OpenMP® Technical Report 4: Version 5.0 Preview 1

This Technical Report augments the OpenMP API Specification, version 4.5, with language features for task reductions, defines a runtime interface for performance and correctness tools (OMPT), extensions to the target constructs, and contains several clarifications and fixes.

All members of the OpenMP Language Working Group

November 10, 2016

Expires November 9, 2018

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

End of Public Comment Period: January 9, 2017

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

This technical report describes possible future directions or extensions to the OpenMP API 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 provision 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 1, November 2016

This page intentionally left blank in published version.

This is Revision 1-TR4 (Initial Official Draft) (10 November 2016) and includes the following internal tickets applied to the 4.5 LaTeX sources: 354, 399, 425, 452, 461, 463, 465-467, 484, 489, 492, 498-500, 502-504, 510, 520, 523, 530, 531, 551

This is a draft; contents will change in official release

Contents

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 runtime system.

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 23 for a listing of current *base languages* for the OpenMP API.

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

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

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

COMMENTS:

For all *base languages*:

- Access to the *structured block* must not be the result of a branch; and
- The point of exit cannot be a branch out of the *structured block*.

For C/C++:

- The point of entry must not be a call to **`setjmp()`**;
- **`longjmp()`** and **`throw()`** must not violate the entry/exit criteria;
- Calls to **`exit()`** are allowed in a *structured block*; and
- An expression statement, iteration statement, selection statement, or try block is considered to be a *structured block* if the corresponding compound statement obtained by enclosing it in **`{`** and **`}`** would be a *structured block*.

For Fortran:

- **`STOP`** statements are allowed in a *structured block*.

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

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

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

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

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

OpenMP program A program that consists of a *base program*, annotated with OpenMP *directives* and 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 *thread 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*.

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 2.17 on page 256 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*, or the *generating task*.

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

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

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

COMMENTS:

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

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

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

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

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

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

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

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

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

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

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

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

For Fortran, *worksharing constructs* are **`do`**, **`sections`**, **`single`** and **`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*.

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 2.14.1 on page 197

1.2.5 Tasking Terminology

task A specific instance of executable code and its *data environment*, generated when a *thread* encounters a **`task`**, **`taskloop`**, **`parallel`**, **`target`**, or **`teams`** *construct* (or any *combined construct* that specifies any of these *constructs*).

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.

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 2.9.6 on page 104.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

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.

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.

1.2.8 Tool Terminology

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

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

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

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

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

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

dispatching a callback 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.

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

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

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

canonical frame 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 frame represents.

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

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

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

signal A software interrupt delivered to a thread.

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*.

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 enclosed in an implicit task region, called an initial task region, that is defined 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 enclosed in an implicit task region, called an initial task region, that is defined by an implicit **parallel** region that surrounds the entire **target** 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 thread teams* where the master thread of each team executes the region. Each of these master threads is an initial thread, and executes sequentially, as if enclosed in an implicit task region that is defined by an implicit parallel region that surrounds the entire **teams** region.

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.

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 2.15.3.3 on page 218 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 2.15 on page 204.

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.

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 1.4.4 on page 21. 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.

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 2.15.6.1 on page 245.

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 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*. The flush operation restricts reordering of memory operations that an implementation might otherwise do. Implementations must not reorder the code for a memory operation for a given variable, or the code for a flush operation for the variable, with respect to a flush operation that refers to the same variable.

If a thread has performed a write to its temporary view of a shared variable since its last flush of that variable, then when it executes another 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 flushes of that variable, the flush ensures that the value of the last write is written to the variable in memory. A flush of a variable executed by a thread also causes its temporary view of the variable to be discarded, so that if its next memory operation 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 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 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 to all other threads and that thread's temporary view of all variables involved is discarded.

The flush operation provides 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, the programmer must 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 happens in the specified order:

1. The value is written to the variable by the first thread.
2. The variable is flushed by the first thread.
3. The variable is flushed by the second thread.
4. The value is read from the variable by the second thread.

Note – OpenMP synchronization operations, described in Section 2.13 on page 165 and in Section 3.3 on page 301, are recommended for enforcing this order. Synchronization through variables is possible but is not recommended because the proper timing of flushes is difficult.

1.4.4 OpenMP Memory Consistency

The restrictions in Section 1.4.3 on page 19 on reordering with respect to flush operations guarantee the following:

- If the intersection of the flush-sets of two 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 either access, modify, or flush the same variable, then they must be completed as if in that thread's program order, as seen by all threads.
- If the intersection of the flush-sets of two flushes is empty, the threads can observe these flushes in any order.

The flush operation can be specified using the **`flush`** directive, and is also implied at various locations in an OpenMP program: see Section 2.13.8 on page 186 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 18

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, *first-party* tools that support monitoring and performance analysis of OpenMP programs developed using any implementation of the OpenMP API, the OpenMP API includes a tool interface known as OMPT.

The OMPT interface 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 4.2.3 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.

1.6 OpenMP Compliance

An implementation of the OpenMP API is compliant if and only if it compiles and executes all 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 25) are for information purposes only and are not part of the specification.

The OpenMP API defines constructs that operate in the context of the base language that is supported by an implementation. If 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

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

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

- 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
  - Polymorphic entities
  - **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 E "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++

C/C++ specific text...

C / C++

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

C

C specific text...

C

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

C90

C90 specific text...

C90

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

C99

C99 specific text...

C99

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

C++

C++ specific text...

C++

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

Fortran

Fortran specific text......

Fortran

Where an entire page consists of, for example, Fortran specific text, a marker is shown at the top of the page like this:

Fortran (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....

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 28)
- Mechanisms to control conditional compilation (Section 2.2 on page 36)
- Control of OpenMP API ICVs (Section 2.3 on page 39)
- How to specify and to use array sections for all base languages (Section 2.4 on page 48)
- Details of each OpenMP directive, including associated events and tool callbacks (Section 2.5 on page 50 to Section 2.17 on page 256)

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 31 and Section 2.1.2 on page 32.

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 2.11 on page 140). 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 2.15.3 on page 215), data copying clauses (Section 2.15.5 on page 240), the **threadprivate** directive (Section 2.15.2 on page 210), the **flush** directive (Section 2.13.8 on page 186), and the **link** clause of the **declare target** directive (Section 2.10.7 on page 124) accept a *list*. The **to** clause of the **declare target** directive (Section 2.10.7 on page 124) accepts an *extended-list*. The **depend** clause (Section 2.13.10 on page 194), 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

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

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

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

Fortran

For all base languages, a *list item* or an *extended list item* is subject to the restrictions specified in Section 2.4 on page 48 and in each of the sections describing clauses and directives for which the *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:

```
declare reduction
declare simd
declare target
distribute parallel do
distribute parallel do simd
distribute simd
do simd
end atomic
end critical
end distribute
end distribute parallel do
end distribute parallel do simd
```

```
end distribute simd
end do
end do simd
end master
end ordered
end parallel
end parallel do
end parallel do simd
end parallel sections
end parallel workshare
end sections
end simd
end single
end target
end target data
end target parallel
end target parallel do
end target parallel do simd
end target simd
end target teams
end target teams distribute
end target teams distribute parallel do
end target teams distribute parallel do simd
end target teams distribute simd
end task
end taskgroup
end taskloop
```

```
end taskloop simd
end teams
end teams distribute
end teams distribute parallel do
end teams distribute parallel do simd
end teams distribute simd
end workshare
parallel do
parallel do simd
parallel sections
parallel workshare
target data
target enter data
target exit data
target parallel
target parallel do
target parallel do simd
target simd
target teams
target teams distribute
target teams distribute parallel do
target teams distribute parallel do simd
target teams distribute simd
target update
taskloop simd
teams distribute
teams distribute parallel do
```

```
teams distribute parallel do simd
teams distribute simd
```

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 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 2.3.2 on page 40.

2.3.1 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.
- *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.
- *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.
- *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 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.

2.3.2 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**	*false*
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
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*
default-device-var	**OMP_DEFAULT_DEVICE**	Implementation defined
max-task-priority-var	**OMP_MAX_TASK_PRIORITY**	*zero*
tool-var	**OMP_TOOL**	*enabled*
tool-libraries-var	**OMP_TOOL_LIBRARIES**	*empty string*

Description

- Each device has its own ICVs.
- The value of the *nthreads-var* ICV is a list.
- The value of the *bind-var* ICV is a list.
- The initial value of *dyn-var* is implementation defined if the implementation supports dynamic adjustment of the number of threads; otherwise, the initial value is *false*.
- The initial value of *max-active-levels-var* is the number of levels of parallelism that the implementation supports. See the definition of *supporting n levels of parallelism* in Section 1.2.7 on page 13 for further details.

The host and target device ICVs are initialized before any OpenMP API construct or OpenMP API routine executes. After the initial values are assigned, the values of any OpenMP environment variables that were set by the user are read and the associated ICVs for the host device are modified 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 434.
- **OMP_NUM_THREADS** environment variable, see Section 5.2 on page 435.
- **OMP_DYNAMIC** environment variable, see Section 5.3 on page 436.
- **OMP_PROC_BIND** environment variable, see Section 5.4 on page 436.
- **OMP_PLACES** environment variable, see Section 5.5 on page 437.
- **OMP_NESTED** environment variable, see Section 5.6 on page 439.
- **OMP_STACKSIZE** environment variable, see Section 5.7 on page 440.
- **OMP_WAIT_POLICY** environment variable, see Section 5.8 on page 441.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 5.9 on page 442.
- **OMP_THREAD_LIMIT** environment variable, see Section 5.10 on page 442.
- **OMP_CANCELLATION** environment variable, see Section 5.11 on page 442.
- **OMP_DEFAULT_DEVICE** environment variable, see Section 5.13 on page 444.
- **OMP_MAX_TASK_PRIORITY** environment variable, see Section 5.14 on page 445.
- **OMP_TOOL** environment variable, see Section 5.15 on page 445.
- **OMP_TOOL_LIBRARIES** environment variable, see Section 5.16 on page 446.

2.3.3 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()**
default-device-var	**omp_set_default_device()**	**omp_get_default_device()**
max-task-priority-var	(none)	**omp_get_max_task_priority()**
tool-var	(none)	(none)
tool-libraries-var	(none)	(none)

Description

- The value of the *nthreads-var* ICV is a list. The runtime call **omp_set_num_threads()** sets the value of the first element of this list, and **omp_get_max_threads()** retrieves the value of the first element of this list.
- The value of the *bind-var* ICV is a list. The runtime call **omp_get_proc_bind()** retrieves 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 2.10.8 on page 129.

- **omp_set_num_threads** routine, see Section 3.2.1 on page 262.
- **omp_get_max_threads** routine, see Section 3.2.3 on page 264.
- **omp_set_dynamic** routine, see Section 3.2.7 on page 268.
- **omp_get_dynamic** routine, see Section 3.2.8 on page 270.
- **omp_get_cancellation** routine, see Section 3.2.9 on page 271.
- **omp_set_nested** routine, see Section 3.2.10 on page 271.
- **omp_get_nested** routine, see Section 3.2.11 on page 273.
- **omp_set_schedule** routine, see Section 3.2.12 on page 274.
- **omp_get_schedule** routine, see Section 3.2.13 on page 276.
- **omp_get_thread_limit** routine, see Section 3.2.14 on page 277.
- **omp_set_max_active_levels** routine, see Section 3.2.15 on page 277.
- **omp_get_max_active_levels** routine, see Section 3.2.16 on page 279.
- **omp_get_level** routine, see Section 3.2.17 on page 280.
- **omp_get_active_level** routine, see Section 3.2.20 on page 283.
- **omp_get_proc_bind** routine, see Section 3.2.22 on page 285.
- **omp_get_place_num_procs()** routine, see Section 3.2.24 on page 288.
- **omp_get_place_proc_ids()** routine, see Section 3.2.25 on page 289.
- **omp_get_partition_num_places()** routine, see Section 3.2.27 on page 291.
- **omp_get_partition_place_nums()** routine, see Section 3.2.28 on page 292.
- **omp_set_default_device** routine, see Section 3.2.29 on page 293.
- **omp_get_default_device** routine, see Section 3.2.30 on page 294.
- **omp_get_max_task_priority** routine, see Section 3.2.36 on page 299.

