do this directly, but instead asks the tool to perform these operations for it using a callback interface exported by the tool.

This architectural layout insulates the tool from the details of the internal structure of the OpenMP runtime. Similarly, the OMPD plugin does not need to be concerned about how to access the OpenMP program. Decoupling the plugin and tool in this way allows for flexibility in how the OpenMP program and tool are deployed, so that, for example, there is no requirement that tool and OpenMP program execute on the same machine.

Cross References

- See Section 4.2 on page 476.

APPENDIX F

Features History

This appendix summarizes the major changes between recent versions of the OpenMP API since version 2.5.

F.1 Deprecated Features

The following features have been deprecated:

- the *nest-var* ICV;
- the **OMP_NESTED** environment variable;
- the **omp_set_nested** and **omp_get_nested** routines;
- the C/C++ type **omp_lock_hint_t** and the corresponding Fortran kind **omp_hint_hint_kind**;
- and the lock hint constants **omp_lock_hint_none**, **omp_lock_hint_uncontended**, **omp_lock_hint_contended**, **omp_lock_hint_nonspeculative**, and **omp_lock_hint_speculative**.

F.2 Version 4.5 to 5.0 Differences

- The memory model was extended to distinguish different types of flush operations according to specified flush properties (see Section 1.4.4 on page 22) and to define a happens before order based on synchronizing flush operations (see Section 1.4.5 on page 24).

- Various changes throughout the specification were made to provide initial support of C11, C++11 and C++14 (see Section 1.7 on page 28).
- Support for several features of Fortran 2003 was added (see Section 1.7 on page 28 for features that are still not supported).
- The list items allowable in a **depend** clause on a task generating construct was extended, including for C/C++ allowing any *lvalue* expression (see Section 2.1 on page 36 and Section 2.18.10 on page 225).
- The **requires** directive (see Section 2.3 on page 46) was added to support applications that require implementation-specific features generally and shared memory across devices specifically.
- The *target-offload-var* internal control variable (see Section 2.4 on page 49) and the **OMP_TARGET_OFFLOAD** environment variable (see Section 5.17 on page 578) were added to support runtime control of the execution of device constructs.
- The default value of the *nest-var* ICV was changed from *false* to implementation defined (see Section 2.4.2 on page 51). The *nest-var* ICV (see Section 2.4.1 on page 49), the **OMP_NESTED** environment variable (see Section 5.6 on page 569), and the **omp_set_nested** and **omp_get_nested** routines were deprecated (see Section 3.2.10 on page 309 and Section 3.2.11 on page 311).
- Iterators (see Section 2.6 on page 62) were added to express that an expression in a list may expand to multiple expressions.
- Predefined memory allocators (see Section 2.7 on page 64) and directives, clauses (see Section 2.14 on page 127 and API routines (see Section 3.6 on page 368) to use them were added to support different kinds of memories.
- The *relational-op* in the canonical loop form for C/C++ was extended to include != (see Section 2.9 on page 75).
- The collapse of associated loops that are imperfectly nested loops was defined for the loop (see Section 2.10.1 on page 78), **simd** (see Section 2.11.1 on page 96), **taskloop** (see Section 2.13.2 on page 114) and **distribute** (see Section 2.15.11 on page 164) constructs.
- SIMD constructs (see Section 2.11 on page 96) were extended to allow the use of **atomic** constructs within them.
- The **nontemporal** clause was added to the **simd** construct (see Section 2.11.1 on page 96).
- The **concurrent** construct was added to support compiler optimization of loops for which iterations may run in any order concurrently (see Section 2.12 on page 107).
- To support task reductions, the **task** (see Section 2.13.1 on page 110) and **target** (see Section 2.15.5 on page 141) constructs were extended to accept the the **in_reduction** clause (see Section 2.20.4.6 on page 274) and the **taskgroup** construct (see Section 2.18.6 on page 204) was extended to accept the **task_reduction** clause Section 2.20.4.5 on page 273).