2.3.4 How ICVs are Scoped

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
default-device-var	data environment
max-task-priority-var	global
tool-var	global
tool-libraries-var	global

Description

- There is one copy per device of each ICV with device scope
- Each data environment has its own copies of ICVs with data environment scope
- Each implicit task has its own copy of ICVs with implicit task scope

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

2.3.4.1 How the Per-Data Environment ICVs Work

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

When a **task** construct is encountered, the generated task inherits the value of *nthreads-var* from the generating task's *nthreads-var* value. When a **parallel** construct is encountered, and the generating task's *nthreads-var* list contains a single element, the generated task(s) inherit that list as the value of *nthreads-var*. When a **parallel** construct is encountered, and the generating task's *nthreads-var* list contains multiple elements, the generated task(s) inherit the value of *nthreads-var* as the list obtained by deletion of the first element from the generating task's *nthreads-var* value. 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.

2.3.5 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)

table continued on next page

table continued from previous page

ICV	construct clause, if used
wait-policy-var	(none)
thread-limit-var	(none)
max-active-levels-var	(none)
active-levels-var	(none)
levels-var	(none)
place-partition-var	(none)
cancel-var	(none)
default-device-var	(none)
max-task-priority-var	(none)
tool-var	(none)
tool-libraries-var	(none)

Description

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

Cross References

- **parallel** construct, see Section 2.5 on page 50.
- **proc_bind** clause, Section 2.5 on page 50.
- **num_threads** clause, see Section 2.5.1 on page 55.
- Loop construct, see Section 2.7.1 on page 62.
- **schedule** clause, see Section 2.7.1.1 on page 70.

2.4 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]
```

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 last example is not contiguous.

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++

- An array section can only be specified for a base language identifier.

C / C++

C

- The type of the variable appearing in an array section must be array or pointer.

C

C++

- If the type of the variable appearing in 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

2.5 `parallel` Construct

Summary

This fundamental construct starts parallel execution. See Section 1.3 on page 15 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)`

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)**

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 2.5.1 on page 55 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 2.5.2 on page 57, 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 2.9 on page 91).

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 2.5.1 on page 55, 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 2.13.4 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 parent 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.

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

A thread dispatches a registered **ompt_callback_thread_end** callback for the *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**.

Restrictions

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

- A program that branches into or out of a **parallel** region is non-conforming.
- A program must not depend on any ordering of the evaluations of the clauses of the **parallel** 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 **proc_bind** clause can appear on the directive.
- At most one **num_threads** clause can appear on the directive. The **num_threads** expression must evaluate to a positive integer value.

C / C++

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

C / C++

Fortran

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

Fortran

Cross References

- **if** clause, see Section 2.12 on page 164.
- **default**, **shared**, **private**, **firstprivate**, and **reduction** clauses, see Section 2.15.3 on page 215.
- **copyin** clause, see Section 2.15.5 on page 240.
- **omp_get_thread_num** routine, see Section 3.2.4 on page 266.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.4.6.11 on page 356.
- **ompt_callback_thread_begin_t**, see Section 4.6.2.1 on page 366.
- **ompt_callback_thread_end_t**, see Section 4.6.2.2 on page 367.
- **ompt_callback_idle_t**, see Section 4.6.2.3 on page 368.
- **ompt_callback_parallel_begin_t**, see Section 4.6.2.4 on page 369.
- **ompt_callback_parallel_end_t**, see Section 4.6.2.5 on page 370.
- **ompt_callback_implicit_task_t**, see Section 4.6.2.11 on page 377.

2.5.1 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*;

let *ThreadsAvailable* = (*thread-limit-var* - *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 2.3 on page 39.

2.5.2 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.

2.6 Canonical Loop Form

C / C++

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.

relational-op	One of the following:
	`<`
	`<=`
	`>`
	`>=`

lb and *b*	Loop invariant expressions of a type compatible with the type of *var*.

continued on next page

continued from previous page

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.
- 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.

C / C++

2.7 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

2.7.1 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:

private(*list***)**

firstprivate(*list***)**

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

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

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

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

collapse(*n***)**

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

nowait

The **for** directive places restrictions on the structure of all associated *for-loops*. Specifically, all associated *for-loops* must have *canonical loop form* (see Section 2.6 on page 58).

C / C++

Fortran

The syntax of the loop construct is as follows:

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

where *clause* is one of the following:

private(*list***)**

firstprivate(*list***)**

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

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

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

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

collapse(*n***)**

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

If an **end do** directive is not specified, an **end do** 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 do** 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 loop construct.

Fortran

Binding

The binding thread set for a loop region is the current team. A loop 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 loop iterations and the implied barrier of the loop region if the barrier is not eliminated by a **nowait** clause.

Description

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

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

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

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 worksharing 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 a set of associated loop(s) were executed sequentially. The **schedule** clause specifies how iterations of these associated loops are divided into contiguous non-empty subsets, called chunks, and how these chunks are distributed among threads of the team. Each thread executes its assigned chunk(s) in the context of its implicit task. The iterations of a given chunk are executed in sequential order by the assigned thread. The *chunk_size* expression is evaluated using the original list items of any variables that are made private in the loop construct. It is unspecified whether, in what order, or how many times, any side effects of the evaluation of this expression occur. The use of a variable in a **schedule** clause expression of a loop construct causes an implicit reference to the variable in all enclosing constructs.

Different loop regions with the same schedule and iteration count, even if they occur in the same parallel region, can distribute iterations among threads differently. The only exception is for the **static** schedule as specified in Table 2.5. Programs that depend on which thread executes a particular iteration under any other circumstances are non-conforming.

See Section 2.7.1.1 on page 70 for details of how the schedule for a worksharing loop is determined.

The schedule *kind* can be one of those specified in Table 2.5.

The schedule *modifier* can be one of those specified in Table 2.6. If the **static** schedule kind is specified or if the **ordered** clause is specified, and if the **nonmonotonic** modifier is not specified, the effect is as if the **`monotonic`** modifier is specified. Otherwise, unless the **`monotonic`** modifier is specified, the effect is as if the **`nonmonotonic`** modifier is specified.

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

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

TABLE 2.5: `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. 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.

table continued on next page

table continued from previous page

`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). 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.

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

A compliant implementation of the **guided** schedule with a *chunk_size* value of k would assign $q = \lceil n/p \rceil$ iterations to the first available thread and set n to the larger of $n - q$ and $p * k$. It would then repeat this process until q is greater than or equal to the number of remaining iterations, at which time the remaining iterations form the final chunk. Another compliant implementation could 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 2.6: 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.
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.

Events

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

The *loop-end* event occurs after a **loop** 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 *loop-begin* and *loop-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_loop** as its *wstype* argument.

Restrictions

Restrictions to the loop construct are as follows:

- All loops associated with the loop construct must be perfectly nested; that is, there must be no intervening code nor any OpenMP directive between any two loops.
- The values of the loop control expressions of the loops associated with the loop construct must be the same for all threads in the team.
- Only one **schedule** clause can appear on a loop directive.
- Only one **collapse** clause can appear on a loop directive.
- *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

Cross References

- **private**, **firstprivate**, **lastprivate**, **linear**, and **reduction** clauses, see Section 2.15.3 on page 215.
- **OMP_SCHEDULE** environment variable, see Section 5.1 on page 434.
- **ordered** construct, see Section 2.13.9 on page 190.
- **depend** clause, see Section 2.13.10 on page 194.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.4.6.11 on page 356.
- **ompt_work_loop**, see Section 4.4.6.14 on page 357.
- **ompt_callback_work_t**, see Section 4.6.2.18 on page 385.

2.7.1.1 Determining the Schedule of a Worksharing Loop

When execution encounters a loop directive, the **schedule** clause (if any) on the directive, and the *run-sched-var* and *def-sched-var* ICVs are used to determine how loop iterations are assigned to threads. See Section 2.3 on page 39 for details of how the values of the ICVs are determined. If the loop directive does not have a **schedule** clause then the current value of the *def-sched-var* ICV determines the schedule. If the loop directive has a **schedule** clause that specifies the **runtime** schedule kind then the current value of the *run-sched-var* ICV determines the schedule. Otherwise, the value of the **schedule** clause determines the schedule. Figure 2.1 describes how the schedule for a worksharing loop is determined.

Cross References

- ICVs, see Section 2.3 on page 39

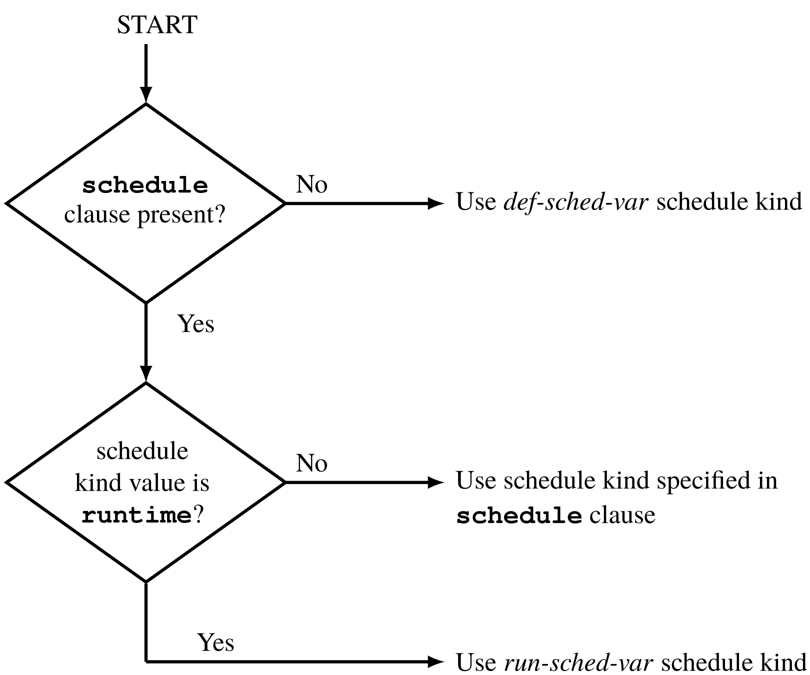

FIGURE 2.1: Determining the **schedule** for a Worksharing Loop

2.7.2 `sections` Construct

Summary

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

Syntax

C / C++

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]
   ...
   }
```

where *clause* is one of the following:

private(*list***)**

firstprivate(*list***)**

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

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

nowait

C / C++

Fortran

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]
```

where *clause* is one of the following:

private(*list***)**

firstprivate(*list***)**

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

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

Fortran

Binding

The binding thread set for a **sections** region is the current team. A **sections** 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 blocks and the implied barrier of 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 2.15.3 on page 215.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.4.6.11 on page 356.
- **ompt_work_sections**, see Section 4.4.6.14 on page 357.
- **ompt_callback_work_t**, see Section 4.6.2.18 on page 385.

2.7.3 `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`

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***`)`**

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

The method of choosing a thread to execute the structured block 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.

Tool Callbacks

A thread dispatches a registered **ompt_callback_work** callback for each occurrence of *single-begin* and *single-end* events in that thread. 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_single_executor** or **ompt_work_single_other** as its *wstype* argument.

Restrictions

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

- The **copyprivate** clause must not be used with the **nowait** clause.
- At most one **nowait** clause can appear on a **single** construct.

C++

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

C++

Cross References

- **private** and **firstprivate** clauses, see Section 2.15.3 on page 215.
- **copyprivate** clause, see Section 2.15.5.2 on page 242.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.4.6.11 on page 356.
- **ompt_work_single_executor** and **ompt_work_single_other**, see Section 4.4.6.14 on page 357.
- **ompt_callback_work_t**, Section 4.6.2.18 on page 385.

Fortran

2.7.4 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**.

Fortran

Cross References

- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.4.6.11 on page 356.
- **ompt_work_workshare**, see Section 4.4.6.14 on page 357.
- **ompt_callback_work_t**, see Section 4.6.2.18 on page 385.

2.8 SIMD Constructs

2.8.1 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*]**)**

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 2.6 on page 58).

C / C++

Fortran

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

where *clause* is one of the following:

safelen(*length***)**

simdlen(*length***)**

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

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

private(*list***)**

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

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

collapse(*n***)**

If an **end simd** directive is not specified, an **end simd** 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 simd** 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

A **simd** region binds to the current task region. The binding thread set of the **simd** region is the 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.

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. If the **safelen** clause is used then no two iterations executed concurrently with SIMD instructions can have a greater distance in the logical iteration space than its value. The parameter of the **safelen** clause must be a constant positive integer expression. If used, 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. 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.

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.

Restrictions

- All loops associated with the construct must be perfectly nested; that is, there must be no intervening code nor any OpenMP directive 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 **`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

- The type of list items appearing in the **`aligned`** clause must be array or pointer.

C

C++

- The type of list items appearing in the **`aligned`** clause must be array, pointer, reference to array, or reference to pointer.
- No exception can be raised in the **`simd`** region.

C++

Fortran

- 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.
- If a list item on the **aligned** clause has the **ALLOCATABLE** attribute, the allocation status must be allocated.
- If a list item on the **aligned** clause has the **POINTER** attribute, the association status must be associated.
- If the type of a list item on the **aligned** clause is either **C_PTR** or Cray pointer, the list item must be defined.

Fortran

Cross References

- **private**, **lastprivate**, **linear** and **reduction** clauses, see Section 2.15.3 on page 215.

2.8.2 declare simd Construct

Summary

The **declare simd** construct can be applied to a function (C, C++ and Fortran) or a subroutine (Fortran) to enable the creation of one or more versions that can process multiple arguments using SIMD instructions from a single invocation in a SIMD loop. The **declare simd** directive is a declarative directive. There may be multiple **declare simd** directives for a function (C, C++, Fortran) or subroutine (Fortran).

Syntax

The syntax of the **declare simd** construct 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 a **`declare simd`** construct on a function enables the creation of 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 this directive are evaluated in the scope of the arguments of the function declaration or definition.

C / C++

Fortran

The use of a **`declare simd`** construct enables the creation of SIMD versions of the specified subroutine or function that can be used to process multiple arguments from a single invocation in a SIMD loop concurrently.

Fortran

If a **`declare simd`** directive contains multiple SIMD declarations, each declaration enables the creation of SIMD versions.

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

The **aligned** clause declares that the target 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 **inbranch** clause specifies that the SIMD version of the function will always be called from inside a conditional statement of a SIMD loop. The **notinbranch** clause specifies that the SIMD version of the function will never be called from inside a conditional statement of a SIMD loop. If neither clause is specified, then the SIMD version of the function may or may not be called from inside a conditional statement of a SIMD loop.

Restrictions

- Each argument can appear in at most one **uniform** or **linear** clause.
- At most one **simdlen** clause can appear in a **declare simd** directive.
- Either **inbranch** or **notinbranch** may be specified, but not both.
- When a *linear-step* expression is specified in a **linear** clause it must be either a constant integer expression or an integer-typed parameter that is specified in a **uniform** clause on the directive.
- The function or subroutine body must be a structured block.
- The execution of the function or subroutine, when called from a SIMD loop, cannot result in the execution of an OpenMP construct except for an **ordered** construct with the **simd** clause.
- The execution of the function or subroutine cannot have any side effects that would alter its execution for concurrent iterations of a SIMD chunk.
- A program that branches into or out of the function is non-conforming.

C / C++

- If the function has any declarations, then the **declare simd** construct for any declaration that has one must be equivalent to the one specified for the definition. Otherwise, the result is unspecified.
- The function cannot contain calls to the **longjmp** or **setjmp** functions.

C / C++

C

- The type of list items appearing in the **aligned** clause must be array or pointer.

C

C++

- The function cannot contain any calls to **throw**.
- The type of list items appearing in the **aligned** clause must be array, pointer, reference to array, or reference to pointer.

C++

Fortran

- *proc-name* must not be a generic name, procedure pointer or entry name.
- If *proc-name* is omitted, the **declare simd** directive must appear in the specification part of a subroutine subprogram or a function subprogram for which creation of the SIMD versions is enabled.
- Any **declare simd** directive must appear in the specification part of a subroutine subprogram, function subprogram or interface body to which it applies.
- If a **declare simd** directive is specified in an interface block for a procedure, it must match a **declare simd** directive in the definition of the procedure.
- If a procedure is declared via a procedure declaration statement, the procedure *proc-name* should appear in the same specification.
- If a **declare simd** directive is specified for a procedure name with explicit interface and a **declare simd** directive is also specified for the definition of the procedure then the two **declare simd** directives must match. Otherwise the result is unspecified.
- Procedure pointers may not be used to access versions created by the **declare simd** directive.
- The type of list items appearing in the **aligned** clause must be **C_PTR** or Cray pointer, or the list item must have the **POINTER** or **ALLOCATABLE** attribute.

Fortran

Cross References

- **`reduction`** clause, see Section 2.15.4.4 on page 236.
- **`linear`** clause, see Section 2.15.3.6 on page 228.

2.8.3 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 2.7.1 on page 62.
- **`simd`** construct, see Section 2.8.1 on page 80.
- Data attribute clauses, see Section 2.15.3 on page 215.
- Events and tool callbacks for the loop construct, see Section 2.7.1 on page 62.

2.9 Tasking Constructs

2.9.1 `task` Construct

Summary

The **`task`** construct defines an explicit task.

Syntax

C / C++

The syntax of the **`task`** construct is as follows:

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

where *clause* is one of the following:

`if(`[**`task :`**] *scalar-expression***`)`**

`final(`*scalar-expression***`)`**

`untied`

`default(shared | none)`

`mergeable`

`private(`*list***`)`**

`firstprivate(`*list***`)`**

`shared(`*list***`)`**

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

`depend(`*dependence-type* **`:`** *locator-list***`)`**

`priority(`*priority-value***`)`**

C / C++

Fortran

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

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

where *clause* is one of the following:

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

final(*scalar-logical-expression***)**

untied

default(private | firstprivate | shared | none)

mergeable

private(*list***)**

firstprivate(*list***)**

shared(*list***)**

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

depend(*dependence-type* **:** *locator-list***)**

priority(*priority-value***)**

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 explicit, non-merged 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`**.

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 2.9.6 on page 104.
- **depend** clause, see Section 2.13.10 on page 194.
- **if** Clause, see Section 2.12 on page 164.
- Data-sharing attribute clauses, Section 2.15.3 on page 215.
- **ompt_callback_task_create_t**, see Section 4.6.2.7 on page 373.

2.9.2 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.

Syntax

C / C++

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

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

where *clause* is one of the following:

if([**taskloop** **:**] *scalar-expr*)

shared(*list***)**

private(*list***)**

firstprivate(*list***)**

lastprivate(*list***)**

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

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

default(shared | none)

grainsize(*grain-size***)**

num_tasks(*num-tasks***)**

collapse(*n***)**

final(*scalar-expr***)**

priority(*priority-value***)**

untied

mergeable

nogroup

The **taskloop** directive places restrictions on the structure of all associated *for-loops*. Specifically, all associated *for-loops* must have canonical loop form (see Section 2.6 on page 58).

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

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, the only loop that is associated with the **taskloop** construct is the one that immediately follows the **taskloop** directive.

If 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.

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.
- All loops associated with the **taskloop** construct must be perfectly nested; that is, there must be no intervening code nor any OpenMP directive 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 2.9.1 on page 91.
- **taskgroup** construct, Section 2.13.6 on page 176.
- Data-sharing attribute clauses, Section 2.15.3 on page 215.
- **if** Clause, see Section 2.12 on page 164.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.4.6.11 on page 356.
- **ompt_work_taskloop**, see Section 4.4.6.14 on page 357.
- **ompt_callback_work_t**, see Section 4.6.2.18 on page 385.

2.9.3 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 2.9.2 on page 95.
- **simd** construct, see Section 2.8.1 on page 80.
- Data-sharing attribute clauses, see Section 2.15.3 on page 215.
- Events and tool callbacks for **taskloop** construct, see Section 2.9.2 on page 95.

2.9.4 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 2.9.6 on page 104.

2.9.5 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.

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**.

Cross References

- **ompt_task_initial**, see Section 4.4.6.17 on page 359.
- **ompt_callback_thread_begin_t**, see Section 4.6.2.1 on page 366.
- **ompt_callback_thread_end_t**, see Section 4.6.2.2 on page 367.
- **ompt_callback_task_create_t**, see Section 4.6.2.7 on page 373.

2.9.6 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. 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 4.6.2.10 on page 376.

2.10 Device Constructs

2.10.1 Device Initialization

Events

The *device-initialize* event occurs in a thread that encounters the first **target**, **target data**, or **target enter data** construct associated with a particular target device after the thread initiates initialization of OpenMP on the device and the device's OpenMP initialization, which may include device-side tool initialization, completes.

Tool Callbacks

A thread dispatches a registered **ompt_callback_device_initialize** callback for each occurrence of a *device-initialize* event in that thread. This callback has type signature **ompt_callback_device_initialize_t**.

Restrictions

No thread may offload execution of an OpenMP construct to a device until any callback associated with a *device-initialize* event completes.

Cross References

- **ompt_callback_device_initialize_t**, see Section 4.6.2.28 on page 396.

2.10.2 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 2.3 on page 39.
- **if** Clause, see Section 2.12 on page 164.
- **map** clause, see Section 2.15.6.1 on page 245.
- **ompt_callback_target_t**, see Section 4.6.2.20 on page 387.

2.10.3 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***)**

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***)**

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 2.15.6.1 on page 245) 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 2.9.1 on page 91.

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 2.9.1 on page 91.

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.

Restrictions

- A program must not depend on any ordering of the evaluations of the clauses of the **target enter 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.
- At most one **if** clause can appear on the directive.
- A *map-type* must be specified in all **map** clauses and must be either **to** or **alloc**.

Cross References

- *default-device-var*, see Section 2.3.1 on page 39.
- **task**, see Section 2.9.1 on page 91.
- **task scheduling constraints**, see Section 2.9.6 on page 104.
- **target data**, see Section 2.10.2 on page 107.
- **target exit data**, see Section 2.10.4 on page 112.
- **if** Clause, see Section 2.12 on page 164.
- **map** clause, see Section 2.15.6.1 on page 245.
- **ompt_callback_target_t**, see Section 4.6.2.20 on page 387.

2.10.4 target exit data Construct

Summary

The **target exit data** directive specifies that list items are unmapped from a device data 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***)**

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***)**

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 2.15.6.1 on page 245) 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 2.9.1 on page 91.

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 2.9.1 on page 91.

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.
- 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**.

Cross References

- *default-device-var*, see Section 2.3.1 on page 39.
- **task**, see Section 2.9.1 on page 91.
- **task scheduling constraints**, see Section 2.9.6 on page 104.
- **target data**, see Section 2.10.2 on page 107.
- **target enter data**, see Section 2.10.3 on page 109.
- **if** Clause, see Section 2.12 on page 164.
- **map** clause, see Section 2.15.6.1 on page 245.
- **ompt_callback_target_t**, see Section 4.6.2.20 on page 387.

2.10.5 `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***)**

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

map([[*map-type-modifier*[,]] *map-type***:**] *list***)**

is_device_ptr(*list***)**

defaultmap(tofrom:scalar)

nowait

depend(*dependence-type***:** *locator-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***)**

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

map([[*map-type-modifier*[,]] *map-type***:**] *list***)**

is_device_ptr(*list***)**

defaultmap(tofrom:scalar)

nowait

depend (*dependence-type* **:** *locator-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. A variable that appears as a list item in a **reduction** clause on the **target** construct has a default data-sharing attribute of shared in the data environment of the *target task*. Likewise, a variable that is mapped in the **target** 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 2.15.6.1 on page 245) 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 the array section is derived from a variable for which the type is pointer then the data-sharing attribute for that variable in the construct is firstprivate. Prior to the execution of the construct, the private variable is initialized with the address of the storage location of the corresponding array section in the device data environment.

If a zero-length array section is a list item in a **map** clause, and the array section is derived from a variable for the which the type is pointer then that variable is initialized with 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 then the 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.
- At most one **defaultmap** clause can appear on the directive.
- A *map-type* in a **map** clause must be **to**, **from**, **tofrom** or **alloc**.
- A list item that appears in an **is_device_ptr** clause must be a valid device pointer in the device data environment.

C

- A list item that appears in an **is_device_ptr** clause must have a type of pointer or array.

C

C++

- A list item that appears in an **is_device_ptr** clause must have a type of pointer, array, reference to pointer or reference to array.
- 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.
- A throw executed inside a **target** region must cause execution to resume within the same **target** region, and the same thread that threw the exception must catch it.

C++

Fortran

- A list item that appears in an **is_device_ptr** clause must be a dummy argument.
- If a list item in a **map** clause is an array section, and the array section is derived from a variable with a **POINTER** or **ALLOCATABLE** attribute then the behavior is unspecified if the corresponding list item's variable is modified in the region.

Fortran

Cross References

- *default-device-var*, see Section 2.3 on page 39.
- **task** construct, see Section 2.9.1 on page 91.
- **task** scheduling constraints, see Section 2.9.6 on page 104
- **target data** construct, see Section 2.10.2 on page 107.
- **if** Clause, see Section 2.12 on page 164.
- **private** and **firstprivate** clauses, see Section 2.15.3 on page 215.
- Data-mapping Attribute Rules and Clauses, see Section 2.15.6 on page 244.
- **ompt_callback_target_t**, see Section 4.6.2.20 on page 387.
- **ompt_callback_target_submit_t**, Section 4.6.2.23 on page 391.

2.10.6 `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***`)`**

and *motion-clause* is one of the following:

`to(`*list***`)`**

`from(`*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***)**

and *motion-clause* is one of the following:

to(*list***)**

from(*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.

For each list item in a **from** clause the value of the corresponding list item is assigned to the original list item.

For each list item in a **to** clause the value of the original list item is assigned to the corresponding list item.

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 2.15.6.1 on page 245) 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 2.9.1 on page 91.

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 2.9.1 on page 91.

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.

Cross References

- *default-device-var*, see Section 2.3 on page 39.
- Array sections, Section 2.4 on page 48
- **task** construct, see Section 2.9.1 on page 91.
- **task** scheduling constraints, see Section 2.9.6 on page 104
- **target data**, see Section 2.10.2 on page 107.
- **if** Clause, see Section 2.12 on page 164.
- **ompt_callback_task_create_t**, see Section 4.6.2.7 on page 373.
- **ompt_callback_target_t**, see Section 4.6.2.20 on page 387.

2.10.7 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
```