- To support taskloop reductions, the **taskloop** (see Section 2.13.2 on page 114) and **taskloop simd** (see Section 2.13.3 on page 121) constructs were extended to accept the **reduction** (see Section 2.20.4.4 on page 272) and **in_reduction** (see Section 2.20.4.6 on page 274) clauses.
- To support mutually exclusive inout sets, a **mutexinoutset** *dependence-type* was added to the **depend** clause (see Section 2.13.6 on page 125 and Section 2.18.10 on page 225).
- To reduce programmer effort implicit declare target directives for some functions (C, C++, Fortran) and subroutines (Fortran) were added (see Section 2.15.5 on page 141 and Section 2.15.7 on page 150).
- Support for nested **declare target** directives was added (see Section 2.15.7 on page 150).
- The **implements** clause was added to the **declare target** directive to support the use of device-specific function implementations (see Section 2.15.7 on page 150).
- The **declare mapper** directive was added to support mapping of complicated data types (see Section 2.15.8 on page 155).
- The **depend** clause was added to the **taskwait** construct (see Section 2.18.5 on page 202).
- To support acquire and release semantics with weak memory ordering, the **acq_rel**, **acquire**, and **release** clauses were added to the **atomic** construct (see Section 2.18.7 on page 206) and **flush** construct (see Section 2.18.8 on page 215).
- The **atomic** construct was extended with the **hint** clause (see Section 2.18.7 on page 206).
- The **depend** clause (see Section 2.18.10 on page 225) was extended to support iterators.
- Lock hints were renamed to synchronization hints, and the old names were deprecated (see Section 2.18.11 on page 229).
- To support conditional assignment to lastprivate variables, the **conditional** modifier was added to the **lastprivate** clause (see Section 2.20.3.5 on page 259).
- The description of the **map** clause was modified to clarify how structure members are mapped. (see Section 2.20.6.1 on page 280).
- The capability to map pointer variables (C/C++) and assign the address of device memory that is mapped by an array section to them was added (see Section 2.20.6.1 on page 280).
- The **defaultmap** clause (see Section 2.20.6.2 on page 288) was extended to allow the **none** parameter to support the requirement that all variables all variables referenced in the construct must be explicitly mapped.
- Runtime routines (see Section 3.2.29 on page 331, Section 3.2.30 on page 332, Section 3.2.31 on page 333, and Section 3.2.32 on page 334) and environment variables (see Section 5.13 on page 574 and Section 5.14 on page 575) were added to provide OpenMP thread affinity information.

- The **omp_get_device_num** runtime routine (see Section 3.2.36 on page 338) was added to support determination of the device on which a thread is executing.
- Support for a first-party tool interface (see Section 4.1 on page 378) was added.
- Support for a third-party tool interface (see Section 4.2 on page 476) was added.
- Support for controlling offloading behavior with the **OMP_TARGET_OFFLOAD** environment variable was added (see Section 5.17 on page 578).

F.3 Version 4.0 to 4.5 Differences

- Support for several features of Fortran 2003 was added (see Section 1.7 on page 28 for features that are still not supported).
- A parameter was added to the **ordered** clause of the loop construct (see Section 2.10.1 on page 78) and clauses were added to the **ordered** construct (see Section 2.18.9 on page 221) to support doacross loop nests and use of the **simd** construct on loops with loop-carried backward dependences.
- The **linear** clause was added to the loop construct (see Section 2.10.1 on page 78).
- The **simdlen** clause was added to the **simd** construct (see Section 2.11.1 on page 96) to support specification of the exact number of iterations desired per SIMD chunk.
- The **priority** clause was added to the **task** construct (see Section 2.13.1 on page 110) to support hints that specify the relative execution priority of explicit tasks. The **omp_get_max_task_priority** routine was added to return the maximum supported priority value (see Section 3.2.41 on page 343) and the **OMP_MAX_TASK_PRIORITY** environment variable was added to control the maximum priority value allowed (see Section 5.16 on page 577).
- Taskloop constructs (see Section 2.13.2 on page 114 and Section 2.13.3 on page 121) were added to support nestable parallel loops that create OpenMP tasks.
- To support interaction with native device implementations, the **use_device_ptr** clause was added to the **target data** construct (see Section 2.15.2 on page 132) and the **is_device_ptr** clause was added to the **target** construct (see Section 2.15.5 on page 141).
- The **nowait** and **depend** clauses were added to the **target** construct (see Section 2.15.5 on page 141) to improve support for asynchronous execution of **target** regions.
- The **private**, **firstprivate** and **defaultmap** clauses were added to the **target** construct (see Section 2.15.5 on page 141).