where *clause* is one of the following:

to (*extended-list*)

link (*list*)

C / C++

Fortran

The syntax of the **declare target** directive is as follows:

```
!$omp declare target (extended-list)
```

or

```
!$omp declare target [clause[ [ , ] clause] ... ]
```

where *clause* is one of the following:

to (*extended-list*)

link (*list*)

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.

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.

C / C++

Fortran

If a procedure that is host or use 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 or use 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.

Fortran

If a variable is treated as if it appeared as a list item in a **to** clause on a **declare target** directive then the corresponding list item in the device data environment of each device 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 list item. The list item is never removed from those device data environments as if its reference count is initialized to positive infinity.

The list items of a **link** clause are not mapped by the **declare target** directive. Instead, their mapping is deferred until they are mapped by **target data** or **target** constructs. They are mapped only for such regions.

C / C++

If a function is referenced in a function that is treated as if it appeared as a list item in a **to** clause on a **declare target** directive then the name of the referenced function is treated as if it had appeared in a **to** clause on a **declare target** directive.

If a variable with static storage duration or a function is referenced in the initializer expression list of a variable with static storage duration that is treated as if it appeared as a list item in a **to** clause on a **declare target** construct then the name of the referenced variable or function is treated as if it had appeared in a **to** clause on a **declare target** directive.

The form of the **declare target** directive that has no clauses and requires a matching **end declare target** directive defines an implicit *extended-list* to an implicit **to** clause. The implicit *extended-list* consists of the variable names of any variable declarations at file or namespace scope that appear between the two directives and of the function names of any function declarations at file, namespace or class scope that appear between the two directives.

C / C++

Fortran

If a procedure is referenced in a procedure that is treated as if it appeared as a list item in a **to** clause on a **declare target** directive then the name of the procedure is treated as if it had appeared in a **to** clause on a **declare target** directive.

If a **declare target** does not have any clauses then an implicit *extended-list* to an implicit **to** clause of one item is formed from the name of the enclosing subroutine subprogram, function subprogram or interface body to which it applies.

Fortran

Restrictions

- A threadprivate variable cannot appear in a **declare target** directive.
- A variable declared in a **declare target** directive must have a mappable type.
- The same list item must not appear multiple times in clauses on the same directive.
- The same list item must not appear in both a **to** clause on one **declare target** directive and a **link** clause on another **declare target** directive.

C / C++

- The *declaration-definition-seq* defined by a **declare target** directive and an **end declare target** directive must not contain any **declare target** directives.

C / C++

C++

- The function names of overloaded functions or template functions may only be specified within an implicit *extended-list*.

C++

Fortran

- If a list item is a procedure name, it must not be a generic name, procedure pointer or entry name.
- Any **declare target** directive with clauses must appear in a specification part of a subroutine subprogram, function subprogram, program or module.
- Any **declare target** directive without clauses must appear in a specification part of a 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 names cannot. This means that a common block name specified in a **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

2.10.8 teams Construct

Summary

The **teams** construct creates a league of thread teams and the master thread of each team executes 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***)**

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***)**

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 thread teams is created and the master thread of each thread 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 teams 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 teams remains constant for the duration of the **teams** region.

Within a **teams** region, team numbers uniquely identify each team. Team numbers are consecutive whole numbers ranging from zero to one less than the number of teams. A thread may obtain its own 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

- **default**, **shared**, **private**, **firstprivate**, and **reduction** clauses, see Section 2.15.3 on page 215.
- **omp_get_num_teams** routine, see Section 3.2.32 on page 295.
- **omp_get_team_num** routine, see Section 3.2.33 on page 297.

2.10.9 `distribute` Construct

Summary

The **`distribute`** construct specifies that the iterations of one or more loops will be executed by the thread teams in the context of their implicit tasks. The iterations are distributed across the master threads of all 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*]**`)`**

All associated *for-loops* must have the canonical form described in Section 2.6 on page 58.

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*]**)**

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 master threads executing an enclosing **teams** region. A **distribute** region binds to this **teams** region. Only the threads executing the binding **teams** region participate in the execution of the loop iterations.

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, the only loop that is associated with the **distribute** construct is the one that immediately follows the **distribute** construct.

If 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.

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 teams of the league in a round-robin fashion in the order of the 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 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 2.7.1 on page 62.
- **teams** construct, see Section 2.10.8 on page 129
- **ompt_work_distribute**, see Section 4.4.6.14 on page 357.
- **ompt_callback_work_t**, see Section 4.6.2.18 on page 385.

2.10.10 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.

Cross References

- **simd** construct, see Section 2.8.1 on page 80.
- **distribute** construct, see Section 2.10.9 on page 132.
- Data attribute clauses, see Section 2.15.3 on page 215.
- Events and tool callbacks for the **distribute** construct, see Section 2.8.1 on page 80.

2.10.11 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.

Restrictions

- The restrictions for the **distribute** and parallel loop constructs apply.
- No **ordered** clause can be specified.
- No **linear** clause can be specified.

Cross References

- **distribute** construct, see Section 2.10.9 on page 132.
- Parallel loop construct, see Section 2.11.1 on page 140.
- Data attribute clauses, see Section 2.15.3 on page 215.
- Events and tool callbacks for **distribute** construct, see Section 2.10.9 on page 132.
- Events and tool callbacks for parallel loop construct, see Section 2.11.1 on page 140.

2.10.12 Distribute Parallel Loop SIMD Construct

Summary

The distribute parallel loop SIMD construct specifies a loop that can be executed concurrently using SIMD instructions in parallel by multiple threads that are members of multiple teams. The distribute parallel loop SIMD construct is a composite construct.

Syntax

C / C++

The syntax of the distribute parallel loop SIMD construct is as follows:

```
#pragma omp distribute parallel for simd [clause[ [,] clause] ... ] newline
    for-loops
```

where *clause* can be any of the clauses accepted by the **distribute** or parallel loop SIMD directives with identical meanings and restrictions

C / C++

Fortran

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.

Cross References

- **distribute** construct, see Section 2.10.9 on page 132.
- Parallel loop SIMD construct, see Section 2.11.4 on page 145.
- Data attribute clauses, see Section 2.15.3 on page 215.
- Events and tool callbacks for **distribute** construct, see Section 2.10.9 on page 132.
- Events and tool callbacks for parallel loop construct, see Section 2.11.1 on page 140.

2.11 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.

2.11.1 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 2.5 on page 50.
- loop SIMD construct, see Section 2.8.3 on page 89.
- Data attribute clauses, see Section 2.15.3 on page 215.

2.11.2 `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

Description

C / C++

The semantics are identical to explicitly specifying a **`parallel`** directive immediately followed by a **`sections`** directive.

C / C++

Fortran

The semantics are identical to explicitly specifying a **`parallel`** directive immediately followed by a **`sections`** directive, and an **`end sections`** directive immediately followed by an **`end parallel`** directive.

Fortran

Restrictions

The restrictions for the **`parallel`** construct and the **`sections`** construct apply.

Cross References

- **`parallel`** construct, see Section 2.5 on page 50.
- **`sections`** construct, see Section 2.7.2 on page 71.
- Data attribute clauses, see Section 2.15.3 on page 215.

Fortran

2.11.3 `parallel workshare` Construct

Summary

The **`parallel workshare`** construct is a shortcut for specifying a **`parallel`** construct containing one **`workshare`** construct and no other statements.