- The **declare target** directive was extended to allow mapping of global variables to be deferred to specific device executions and to allow an *extended-list* to be specified in C/C++ (see Section 2.15.7 on page 150).
- To support unstructured data mapping for devices, the **target enter data** (see Section 2.15.3 on page 134) and **target exit data** (see Section 2.15.4 on page 137) constructs were added and the **map** clause (see Section 2.20.6.1 on page 280) was updated.
- To support a more complete set of device construct shortcuts, the **target parallel** (see Section 2.16.5 on page 175), target parallel loop (see Section 2.16.6 on page 176), target parallel loop SIMD (see Section 2.16.7 on page 178), and **target simd** (see Section 2.16.8 on page 179), combined constructs were added.
- The **if** clause was extended to take a *directive-name-modifier* that allows it to apply to combined constructs (see Section 2.17 on page 192).
- The **hint** clause was addded to the **critical** construct (see Section 2.18.2 on page 195).
- The **source** and **sink** dependence types were added to the **depend** clause (see Section 2.18.10 on page 225) to support doacross loop nests.
- The implicit data-sharing attribute for scalar variables in **target** regions was changed to **firstprivate** (see Section 2.20.1.1 on page 240).
- Use of some C++ reference types was allowed in some data sharing attribute clauses (see Section 2.20.3 on page 249).
- Semantics for reductions on C/C++ array sections were added and restrictions on the use of arrays and pointers in reductions were removed (see Section 2.20.4.4 on page 272).
- The **ref**, **val**, and **uval** modifiers were added to the **linear** clause (see Section 2.20.3.6 on page 262).
- Support was added to the map clauses to handle structure elements (see Section 2.20.6.1 on page 280).
- Query functions for OpenMP thread affinity were added (see Section 3.2.23 on page 325 to Section 3.2.28 on page 330).
- The lock API was extended with lock routines that support storing a hint with a lock to select a desired lock implementation for a lock's intended usage by the application code (see Section 3.3.2 on page 347).
- Device memory routines were added to allow explicit allocation, deallocation, memory transfers and memory associations (see Section 3.5 on page 358).
- C/C++ Grammar (previously Appendix B) was moved to a separate document.

F.4 Version 3.1 to 4.0 Differences

- Various changes throughout the specification were made to provide initial support of Fortran 2003 (see Section 1.7 on page 28).
- C/C++ array syntax was extended to support array sections (see Section 2.5 on page 60).
- The **proc_bind** clause (see Section 2.8.2 on page 73), the **OMP_PLACES** environment variable (see Section 5.5 on page 567), and the **omp_get_proc_bind** runtime routine (see Section 3.2.22 on page 323) were added to support thread affinity policies.
- SIMD constructs were added to support SIMD parallelism (see Section 2.11 on page 96).
- Device constructs (see Section 2.15 on page 131), the **OMP_DEFAULT_DEVICE** environment variable (see Section 5.15 on page 577), the **omp_set_default_device**, **omp_get_default_device**, **omp_get_num_devices**, **omp_get_num_teams**, **omp_get_team_num**, and **omp_is_initial_device** routines were added to support execution on devices.
- Implementation defined task scheduling points for untied tasks were removed (see Section 2.13.6 on page 125).
- The **depend** clause (see Section 2.18.10 on page 225) was added to support task dependences.
- The **taskgroup** construct (see Section 2.18.6 on page 204) was added to support more flexible deep task synchronization.
- The **reduction** clause (see Section 2.20.4.4 on page 272) was extended and the **declare reduction** construct (see Section 2.21 on page 289) was added to support user defined reductions.
- The **atomic** construct (see Section 2.18.7 on page 206) was extended to support atomic swap with the **capture** clause, to allow new atomic update and capture forms, and to support sequentially consistent atomic operations with a new **seq_cst** clause.
- The **cancel** construct (see Section 2.19.1 on page 232), the **cancellation point** construct (see Section 2.19.2 on page 237), the **omp_get_cancellation** runtime routine (see Section 3.2.9 on page 308) and the **OMP_CANCELLATION** environment variable (see Section 5.11 on page 572) were added to support the concept of cancellation.
- The **OMP_DISPLAY_ENV** environment variable (see Section 5.12 on page 573) was added to display the value of ICVs associated with the OpenMP environment variables.
- Examples (previously Appendix A) were moved to a separate document.