Syntax

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 2.5 on page 50.
- **workshare** construct, see Section 2.7.4 on page 76.
- Data attribute clauses, see Section 2.15.3 on page 215.

Fortran

2.11.4 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 2.5 on page 50.
- loop SIMD construct, see Section 2.8.3 on page 89.
- Data attribute clauses, see Section 2.15.3 on page 215.

2.11.5 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.

Cross References

- **parallel** construct, see Section 2.5 on page 50.
- **target** construct, see Section 2.10.5 on page 116.
- **if** Clause, see Section 2.12 on page 164.
- Data attribute clauses, see Section 2.15.3 on page 215.

2.11.6 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 2.10.5 on page 116.
- Parallel loop construct, see Section 2.11.1 on page 140.
- **if** Clause, see Section 2.12 on page 164.
- Data attribute clauses, see Section 2.15.3 on page 215.

2.11.7 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 2.10.5 on page 116.
- Parallel loop SIMD construct, see Section 2.11.4 on page 145.
- **if** Clause, see Section 2.12 on page 164.
- Data attribute clauses, see Section 2.15.3 on page 215.

2.11.8 `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 2.8.1 on page 80.
- **target** construct, see Section 2.10.5 on page 116.
- Data attribute clauses, see Section 2.15.3 on page 215.

2.11.9 target teams Construct

Summary

The **target teams** construct is a shortcut for specifying a **target** construct containing a **teams** construct and no other statements.

Syntax

C / C++

The syntax of the **target teams** construct is as follows:

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

where *clause* can be any of the clauses accepted by the **target** or **teams** directives with identical meanings and restrictions.

C / C++

Fortran

The syntax of the **target teams** construct is as follows:

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

where *clause* can be any of the clauses accepted by the **target** or **teams** directives with identical meanings and restrictions.

Fortran

Description

The semantics are identical to explicitly specifying a **target** directive immediately followed by a **teams** directive.

Restrictions

The restrictions for the **target** and **teams** constructs apply.

Cross References

- **target** construct, see Section 2.10.5 on page 116.
- **teams** construct, see Section 2.10.8 on page 129.
- Data attribute clauses, see Section 2.15.3 on page 215.

2.11.10 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.

Cross References

- **teams** construct, see Section 2.10.8 on page 129.
- **distribute** construct, see Section 2.10.9 on page 132.
- Data attribute clauses, see Section 2.15.3 on page 215.

2.11.11 teams distribute simd Construct

Summary

The **teams distribute simd** construct is a shortcut for specifying a **teams** construct containing a **distribute simd** construct and no other statements.

Syntax

C / C++

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

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

where *clause* can be any of the clauses accepted by the **teams** or **distribute simd** directives with identical meanings and restrictions.

C / C++

Fortran

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

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

where *clause* can be any of the clauses accepted by the **teams** or **distribute simd** directives with identical meanings and restrictions.

If an **end teams distribute simd** directive is not specified, an **end teams distribute 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 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 simd** constructs apply.

Cross References

- **teams** construct, see Section 2.10.8 on page 129.
- **distribute simd** construct, see Section 2.10.10 on page 135.
- Data attribute clauses, see Section 2.15.3 on page 215.

2.11.12 `target teams distribute` Construct

Summary

The **`target teams distribute`** construct is a shortcut for specifying a **`target`** construct containing a **`teams distribute`** construct and no other statements.

Syntax

C / C++

The syntax of the **`target teams distribute`** construct is as follows:

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

where *clause* can be any of the clauses accepted by the **`target`** or **`teams distribute`** directives with identical meanings and restrictions.

C / C++

Fortran

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.

Cross References

- **`target`** construct, see Section 2.10.2 on page 107.
- **`teams distribute`** construct, see Section 2.11.10 on page 153.
- Data attribute clauses, see Section 2.15.3 on page 215.

2.11.13 `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.

Syntax

C / C++

The syntax of the **`target teams distribute simd`** construct is as follows:

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

where *clause* can be any of the clauses accepted by the **`target`** or **`teams distribute simd`** directives with identical meanings and restrictions.

C / C++

Fortran

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]
```

where *clause* can be any of the clauses accepted by the **`target`** or **`teams distribute simd`** directives with identical meanings and restrictions.

If an **`end target teams distribute simd`** directive is not specified, an **`end target teams distribute 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 simd** directive.

Restrictions

The restrictions for the **target** and **teams distribute simd** constructs apply.

Cross References

- **target** construct, see Section 2.10.2 on page 107.
- **teams distribute simd** construct, see Section 2.11.11 on page 154.
- Data attribute clauses, see Section 2.15.3 on page 215.

2.11.14 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 2.10.8 on page 129.
- Distribute parallel loop construct, see Section 2.10.11 on page 136.
- Data attribute clauses, see Section 2.15.3 on page 215.

2.11.15 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 2.10.5 on page 116.
- Teams distribute parallel loop construct, see Section 2.11.14 on page 158.
- **if** Clause, see Section 2.12 on page 164.
- Data attribute clauses, see Section 2.15.3 on page 215.

2.11.16 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 2.10.8 on page 129.
- Distribute parallel loop SIMD construct, see Section 2.10.12 on page 138.
- Data attribute clauses, see Section 2.15.3 on page 215.

2.11.17 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 2.10.5 on page 116.
- Teams distribute parallel loop SIMD construct, see Section 2.11.16 on page 161.
- **`if`** Clause, see Section 2.12 on page 164.
- Data attribute clauses, see Section 2.15.3 on page 215.

2.12 `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

The syntax of the **`if`** clause is as follows:

`if` **(**[*directive-name-modifier* **:**] *scalar-logical-expression*)

Fortran

Description

The effect of the **`if`** clause depends on the construct to which it is applied. For combined or composite constructs, the **`if`** clause only applies to the semantics of the construct named in the *directive-name-modifier* if one is specified. If no *directive-name-modifier* is specified for a combined or composite construct then the **`if`** clause applies to all constructs to which an **`if`** clause can apply.

2.13 Master and Synchronization Constructs and Clauses

OpenMP provides the following synchronization constructs:

- the **master** construct;
- the **critical** construct;
- the **barrier** construct;
- the **taskwait** construct;
- the **taskgroup** construct;
- the **atomic** construct;
- the **flush** construct;
- the **ordered** construct.

2.13.1 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 4.4.6.11 on page 356.
- **ompt_callback_master_t**, see Section 4.6.2.6 on page 371.

2.13.2 `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 lock hint (as described in Section 3.3.2 on page 304).

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_lock_hint_kind`** and a value that is a valid lock hint (as described in Section 3.3.2 on page 304).

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.

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_lock_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

- **omp_init_lock_with_hint** and **omp_init_nest_lock_with_hint**, see Section 3.3.2 on page 304.
- **ompt_mutex_critical**, see Section 4.4.6.15 on page 358.
- **ompt_callback_mutex_acquire_t**, see Section 4.6.2.15 on page 381.
- **ompt_callback_mutex_t**, see Section 4.6.2.16 on page 383.

2.13.3 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 4.4.6.11 on page 356.
- **ompt_sync_region_barrier**, see Section 4.4.6.12 on page 357.
- **ompt_callback_sync_region_t**, see Section 4.6.2.12 on page 378.
- **ompt_callback_cancel_t**, see Section 4.6.2.27 on page 395.

2.13.4 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 2.9.6 on page 104.

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 4.4.6.11 on page 356.
- **ompt_sync_region_barrier**, see Section 4.4.6.12 on page 357
- **ompt_cancel_detected**, see Section 4.4.6.23 on page 362.
- **ompt_callback_sync_region_t**, see Section 4.6.2.12 on page 378.
- **ompt_callback_cancel_t**, see Section 4.6.2.27 on page 395.

2.13.5 `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***`)`**

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***`)`**

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.

Cross References

- **task** construct, see Section 2.9.1 on page 91.
- Task scheduling, see Section 2.9.6 on page 104.

- **depend** clause, see Section 2.13.10 on page 194.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.4.6.11 on page 356.
- **ompt_sync_region_taskwait**, see Section 4.4.6.12 on page 357.
- **ompt_callback_sync_region_t**, see Section 4.6.2.12 on page 378.

2.13.6 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
```

C / C++

where *clause* is one of the following:

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

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***)**

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 2.9.6 on page 104.
- **task_reduction** Clause, see Section 2.15.4.5 on page 238.
- **ompt_scope_begin** and **ompt_scope_end**, see Section 4.4.6.11 on page 356.
- **ompt_sync_region_taskgroup**, see Section 4.4.6.12 on page 357.
- **ompt_callback_sync_region_t**, see Section 4.6.2.12 on page 378.

2.13.7 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 is one of the following:

read

write

update

capture

C / C++

The syntax of the **atomic** construct takes one of the following forms:

```
#pragma omp atomic [seq_cst[,]] atomic-clause [[,]seq_cst] new-line
    expression-stmt
```

or

```
#pragma omp atomic [seq_cst] new-line
    expression-stmt
```

or

```
#pragma omp atomic [seq_cst[,]] capture [[,]seq_cst] new-line
    structured-block
```

where *expression-stmt* is an expression statement with one of the following forms:

- If *atomic-clause* is **read**:

```
v = x;
```

- If *atomic-clause* is **write**:

```
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*.
- Neither of *x* and *expr* (as applicable) may access the storage location designated by *v*.
- *expr* is an expression with scalar type.
- *binop* is one of **+**, *****, **-**, **/**, **&**, **^**, **|**, **<<**, or **>>**.
- *binop*, *binop***=**, **++**, and **--** are not overloaded operators.
- The expression *x binop expr* must be numerically equivalent to *x binop (expr)*. This requirement is satisfied if the operators in *expr* have precedence greater than *binop*, or by using parentheses around *expr* or subexpressions of *expr*.
- The expression *expr binop x* must be numerically equivalent to *(expr) binop x*. This requirement is satisfied if the operators in *expr* have precedence equal to or greater than *binop*, or by using parentheses around *expr* or subexpressions of *expr*.
- For forms that allow multiple occurrences of *x*, the number of times that *x* is evaluated is unspecified.

C / C++

Fortran

The syntax of the **atomic** construct takes any of the following forms:

```
!$omp atomic [seq_cst[,]] read [[,]seq_cst]
      capture-statement
[!$omp end atomic]
```

or

```
!$omp atomic [seq_cst[,]] write [[,]seq_cst]
      write-statement
[!$omp end atomic]
```

or

Fortran (cont.)