F.5 Version 3.0 to 3.1 Differences

- The **final** and **mergeable** clauses (see Section 2.13.1 on page 110) were added to the **task** construct to support optimization of task data environments.
- The **taskyield** construct (see Section 2.13.4 on page 123) was added to allow user-defined task scheduling points.
- The **atomic** construct (see Section 2.18.7 on page 206) was extended to include **read**, **write**, and **capture** forms, and an **update** clause was added to apply the already existing form of the **atomic** construct.
- Data environment restrictions were changed to allow **intent(in)** and **const**-qualified types for the **firstprivate** clause (see Section 2.20.3.4 on page 256).
- Data environment restrictions were changed to allow Fortran pointers in **firstprivate** (see Section 2.20.3.4 on page 256) and **lastprivate** (see Section 2.20.3.5 on page 259).
- New reduction operators **min** and **max** were added for C and C++
- The nesting restrictions in Section 2.22 on page 295 were clarified to disallow closely-nested OpenMP regions within an **atomic** region. This allows an **atomic** region to be consistently defined with other OpenMP regions so that they include all code in the atomic construct.
- The **omp_in_final** runtime library routine (see Section 3.2.21 on page 322) was added to support specialization of final task regions.
- The *nthreads-var* ICV has been modified to be a list of the number of threads to use at each nested parallel region level. The value of this ICV is still set with the **OMP_NUM_THREADS** environment variable (see Section 5.2 on page 565), but the algorithm for determining the number of threads used in a parallel region has been modified to handle a list (see Section 2.8.1 on page 71).
- The *bind-var* ICV has been added, which controls whether or not threads are bound to processors (see Section 2.4.1 on page 49). The value of this ICV can be set with the **OMP_PROC_BIND** environment variable (see Section 5.4 on page 566).
- Descriptions of examples (previously Appendix A) were expanded and clarified.
- Replaced incorrect use of **omp_integer_kind** in Fortran interfaces (see Section B.3 on page 617 and Section B.4 on page 625) with **selected_int_kind(8)**.

F.6 Version 2.5 to 3.0 Differences

The concept of tasks has been added to the OpenMP execution model (see Section 1.2.5 on page 10 and Section 1.3 on page 18).

- The **task** construct (see Section 2.13 on page 110) has been added, which provides a mechanism for creating tasks explicitly.
- The **taskwait** construct (see Section 2.18.5 on page 202) has been added, which causes a task to wait for all its child tasks to complete.
- The OpenMP memory model now covers atomicity of memory accesses (see Section 1.4.1 on page 20). The description of the behavior of **volatile** in terms of **flush** was removed.
- In Version 2.5, there was a single copy of the *nest-var*, *dyn-var*, *nthreads-var* and *run-sched-var* internal control variables (ICVs) for the whole program. In Version 3.0, there is one copy of these ICVs per task (see Section 2.4 on page 49). As a result, the **omp_set_num_threads**, **omp_set_nested** and **omp_set_dynamic** runtime library routines now have specified effects when called from inside a **parallel** region (see Section 3.2.1 on page 300, Section 3.2.7 on page 306 and Section 3.2.10 on page 309).
- The definition of active **parallel** region has been changed: in Version 3.0 a **parallel** region is active if it is executed by a team consisting of more than one thread (see Section 1.2.2 on page 2).
- The rules for determining the number of threads used in a **parallel** region have been modified (see Section 2.8.1 on page 71).
- In Version 3.0, the assignment of iterations to threads in a loop construct with a **static** schedule kind is deterministic (see Section 2.10.1 on page 78).
- In Version 3.0, a loop construct may be associated with more than one perfectly nested loop. The number of associated loops may be controlled by the **collapse** clause (see Section 2.10.1 on page 78).
- Random access iterators, and variables of unsigned integer type, may now be used as loop iterators in loops associated with a loop construct (see Section 2.10.1 on page 78).
- The schedule kind **auto** has been added, which gives the implementation the freedom to choose any possible mapping of iterations in a loop construct to threads in the team (see Section 2.10.1 on page 78).
- Fortran assumed-size arrays now have predetermined data-sharing attributes (see Section 2.20.1.1 on page 240).
- In Fortran, **firstprivate** is now permitted as an argument to the **default** clause (see Section 2.20.3.1 on page 250).

1 • For list items in the **private** clause, implementations are no longer permitted to use the storage
2 of the original list item to hold the new list item on the master thread. If no attempt is made to
3 reference the original list item inside the **parallel** region, its value is well defined on exit
4 from the **parallel** region (see Section 2.20.3.3 on page 252).

5 • In Version 3.0, Fortran allocatable arrays may appear in **private**, **firstprivate**,
6 **lastprivate**, **reduction**, **copyin** and **copyprivate** clauses. (see Section 2.20.2 on
7 page 244, Section 2.20.3.3 on page 252, Section 2.20.3.4 on page 256, Section 2.20.3.5 on
8 page 259, Section 2.20.4.4 on page 272, Section 2.20.5.1 on page 275 and Section 2.20.5.2 on
9 page 277).

10 • In Version 3.0, static class members variables may appear in a **threadprivate** directive (see
11 Section 2.20.2 on page 244).

12 • Version 3.0 makes clear where, and with which arguments, constructors and destructors of
13 private and threadprivate class type variables are called (see Section 2.20.2 on page 244,
14 Section 2.20.3.3 on page 252, Section 2.20.3.4 on page 256, Section 2.20.5.1 on page 275 and
15 Section 2.20.5.2 on page 277).

16 • The runtime library routines **omp_set_schedule** and **omp_get_schedule** have been
17 added; these routines respectively set and retrieve the value of the *run-sched-var* ICV (see
18 Section 3.2.12 on page 311 and Section 3.2.13 on page 313).

19 • The *thread-limit-var* ICV has been added, which controls the maximum number of threads
20 participating in the OpenMP program. The value of this ICV can be set with the
21 **OMP_THREAD_LIMIT** environment variable and retrieved with the
22 **omp_get_thread_limit** runtime library routine (see Section 2.4.1 on page 49,
23 Section 3.2.14 on page 314 and Section 5.10 on page 572).

24 • The *max-active-levels-var* ICV has been added, which controls the number of nested active
25 **parallel** regions. The value of this ICV can be set with the **OMP_MAX_ACTIVE_LEVELS**
26 environment variable and the **omp_set_max_active_levels** runtime library routine, and
27 it can be retrieved with the omp_get_max_active_levels runtime library routine (see Section 2.4.1
28 on page 49, Section 3.2.15 on page 315, Section 3.2.16 on page 317 and Section 5.9 on page 572).

29 • The *stacksize-var* ICV has been added, which controls the stack size for threads that the OpenMP
30 implementation creates. The value of this ICV can be set with the **OMP_STACKSIZE**
31 environment variable (see Section 2.4.1 on page 49 and Section 5.7 on page 570).

32 • The *wait-policy-var* ICV has been added, which controls the desired behavior of waiting threads.
33 The value of this ICV can be set with the **OMP_WAIT_POLICY** environment variable (see
34 Section 2.4.1 on page 49 and Section 5.8 on page 571).

35 • The **omp_get_level** runtime library routine has been added, which returns the number of
36 nested **parallel** regions enclosing the task that contains the call (see Section 3.2.17 on
37 page 318).

38 • The **omp_get_ancestor_thread_num** runtime library routine has been added, which

returns, for a given nested level of the current thread, the thread number of the ancestor (see Section 3.2.18 on page 319).

- The **omp_get_team_size** runtime library routine has been added, which returns, for a given nested level of the current thread, the size of the thread team to which the ancestor belongs (see Section 3.2.19 on page 320).
- The **omp_get_active_level** runtime library routine has been added, which returns the number of nested, active **parallel** regions enclosing the task that contains the call (see Section 3.2.20 on page 321).
- In Version 3.0, locks are owned by tasks, not by threads (see Section 3.3 on page 344).

Index

Symbols
_OPENMP macro, 571–573
_OPENMP macro, 44

A
affinity, 73
allocate, 127, 130
array sections, 60
atomic, 206
atomic construct, 627
attribute clauses, 249
attributes, data-mapping, 279
attributes, data-sharing, 239
auto, 83