```
!$omp atomic [seq_cst[,]] update [[,]seq_cst]
      update-statement
[!$omp end atomic]
```

or

```
!$omp atomic [seq_cst]
      update-statement
[!$omp end atomic]
```

or

```
!$omp atomic [seq_cst[,]] capture [[,]seq_cst]
      update-statement
      capture-statement
!$omp end atomic
```

or

```
!$omp atomic [seq_cst[,]] capture [[,]seq_cst]
      capture-statement
      update-statement
!$omp end atomic
```

or

```
!$omp atomic [seq_cst[,]] capture [[,]seq_cst]
      capture-statement
      write-statement
!$omp end atomic
```

where *write-statement* has the following form (if *atomic-clause* is **capture** or **write**):

x **=** *expr*

where *capture-statement* has the following form (if *atomic-clause* is **capture** or **read**):

v **=** *x*

and where *update-statement* has one of the following forms (if *atomic-clause* is **update**, **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

Binding

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 for the **atomic** region is all threads on the device. Otherwise, the binding thread set for the **atomic** region is all threads in the contention group. **atomic** regions enforce exclusive access with respect to other **atomic** regions that access the same storage location x among all threads in the binding thread set without regard to the teams to which the threads belong.

Description

The **atomic** construct with the **read** clause forces an atomic read of the location designated by x regardless of the native machine word size.

The **atomic** construct with the **write** clause forces an atomic write of the location designated by x regardless of the native machine word size.

The **atomic** construct with the **update** clause forces an atomic update of the location designated by x using the designated operator or intrinsic. Note that when no clause is present, the semantics are equivalent to atomic update. Only the read and write of the location designated by x are performed mutually atomically. The evaluation of *expr* or *expr_list* need not be atomic with respect to the read or write of the location designated by x. No task scheduling points are allowed between the read and the write of the location designated by x.

The **atomic** construct with the **capture** clause forces an atomic update of the location designated by x using the designated operator or intrinsic while also capturing the original or final value of the location designated by x with respect to the atomic update. The original or final value of the location designated by x is written in the location designated by v depending on the form of the **atomic** construct structured block or statements following the usual language semantics. Only the read and write of the location designated by x are performed mutually atomically. Neither the evaluation of *expr* or *expr_list*, nor the write to the location designated by v, need be atomic with respect to the read or write of the location designated by x. No task scheduling points are allowed between the read and the write of the location designated by x.

Any **atomic** construct with a **seq_cst** clause forces the atomically performed operation to include an implicit flush operation without a list.

Note – As with other implicit flush regions, Section 1.4.4 on page 21 reduces the ordering that must be enforced. The intent is that, when the analogous operation exists in C++11 or C11, a sequentially consistent **atomic** construct has the same semantics as a **memory_order_seq_cst** atomic operation in C++11/C11. Similarly, a non-sequentially consistent **atomic** construct has the same semantics as a **memory_order_relaxed** atomic operation in C++11/C11.

Unlike non-sequentially consistent **atomic** constructs, sequentially consistent **atomic** constructs preserve the interleaving (sequentially consistent) behavior of correct, data race free programs. However, they are not designed to replace the **flush** directive as a mechanism to enforce ordering for non-sequentially consistent **atomic** constructs, and attempts to do so require extreme caution. For example, a sequentially consistent **atomic write** construct may appear to be reordered with a subsequent non-sequentially consistent **atomic write** construct, since such reordering would not be observable by a correct program if the second write were outside an **atomic** directive.

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.

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 **`seq_cst`** clause may appear on the construct.

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 2.13.2 on page 167.
- **`barrier`** construct, see Section 2.13.3 on page 170.
- **`flush`** construct, see Section 2.13.8 on page 186.
- **`ordered`** construct, see Section 2.13.9 on page 190.
- **`reduction`** clause, see Section 2.15.4.4 on page 236.
- lock routines, see Section 3.3 on page 301.
- **`ompt_mutex_atomic`**, see Section 4.4.6.15 on page 358.
- **`ompt_callback_mutex_acquire_t`**, see Section 4.6.2.15 on page 381.
- **`ompt_callback_mutex_t`**, see Section 4.6.2.16 on page 383.

2.13.8 `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 18 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 [(list)] new-line
```

C / C++

Fortran

The syntax of the **`flush`** construct is as follows:

```
!$omp flush [(list)]
```

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.

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.

A **flush** region without a list 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.
- At entry to and exit from the **atomic** operation (read, write, update, or capture) performed in a sequentially consistent atomic region.
- 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.
- During a **cancel** or **cancellation point** region, if the *cancel-var* ICV is *true* and cancellation has been activated.

A **flush** region with a list is implied at the following locations:

- At entry to and exit from the **atomic** operation (read, write, update, or capture) performed in a non-sequentially consistent **atomic** region, where the list contains only the storage location designated as x according to the description of the syntax of the **atomic** construct in Section 2.13.7 on page 178.

Note – A **flush** region is not implied at the following locations:

- At entry to worksharing regions.
- At entry to or exit from a **master** region.

Events

The *flush* event occurs in a thread encountering the **flush** construct.

Tool Callbacks

A thread dispatches a registered **ompt_callback_flush** callback for each occurrence of a *flush* event in that thread. This callback has the type signature **ompt_callback_flush_t**.

Cross References

- **ompt_callback_flush_t**, see Section 4.6.2.19 on page 386.

2.13.9 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 2.7.1 on page 62.
- **simd** construct, see Section 2.8.1 on page 80.
- parallel loop construct, see Section 2.11.1 on page 140.
- **depend** Clause, see Section 2.13.10 on page 194
- **ompt_mutex_ordered**, see Section 4.4.6.15 on page 358.

- **ompt_callback_mutex_acquire_t**, see Section 4.6.2.15 on page 381.
- **ompt_callback_mutex_t**, see Section 4.6.2.16 on page 383.

2.13.10 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)
```

where *dependence-type* is one of the following:

in

out

inout

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**, or **inout**.

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** 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.

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**, 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.

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.

The list items that appear in the **depend** clause may include array sections.

Fortran

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 2.4 on page 48.
- **task** construct, see Section 2.9.1 on page 91.
- **target enter data** construct, see Section 2.10.3 on page 109.
- **target exit data** construct, see Section 2.10.4 on page 112.
- **target** construct, see Section 2.10.5 on page 116.
- **target update** construct, see Section 2.10.6 on page 121.
- Task scheduling constraints, see Section 2.9.6 on page 104.
- **ordered** construct, see Section 2.13.9 on page 190.
- **ompt_callback_task_dependences_t**, see Section 4.6.2.8 on page 374.
- **ompt_callback_task_dependence_t**, see Section 4.6.2.9 on page 375.

2.14 Cancellation Constructs

2.14.1 cancel Construct

Summary

The **cancel** construct activates cancellation of the innermost enclosing region of the type specified. The **cancel** construct is a stand-alone directive.

Syntax

C / C++

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

```
#pragma omp cancel construct-type-clause [ [ , ] if-clause] new-line
```

where *construct-type-clause* is one of the following:

parallel

sections

for

taskgroup

and *if-clause* is

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

The *cancel* event occurs after a task encounters a **cancel** construct if the *cancel-var* ICV is *true*.

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.

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 2.3.1 on page 39.
- **cancellation point** construct, see Section 2.14.2 on page 202.
- **if** Clause, see Section 2.12 on page 164.
- **omp_get_cancellation** routine, see Section 3.2.9 on page 271.
- **ompt_callback_cancel_t**, see Section 4.6.2.27 on page 395.

2.14.2 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 2.3.1 on page 39.
- **cancel** construct, see Section 2.14.1 on page 197.
- **omp_get_cancellation** routine, see Section 3.2.9 on page 271.
- **ompt_callback_cancel_t**, see Section 4.6.2.27 on page 395.

2.15 Data Environment

This section presents a directive and several clauses for controlling the data environment during the execution of **teams**, **parallel**, **simd**, task generating, and worksharing regions.

- Section 2.15.1 on page 205 describes how the data-sharing attributes of variables referenced in **teams**, **parallel**, **simd**, task generating, and worksharing regions are determined.
- The **threadprivate** directive, which is provided to create threadprivate memory, is described in Section 2.15.2 on page 210.
- Clauses that may be specified on directives to control the data-sharing attributes of variables referenced in **teams**, **parallel**, **simd**, task generating, or worksharing constructs are described in Section 2.15.3 on page 215

- Clauses that may be specified on directives to copy data values from private or threadprivate variables on one thread to the corresponding variables on other threads in the team are described in Section 2.15.5 on page 240.
- Clauses that may be specified on directives to control the data-mapping of variables to a device data environment are described in Section 2.15.6.1 on page 245.

2.15.1 Data-sharing Attribute Rules

This section describes how the data-sharing attributes of variables referenced in **target**, **parallel**, **task**, **taskloop**, **simd**, and worksharing regions are determined. The following two cases are described separately:

- Section 2.15.1.1 on page 205 describes the data-sharing attribute rules for variables referenced in a construct.
- Section 2.15.1.2 on page 209 describes the data-sharing attribute rules for variables referenced in a region, but outside any construct.

2.15.1.1 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** 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 is a list item in a **map** clause on the **target** construct and the array section is derived from a variable for which the type is pointer then that variable is firstprivate.

C / C++

Fortran

- Variables and common blocks appearing in **threadprivate** directives are threadprivate.
- The loop iteration variable(s) in the associated *do-loop(s)* of a **do**, **parallel do**, **taskloop**, or **distribute** construct is (are) private.
- The loop iteration variable in the associated *do-loop* of a **simd** construct with just one associated *do-loop* is linear with a *linear-step* that is the increment of the associated *do-loop*.
- The loop iteration variables in the associated *do-loops* of a **simd** construct with multiple associated *do-loops* are lastprivate.
- A loop iteration variable for a sequential loop in a **parallel** or task generating construct is private in the innermost such construct that encloses the loop.
- Implied-do indices and **forall** indices are private.
- Cray pointees have the same the data-sharing attribute as the storage with which their Cray pointers are associated.
- Assumed-size arrays are shared.
- An associate name preserves the association with the selector established at the **ASSOCIATE** statement.

Fortran

Variables with predetermined data-sharing attributes may not be listed in data-sharing attribute clauses, except for the cases listed below. For these exceptions only, listing a predetermined variable in a data-sharing attribute clause is allowed and overrides the variable's predetermined data-sharing attributes.

C / C++

- The loop iteration variable(s) in the associated *for-loop(s)* of a **for**, **parallel for**, **taskloop**, or **distribute** construct may be listed in a **private** or **lastprivate** clause.
- The loop iteration variable in the associated *for-loop* of a **simd** construct with just one associated *for-loop* may be listed in a **linear** clause 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* may be listed in a **lastprivate** clause.
- Variables with **const**-qualified type having no mutable member may be listed in a **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 2.15.3 on page 215.

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 2.15.3.1 on page 216).

- 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 2.15.6 on page 244) 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 2.15.3.4 on page 223.

2.15.1.2 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

2.15.2 `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 15 and Section 2.9 on page 91.

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 2.3 on page 39.
- Number of threads used to execute a **parallel** region, see Section 2.5.1 on page 55.
- **copyin** clause, see Section 2.15.5.1 on page 240.

2.15.3 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 28). 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 2.15.4.

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

2.15.3.1 `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 2.15.1.1 on page 205).

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.

Restrictions

The restrictions to the **`default`** clause are as follows:

- Only a single **`default`** clause may be specified on a **`parallel`**, **`task`**, **`taskloop`** or **`teams`** directive.

2.15.3.2 `shared` Clause

Summary

The **`shared`** clause declares one or more list items to be shared by tasks generated by a **`parallel`**, **`teams`**, or task generating construct.

Syntax

The syntax of the **`shared`** clause is as follows:

```
shared(list)
```

Description

All references to a list item within a task refer to the storage area of the original variable at the point the directive was encountered.

The programmer must ensure, by adding proper synchronization, that storage shared by an explicit task region does not reach the end of its lifetime before the explicit task region completes its execution.

Fortran

The association status of a shared pointer becomes undefined upon entry to and on exit from the **`parallel`**, **`teams`**, or task generating construct if it is associated with a target or a subobject of a target that is in a **`private`**, **`firstprivate`**, **`lastprivate`**, or **`reduction`** clause in the 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 2.15.1.2 on page 209. 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

2.15.3.3 `private` Clause

Summary

The **private** clause declares one or more list items to be private to a task or to a SIMD lane.

Syntax

The syntax of the private clause is as follows:

```
private(list)
```

Description

Each task that references a list item that appears in a **private** clause in any statement in the construct receives a new list item. Each SIMD lane used in a **simd** construct that references a list item that appears in a private clause in any statement in the construct receives a new list item. Language-specific attributes for new list items are derived from the corresponding original list item. Inside the construct, all references to the original list item are replaced by references to the new list item. In the rest of the region, it is unspecified whether references are to the new list item or the original list item.

C++

If the construct is contained in a member function, it is unspecified anywhere in the region if accesses through the implicit **this** pointer refer to the new list item or the original list item.

C++

Therefore, if an attempt is made to reference the original item, its value after the region is also unspecified. If a SIMD construct or a task does not reference a list item that appears in a **private** clause, it is unspecified whether SIMD lanes or the task receive a new list item.

The value and/or allocation status of the original list item will change only:

- if accessed and modified via pointer,
- if possibly accessed in the region but outside of the construct,
- as a side effect of directives or clauses, or

Fortran

- if accessed and modified via construct association.

Fortran

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.

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.

Fortran

2.15.3.4 `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)
```

Description

The **`firstprivate`** clause provides a superset of the functionality provided by the **`private`** clause.