B
barrier, 198
barrier, implicit, 200

C
C/C++ stub routines, 580
cancel, 232
cancellation constructs, 232
    cancel, 232
    cancellation point, 237
cancellation point, 237
canonical loop form, 75
capture, atomic, 206
clauses
    allocate, 130
    attribute data-sharing, 249
    collapse, 78, 80
    copyin, 275
    copyprivate, 277
    data copying, 275
    data-sharing, 249
    default, 250
    defaultmap, 288
    depend, 225
    firstprivate, 256
    hint, 229
    if Clause, 192
    in_reduction, 274
    lastprivate, 259
    linear, 262
    map, 280
    private, 252
    reduction, 272
    schedule, 80
    shared, 251
    task_reduction, 273
combined constructs, 169
    parallel loop construct, 169
    parallel loop SIMD construct, 173
    parallel sections, 171
    parallel workshare, 172
    target parallel, 175
    target parallel loop, 176
    target parallel loop SIMD, 178
    target simd, 179
    target teams, 180
    target teams distribute, 184
    target teams distribute parallel loop
        construct, 188
    target teams distribute parallel loop
        SIMD construct, 191
    target teams distribute simd,
        185
    teams distribute, 182
    teams distribute parallel loop construct,
        187
    teams distribute parallel loop SIMD
        construct, 189

teams distribute simd, 183
compilation sentinels, 45
compliance, 28
concurrent, 107
conditional compilation, 44
constructs
atomic, 206
barrier, 198
cancel, 232
cancellation constructs, 232
cancellation point, 237
combined constructs, 169
concurrent, 107
critical, 195
declare mapper, 155
declare target, 150
device constructs, 131
distribute, 160
distribute parallel do, 165
distribute parallel do simd, 167
distribute parallel for, 165
distribute parallel for simd, 167
distribute parallel loop, 165
distribute parallel loop SIMD, 167
distribute simd, 164
do *Fortran*, 78
flush, 215
for, *C/C++*, 78
loop, 78
Loop SIMD, 105
master, 194
ordered, 221
parallel, 66
parallel do *Fortran*, 169
parallel for *C/C++*, 169
parallel loop construct, 169
parallel loop SIMD construct, 173
parallel sections, 171
parallel workshare, 172
sections, 86
simd, 96
single, 90
target, 141
target data, 132
target enter data, 134
target exit data, 137
target parallel, 175
target parallel do, 176
target parallel do simd, 178
target parallel for, 176
target parallel for simd, 178
target parallel loop, 176
target parallel loop SIMD, 178
target simd, 179
target teams, 180
target teams distribute, 184
target teams distribute parallel loop construct, 188
target teams distribute parallel loop SIMD construct, 191
target teams distribute simd, 185
target update, 146
task, 110
taskgroup, 204
tasking constructs, 110
taskloop, 114
taskloop simd, 121
taskwait, 202
taskyield, 123
teams, 157
teams distribute, 182
teams distribute parallel loop construct, 187
teams distribute parallel loop SIMD construct, 189
teams distribute simd, 183
workshare, 93
worksharing, 77
controlling OpenMP thread affinity, 73
copyin, 275
copyprivate, 277
critical, 195

D
data copying clauses, 275
data environment, 239
data terminology, 12
data-mapping rules and clauses, 279
data-sharing attribute clauses, 249
data-sharing attribute rules, 239
declare mapper, 155
declare reduction, 289
declare simd, 100
declare target, 150
default, 250
defaultmap, 288
depend, 225
deprecated features, 637
device constructs, 131
  declare mapper, 155
  declare target, 150
  device constructs, 131
  distribute, 160
  distribute parallel loop, 165
  distribute parallel loop SIMD, 167
  distribute simd, 164
  target, 141
  target update, 146
  teams, 157
device data environments, 22, 134, 137
device memory routines, 358
directive format, 36
directives, 35
  allocate, 127
  declare mapper, 155
  declare reduction, 289
  declare simd, 100
  declare target, 150
  memory management directives, 127
  requires, 46
  threadprivate, 244
distribute, 160
distribute parallel loop construct, 165
distribute parallel loop SIMD construct, 167
distribute simd, 164
do, *Fortran*, 78
do simd, 105
dynamic, 82
dynamic thread adjustment, 625

E
environment variables, 561
  OMP_AFFINITY_FORMAT, 573
  OMP_ALLOCATOR, 578
  OMP_CANCELLATION, 570
  OMP_DEFAULT_DEVICE, 575
  OMP_DISPLAY_AFFINITY, 572
  OMP_DISPLAY_ENV, 571
  OMP_DYNAMIC, 564
  OMP_MAX_ACTIVE_LEVELS, 570
  OMP_MAX_TASK_PRIORITY, 575
  OMP_NESTED, 567
  OMP_NUM_THREADS, 563
  OMP_PLACES, 565
  OMP_PROC_BIND, 564
  OMP_SCHEDULE, 562
  OMP_STACKSIZE, 568
  OMP_TARGET_OFFLOAD, 576
  OMP_THREAD_LIMIT, 570
  OMP_TOOL, 577
  OMP_TOOL_LIBRARIES, 577
  OMP_WAIT_POLICY, 569
  OMPD_ENABLED, 578
event callback registration, 381
event callback signatures, 406
execution environment routines, 300
execution model, 18

F
features history, 637
firstprivate, 256
fixed source form conditional compilation sentinels, 45
fixed source form directives, 39
flush, 215
flush operation, 22
flush synchronization, 24
flush-set, 22
for, *C/C++*, 78
for simd, 105

frames, 555
free source form conditional compilation sentinel, 45
free source form directives, 40

G
glossary, 2
guided, 82

H
happens before, 24
header files, 298, 603
history of features, 637

I
ICVs (internal control variables), 49
if Clause, 192
implementation, 625
implementation terminology, 15
implicit barrier, 200
in_reduction, 274
include files, 298, 603
interface declarations, 603
internal control variables, 625
internal control variables (ICVs), 49
introduction, 1
iterators, 62

L
lastprivate, 259
linear, 262
lock routines, 344
loop, 78
loop SIMD Construct, 105
loop terminology, 8

M
map, 280
master, 194
master and synchronization constructs and clauses, 193
memory allocators, 64
memory management, 127
memory management directives
memory management directives, 127
memory management routines, 368
memory model, 20
modification order, 20
modifying and retrieving ICV values, 53
modifying ICV's, 51

N
nesting of regions, 295
normative references, 28

O
omp_get_num_teams, 339
OMP_AFFINITY_FORMAT, 573
omp_alloc, 371
OMP_ALLOCATOR, 578
OMP_CANCELLATION, 570
omp_capture_affinity, 334
OMP_DEFAULT_DEVICE, 575
omp_destroy_lock, 349
omp_destroy_nest_lock, 349
OMP_DISPLAY_AFFINITY, 572
omp_display_affinity, 333
OMP_DISPLAY_ENV, 571
OMP_DYNAMIC, 564
omp_free, 372
omp_get_active_level, 321
omp_get_affinity_format, 332
omp_get_ancestor_thread_num, 319
omp_get_cancellation, 308
omp_get_default_allocator, 370
omp_get_default_device, 337
omp_get_device_num, 338
omp_get_dynamic, 308
omp_get_initial_device, 342
omp_get_level, 318
omp_get_max_active_levels, 317
omp_get_max_task_priority, 343
omp_get_max_threads, 302
omp_get_nested, 311
omp_get_num_devices, 337
omp_get_num_places, 325
omp_get_num_procs, 305

omp_get_num_threads, 301
omp_get_partition_num_places, 329
omp_get_partition_place_nums, 330
omp_get_place_num, 328
omp_get_place_num_procs, 326
omp_get_place_proc_ids, 327
omp_get_proc_bind, 323
omp_get_schedule, 313
omp_get_team_num, 340
omp_get_team_size, 320
omp_get_thread_limit, 314
omp_get_thread_num, 304
omp_get_wtick, 357
omp_get_wtime, 356
omp_in_final, 322
omp_in_parallel, 305
omp_init_lock, 346, 347
omp_init_nest_lock, 346, 347
omp_is_initial_device, 341
OMP_MAX_ACTIVE_LEVELS, 570
OMP_MAX_TASK_PRIORITY, 575
OMP_NESTED, 567
OMP_NUM_THREADS, 563
OMP_PLACES, 565
OMP_PROC_BIND, 564
OMP_SCHEDULE, 562
omp_set_affinity_format, 331
omp_set_default_allocator, 369
omp_set_default_device, 336
omp_set_dynamic, 306
omp_set_lock, 350
omp_set_max_active_levels, 315
omp_set_nest_lock, 350
omp_set_nested, 309
omp_set_num_threads, 300
omp_set_schedule, 311
OMP_STACKSIZE, 568
omp_target_alloc, 358
omp_target_associate_ptr, 365
omp_target_disassociate_ptr, 367
omp_target_free, 360
omp_target_is_present, 361
omp_target_memcpy, 362
omp_target_memcpy_rect, 363
OMP_TARGET_OFFLOAD, 576
omp_test_lock, 354
omp_test_nest_lock, 354
OMP_THREAD_LIMIT, 570
OMP_TOOL, 577
OMP_TOOL_LIBRARIES, 577
omp_unset_lock, 352
omp_unset_nest_lock, 352
OMP_WAIT_POLICY, 569
ompd diagram, 635
ompd_bp_parallel_begin, 544
ompd_bp_parallel_end, 546
ompd_bp_task_begin, 547
ompd_bp_task_end, 548
ompd_callback_device_host_fn_t, 497
ompd_callback_get_address_space_conte for_thread_fn_t, 490
ompd_callback_get_process_context_for context_fn_t, 491
ompd_callback_get_thread_context_for_ thread_id_fn_t, 490
ompd_callback_memory_alloc_fn_t, 488
ompd_callback_memory_free_fn_t, 489
ompd_callback_memory_read_fn_t, 495
ompd_callback_memory_write_fn_t, 496
ompd_callback_print_string_fn_t, 498
ompd_callback_sizeof_fn_t, 493
ompd_callback_symbol_addr_fn_t, 493
ompd_callbacks_t, 499

ompd_dll_locations_valid, 560
ompd_dll_locations, 559
ompd_enable, 549
OMPD_ENABLED, 578
ompt_callback_buffer_complete_t, 435
ompt_callback_buffer_request_t, 434
ompt_callback_cancel_t, 437
ompt_callback_control_tool_t, 436
ompt_callback_device_finalize_t, 440
ompt_callback_device_initialize_t, 438
ompt_callback_flush_t, 424
ompt_callback_idle_t, 407
ompt_callback_implicit_task_t, 416
ompt_callback_master_t, 411
ompt_callback_mutex_acquire_t, 419
ompt_callback_mutex_t, 420
ompt_callback_nest_lock_t, 421
ompt_callback_parallel_begin_t, 408
ompt_callback_parallel_end_t, 410
ompt_callback_sync_region_t, 418
ompt_callback_device_load_t, 426
ompt_callback_device_unload_t, 428
ompt_callback_target_data_op_t, 428
ompt_callback_target_map_t, 431
ompt_callback_target_submit_t, 432
ompt_callback_target_t, 425
ompt_callback_task_create_t, 412
ompt_callback_task_dependence_t, 414
ompt_callback_task_dependences_t, 413
ompt_callback_task_schedule_t, 415
ompt_callback_thread_begin_t, 406
ompt_callback_thread_end_t, 407
ompt_callback_work_t, 423
OpenMP compliance, 28
ordered, 221

P
parallel, 66
parallel loop construct, 169
parallel loop SIMD construct, 173
parallel sections, 171
parallel workshare, 172
private, 252

R
read, atomic, 206
reduction, 272
reduction clauses, 265
requires, 46
runtime library definitions, 298
runtime library routines, 297

S
scheduling, 125
sections, 86
shared, 251
simd, 96
SIMD Constructs, 96
Simple Lock Routines, 344
single, 90
stand-alone directives, 43
stub routines, 580
sync-set, 22
synchronization constructs, 193
synchronization hints, 229
synchronization terminology, 9

T
target, 141

target data, 132
target memory routines, 358
target parallel, 175
target parallel loop construct, 176
target parallel loop SIMD construct, 178
target simd, 179
target teams, 180
target teams distribute, 184
target teams distribute parallel loop
construct, 188
target teams distribute parallel loop SIMD
construct, 191
target teams distribute simd, 185
target update, 146
task, 110
task frame management, 633
task scheduling, 125
task_reduction, 273
taskgroup, 204
tasking constructs, 110
tasking terminology, 10
taskloop, 114
taskloop simd, 121
taskwait, 202
taskyield, 123
teams, 157
teams distribute, 182
teams distribute parallel loop construct, 187
teams distribute parallel loop SIMD
construct, 189
teams distribute simd, 183
thread affinity, 73
threadprivate, 244
timer, 356
timing routines, 356
tool control, 372
tool initialization, 379
tool support, 377
tracing device activity, 383

U
update, atomic, 206

V
variables, environment, 561

W
wait identifier, 557
wall clock timer, 356
workshare, 93
worksharing
constructs, 77
parallel, 169
scheduling, 86
worksharing constructs, 77
write, atomic, 206