A list item that appears in a **`firstprivate`** clause is subject to the **`private`** clause semantics described in Section 2.15.3.3 on page 218, except as noted. In addition, the new list item is initialized from the original list item existing before the construct. The initialization of the new list item is done once for each task that references the list item in any statement in the construct. The initialization is done prior to the execution of the construct.

For a **`firstprivate`** clause on a **`parallel`**, **`task`**, **`taskloop`**, **`target`**, or **`teams`** construct, the initial value of the new list item is the value of the original list item that exists immediately prior to the construct in the task region where the construct is encountered. For a **`firstprivate`** clause on a worksharing construct, the initial value of the new list item for each implicit task of the threads that execute the worksharing construct is the value of the original list item that exists in the implicit task immediately prior to the point in time that the worksharing construct is encountered.

To avoid race conditions, concurrent updates of the original list item must be synchronized with the read of the original list item that occurs as a result of the **`firstprivate`** clause.

If a list item appears in both **`firstprivate`** and **`lastprivate`** clauses, the update required for **`lastprivate`** occurs after all the initializations for **`firstprivate`**.

C / C++

For variables of non-array type, the initialization occurs by copy assignment. For an array of elements of non-array type, each element is initialized as if by assignment from an element of the original array to the corresponding element of the new array.

C / C++

C++

For variables of class type, a copy constructor is invoked to perform the initialization. The order in which copy constructors for different variables of class 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.

Fortran

2.15.3.5 `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 2.15.3.3 on page 218. In addition, when a **lastprivate** clause without the **conditional** modifier appears on the directive that identifies a worksharing construct or a SIMD construct, 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.

The original list item becomes defined at the end of the construct if there is an implicit barrier at that point. 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.

If the **lastprivate** clause is used on a construct that does not end with an implicit barrier, 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.

Fortran

2.15.3.6 `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 2.15.3.3 on page 218 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

2.15.4 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.

2.15.4.1 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 2.7 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 2.7: Implicitly Declared C/C++ *reduction-identifiers*

<table>
<tr><th>Identifier</th><th>Initializer</th><th>Combiner</th></tr>
<tr><td>+</td><td>omp_priv = 0</td><td>omp_out += omp_in</td></tr>
<tr><td>*</td><td>omp_priv = 1</td><td>omp_out *= omp_in</td></tr>
<tr><td>-</td><td>omp_priv = 0</td><td>omp_out += omp_in</td></tr>
<tr><td>&</td><td>omp_priv = ~0</td><td>omp_out &= omp_in</td></tr>
<tr><td>|</td><td>omp_priv = 0</td><td>omp_out |= omp_in</td></tr>
<tr><td>^</td><td>omp_priv = 0</td><td>omp_out ^= omp_in</td></tr>
<tr><td>&&</td><td>omp_priv = 1</td><td>omp_out = omp_in && omp_out</td></tr>
<tr><td>||</td><td>omp_priv = 0</td><td>omp_out = omp_in || omp_out</td></tr>
<tr><td>max</td><td>omp_priv = Least representable number in the reduction list item type</td><td>omp_out = omp_in > omp_out ? omp_in : omp_out</td></tr>
<tr><td>min</td><td>omp_priv = Largest representable number in the reduction list item type</td><td>omp_out = omp_in < omp_out ? omp_in : omp_out</td></tr>
</table>

C / C++

Fortran

Table 2.8 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 2.8: 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.

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, 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

2.15.4.2 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 2.15.4.1, 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. 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*. 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.

2.15.4.3 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 2.15.4.1, 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.

2.15.4.4 `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 2.15.4.1.

Description

The **reduction** clause is a reduction scoping clause and a reduction participating clause, as described in Sections 2.15.4.2 and 2.15.4.3.

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 as if the **private** clause had been used. 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*.

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 2.15.4.1, 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++

2.15.4.5 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 2.15.4.1.

Description

The **task_reduction** clause is a reduction scoping clause, as described in 2.15.4.2.

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 2.15.4.1, apply to this clause.

2.15.4.6 `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 2.15.4.1

Description

The **`in_reduction`** clause is a reduction participating clause, as described in Section 2.15.4.3.

Restrictions

The restrictions to the **`in_reduction`** clause are as follows:

- All the common restrictions to all reduction clauses, which are listed in Section 2.15.4.1, apply to this clause.
- A list item that appears in an **`in_reduction`** clause of a **`task`** 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.

2.15.5 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 28). 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

2.15.5.1 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.

Fortran

2.15.5.2 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)
```

Description

The effect of the **copyprivate** clause on the specified list items occurs after the execution of the structured block associated with the **single** construct (see Section 2.7.3 on page 74), and before any of the threads in the team have left the barrier at the end of the construct.

C / C++

In all other implicit tasks belonging to the **parallel** region, each specified list item becomes defined with the value of the corresponding list item in the implicit task associated with the thread that executed the structured block. For variables of non-array type, the definition occurs by copy assignment. For an array of elements of non-array type, each element is copied by copy assignment from an element of the array in the data environment of the implicit task associated with the thread that executed the structured block to the corresponding element of the array in the data environment of the other implicit tasks

C / C++

C++

For class types, a copy assignment operator is invoked. The order in which copy assignment operators for different variables of class type are called is unspecified.

C++

Fortran

If a list item does not have the **POINTER** attribute, then in all other implicit tasks belonging to the **parallel** region, the list item becomes defined as if by intrinsic assignment with the value of the corresponding list item in the implicit task associated with the thread that executed the structured block.

If the list item has the **POINTER** attribute, then, in all other implicit tasks belonging to the **parallel** region, the list item receives, as if by pointer assignment, the same association status of the corresponding list item in the implicit task associated with the thread that executed the structured block.

The order in which any final subroutines for different variables of a finalizable type are called is unspecified.

Fortran

Note – The **copyprivate** clause is an alternative to using a shared variable for the value when providing such a shared variable would be difficult (for example, in a recursion requiring a different variable at each level).

Restrictions

The restrictions to the **copyprivate** clause are as follows:

- All list items that appear in the **copyprivate** clause must be either threadprivate or private in the enclosing context.
- A list item that appears in a **copyprivate** clause may not appear in a **private** or **firstprivate** clause on the **single** construct.

C++

- A variable of class type (or array thereof) that appears in a **copyprivate** clause requires an accessible unambiguous copy assignment operator for the class type.

C++

Fortran

- A common block that appears in a **copyprivate** clause must be threadprivate.
- Pointers with the **INTENT(IN)** attribute may not appear in the **copyprivate** clause.
- The list item with the **ALLOCATABLE** attribute must have the allocation status of allocated when the intrinsic assignment is performed.

Fortran

2.15.6 Data-mapping Attribute Rules and Clauses

This section describes how the data-mapping attributes of any variable referenced in a **target** region are determined. When specified, explicit **map** clauses on **target data** and **target** directives determine these attributes. Otherwise, the following data-mapping rules apply for variables referenced in a **target** construct that are not declared in the construct and do not appear in data-sharing attribute or **map** clauses:

Certain variables and objects have predetermined data-mapping attributes as follows:

- If a variable appears in a **to** or **link** clause on a **declare target** directive then it is treated as if it had appeared in a **map** clause with a *map-type* of **tofrom**.

C / C++

- A variable that is of type pointer is treated as if it had appeared in a **map** clause as a zero-length 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.

C++

Otherwise, the following implicit data-mapping attribute rules apply:

- If a **defaultmap(tofrom:scalar)** clause is not present then a scalar variable is not mapped, but instead has an implicit data-sharing attribute of firstprivate (see Section 2.15.1.1 on page 205).
- 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**.

2.15.6.1 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 **always**.

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.

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.
2. 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 the corresponding list item is assigned the value of the original list item.
4. If the corresponding list item's reference count is one, then:
   a) If the *map-type* is **from** or **alloc**, the value of the corresponding list item is undefined.

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 *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 the original list item is assigned the value of the corresponding list item.

c) If the corresponding list item's reference count is zero, then the corresponding list item is removed from the device data environment

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-transfer* event occurs when a thread initiates a data transfer to or from 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_transfer** callback for each occurrence of a *target-transfer* event in that thread. The callback occurs in the context of the target task. The callback has type signature **ompt_callback_target_transfer_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.
- At most one list item can be an array item derived from a given variable in **`map`** clauses of the same construct.
- List items of **`map`** clauses in the same construct must not share original storage.
- 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 correspnding 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.
- If variables that share storage are mapped, the behavior is unspecified.
- A list item must have a mappable type.
- **`threadprivate`** variables cannot appear in a **`map`** clause.

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.

C++

C / C++

- Initialization and assignment are through bitwise copy.
- A variable for which the type is pointer and an array section derived from that variable must not appear as list items of **`map`** clauses of the same construct.
- A list item cannot be a variable that is a member of a structure with a union type.
- A bit-field cannot appear in a **`map`** clause.

C / C++

Fortran

- The value of the new list item becomes that of the original list item in the map initialization and assignment.
- A list item must not contain any components that have the **ALLOCATABLE** attribute.
- If the allocation status of a list item with the **ALLOCATABLE** attribute is unallocated upon entry to a **target** region, the list item must be unallocated upon exit from the region.
- If the allocation status of a list item with the **ALLOCATABLE** attribute is allocated upon entry to a **target** region, the allocation status of the corresponding list item must not be changed and must not be reshaped in the region.
- If an array section is mapped and the size of the section is smaller than that of the whole array, the behavior of referencing the whole array in the **target** region is unspecified.

Fortran

2.15.6.2 `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

C / C++

The syntax of the **defaultmap** clause is as follows:

```
defaultmap(tofrom:scalar)
```

C / C++

Fortran

The syntax of the **defaultmap** clause is as follows:

```
defaultmap(tofrom:scalar)
```

Fortran

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*.

2.16 `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
- *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

- Only the variables **`omp_in`** and **`omp_out`** are allowed in the *combiner*.
- Only the variables **`omp_priv`** and **`omp_orig`** are allowed in the *initializer-clause*.
- 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

- If the *initializer-expr* is a subroutine name with an argument list, then one of the arguments must be **`omp_priv`**.
- If the **`declare reduction`** directive appears in the specification part of a module and the corresponding reduction clause does not appear in the same module, the *reduction-identifier* must be the same as the name of a user-defined operator, one of the allowed operators that is extended or a generic name that is the same as the name of one of the allowed intrinsic procedures.

- If the **declare reduction** directive appears in the specification of a module, if the corresponding **reduction** clause does not appear in the same module, and 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 the same as one of the allowed intrinsic procedures then the interface for that operator or the generic name must be defined in the specification of the same module, or must be accessible by use association.
- Any subroutine or function used in the **initializer** clause or *combiner* expression must be an intrinsic function, or must have an accessible interface.
- Any user-defined operator or extended operator used in the **initializer** clause or *combiner* expression must have an accessible interface.
- If any subroutine, function, user-defined operator, or extended operator is used in the **initializer** clause or *combiner* expression, it must be accessible to the subprogram in which the corresponding **reduction** clause is specified.
- If the length type parameter is specified for a character type, it must be a constant, a colon or an *****.
- If a character type with deferred or assumed length parameter is specified in a **declare reduction** directive, no other **declare reduction** directive with Fortran character type of the same kind parameter and the same *reduction-identifier* is allowed in the same scope.
- Any subroutine used in the **initializer** clause or *combiner* expression must not have any alternate returns appear in the argument list.

Fortran

Cross References

- **reduction** clause, Section 2.15.4.4 on page 236.

2.17 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.
- An **ordered** construct with the **simd** clause is the only OpenMP construct that can be encountered during execution of a **simd** region.
- 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, 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.
- The region associated with the **distribute** construct must be strictly nested inside a **teams** region.
- 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 **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*.

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 260).
- Execution environment routines that can be used to control and to query the parallel execution environment (Section 3.2 on page 261).
- Lock routines that can be used to synchronize access to data (Section 3.3 on page 301).
- Portable timer routines (Section 3.4 on page 314).
- Device memory routines that can be used to allocate memory and to manage pointers on target devices (Section 3.5 on page 317).
- Execution routines to control the application monitoring (Section 3.6 on page 327)

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**.

See Section Section B.1 on page 470 for an example of this file.

C / C++

Fortran

The OpenMP Fortran API runtime library routines are external procedures. The return values of these routines are of default kind, unless otherwise specified.

Interface declarations for the OpenMP Fortran runtime library routines described in this chapter shall be provided in the form of a Fortran **include** file named **omp_lib.h** or a Fortran 90 **module** named **omp_lib**. It is implementation defined whether the **include** file or the **module** file (or both) is provided.

These files define the following:

- The interfaces of all of the routines in this chapter.
- The **integer parameter omp_lock_kind**.
- The **integer parameter omp_nest_lock_kind**.
- The **integer parameter omp_lock_hint_kind**.
- The **integer parameter omp_sched_kind**.
- The **integer parameter omp_proc_bind_kind**.
- The **integer parameter openmp_version** with a value *yyyymm* where *yyyy* and *mm* are the year and month designations of the version of the OpenMP Fortran API that the implementation supports. This value matches that of the C preprocessor macro **_OPENMP**, when a macro preprocessor is supported (see Section 2.2 on page 36).

See Section B.1 on page 474 and Section B.3 on page 478 for examples of these files.

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 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

C / C++

```
void omp_set_num_threads(int num_threads);
```

C / C++

Fortran

```
subroutine omp_set_num_threads(num_threads)
integer num_threads
```

Fortran

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 2.3 on page 39.
- **parallel** construct and **num_threads** clause, see Section 2.5 on page 50.
- Determining the number of threads for a **parallel** region, see Section 2.5.1 on page 55.
- **omp_get_max_threads** routine, see Section 3.2.3 on page 264.
- **OMP_NUM_THREADS** environment variable, see Section 5.2 on page 435.

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 2.5 on page 50.
- Determining the number of threads for a **parallel** region, see Section 2.5.1 on page 55.
- **omp_set_num_threads** routine, see Section 3.2.1 on page 262.
- **OMP_NUM_THREADS** environment variable, see Section 5.2 on page 435.

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

C / C++

```
int omp_get_max_threads(void);
```

C / C++

Fortran

```
integer function omp_get_max_threads()
```

Fortran

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 2.3 on page 39.
- **parallel** construct, see Section 2.5 on page 50.
- **num_threads** clause, see Section 2.5 on page 50.
- Determining the number of threads for a **parallel** region, see Section 2.5.1 on page 55.
- **omp_set_num_threads** routine, see Section 3.2.1 on page 262.
- **OMP_NUM_THREADS** environment variable, see Section 5.2 on page 435.

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 263.

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 2.3 on page 39.
- **parallel** construct, see Section 2.5 on page 50.
- **omp_get_active_level** routine, see Section 3.2.20 on page 283.

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 2.3 on page 39.
- Determining the number of threads for a **parallel** region, see Section 2.5.1 on page 55.
- **omp_get_num_threads** routine, see Section 3.2.2 on page 263.
- **omp_get_dynamic** routine, see Section 3.2.8 on page 270.
- **OMP_DYNAMIC** environment variable, see Section 5.3 on page 436.

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 / C++

```
int omp_get_dynamic(void);
```

C / C++

Fortran

```
logical function omp_get_dynamic()
```

Fortran

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 2.3 on page 39.
- Determining the number of threads for a **parallel** region, see Section 2.5.1 on page 55.
- **omp_set_dynamic** routine, see Section 3.2.7 on page 268.
- **OMP_DYNAMIC** environment variable, see Section 5.3 on page 436.

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 2.3.1 on page 39.
- **cancel** construct, see Section 2.14.1 on page 197
- **OMP_CANCELLATION** environment variable, see Section 5.11 on page 442

3.2.10 omp_set_nested

Summary

The **omp_set_nested** routine enables or disables nested parallelism, by setting the *nest-var* ICV.

Format

C / C++

```
void omp_set_nested(int nested);
```

C / C++

Fortran

```
subroutine omp_set_nested(nested)
logical nested
```

Fortran

Binding

The binding task set for an **omp_set_nested** region is the generating task.

Effect

For implementations that support nested parallelism, if the argument to **omp_set_nested** evaluates to *true*, nested parallelism is enabled for the current task; otherwise, nested parallelism is disabled for the current task. For implementations that do not support nested parallelism, this routine has no effect: the value of *nest-var* remains *false*.

Cross References

- *nest-var* ICV, see Section 2.3 on page 39.
- Determining the number of threads for a **parallel** region, see Section 2.5.1 on page 55.
- **omp_set_max_active_levels** routine, see Section 3.2.15 on page 277.
- **omp_get_max_active_levels** routine, see Section 3.2.16 on page 279.
- **omp_get_nested** routine, see Section 3.2.11 on page 273.
- **OMP_NESTED** environment variable, see Section 5.6 on page 439.

3.2.11 omp_get_nested

Summary

The **omp_get_nested** routine returns the value of the *nest-var* ICV, which determines if nested parallelism is enabled or disabled.

Format

C / C++

```
int omp_get_nested(void);
```

C / C++

Fortran

```
logical function omp_get_nested()
```

Fortran

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*.

Cross References

- *nest-var* ICV, see Section 2.3 on page 39.
- Determining the number of threads for a **parallel** region, see Section 2.5.1 on page 55.
- **omp_set_nested** routine, see Section 3.2.10 on page 271.
- **OMP_NESTED** environment variable, see Section 5.6 on page 439.

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.

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

Constraints on Arguments

The first argument passed to this routine can be one of the valid OpenMP schedule kinds (except for **runtime**) or any implementation specific schedule. 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, which can be extended with 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 2.3 on page 39.
- Determining the schedule of a worksharing loop, see Section 2.7.1.1 on page 70.
- **omp_get_schedule** routine, see Section 3.2.13 on page 276.
- **OMP_SCHEDULE** environment variable, see Section 5.1 on page 434.

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 274, 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 274.

Cross References

- *run-sched-var* ICV, see Section 2.3 on page 39.
- Determining the schedule of a worksharing loop, see Section 2.7.1.1 on page 70.
- **omp_set_schedule** routine, see Section 3.2.12 on page 274.
- **OMP_SCHEDULE** environment variable, see Section 5.1 on page 434.

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 2.3 on page 39.
- **OMP_THREAD_LIMIT** environment variable, see Section 5.10 on page 442.

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 2.3 on page 39.
- **omp_get_max_active_levels** routine, see Section 3.2.16 on page 279.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 5.9 on page 442.

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

C / C++

```
int omp_get_max_active_levels(void);
```

C / C++

Fortran

```
integer function omp_get_max_active_levels()
```

Fortran

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 2.3 on page 39.
- **`omp_set_max_active_levels`** routine, see Section 3.2.15 on page 277.
- **`OMP_MAX_ACTIVE_LEVELS`** environment variable, see Section 5.9 on page 442.

3.2.17 omp_get_level

Summary

The **omp_get_level** routine returns the value of the *levels-var* ICV.

Format

C / C++

```
int omp_get_level(void);
```

C / C++

Fortran

```
integer function omp_get_level()
```

Fortran

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 2.3 on page 39.
- **omp_get_active_level** routine, see Section 3.2.20 on page 283.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 5.9 on page 442.

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 266.
- **omp_get_level** routine, see Section 3.2.17 on page 280.
- **omp_get_team_size** routine, see Section 3.2.19 on page 282.

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

C / C++

```
int omp_get_team_size(int level);
```

C / C++

Fortran

```
integer function omp_get_team_size(level)
integer level
```

Fortran

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 263.
- **omp_get_level** routine, see Section 3.2.17 on page 280.
- **omp_get_ancestor_thread_num** routine, see Section 3.2.18 on page 281.

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 2.3 on page 39.
- **omp_get_level** routine, see Section 3.2.17 on page 280.

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 2.9.1 on page 91.

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 2.5.2 on page 57 for the rules governing the thread affinity policy.

Cross References

- *bind-var* ICV, see Section 2.3 on page 39.
- Controlling OpenMP thread affinity, see Section 2.5.2 on page 57.
- **OMP_PROC_BIND** environment variable, see Section 5.4 on page 436.

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 2.3 on page 39.
- **OMP_PLACES** environment variable, see Section 5.5 on page 437.

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 / C++

```
int omp_get_place_num_procs(int place_num);
```

C / C++

Fortran

```
integer function omp_get_place_num_procs(place_num)
integer place_num
```

Fortran

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 437.

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 288.
- **omp_get_num_places** routine, see Section 3.2.23 on page 287.
- **OMP_PLACES** environment variable, see Section 5.5 on page 437.

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++

```c
int omp_get_place_num(void);
```

C / C++

Fortran

```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 2.5.2 on page 57.
- **omp_get_num_places** routine, see Section 3.2.23 on page 287.
- **OMP_PLACES** environment variable, see Section 5.5 on page 437.

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++

```
int omp_get_partition_num_places(void);
```

C / C++

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 2.3 on page 39.
- Controlling OpenMP thread affinity, see Section 2.5.2 on page 57.
- **`OMP_PLACES`** environment variable, see Section 5.5 on page 437.

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 2.3 on page 39.
- Controlling OpenMP thread affinity, see Section 2.5.2 on page 57.
- **omp_get_partition_num_places** routine, see Section 3.2.27 on page 291.
- **OMP_PLACES** environment variable, see Section 5.5 on page 437.

3.2.29 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 2.3 on page 39.
- **omp_get_default_device**, see Section 3.2.30 on page 294.
- **OMP_DEFAULT_DEVICE** environment variable, see Section 5.13 on page 444

3.2.30 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 2.3 on page 39.
- **omp_set_default_device**, see Section 3.2.29 on page 293.
- **OMP_DEFAULT_DEVICE** environment variable, see Section 5.13 on page 444.

3.2.31 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.

3.2.32 omp_get_num_teams

Summary

The **omp_get_num_teams** routine returns the number of teams in the current **teams** region.

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 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 2.10.8 on page 129.

3.2.33 omp_get_team_num

Summary

The **omp_get_team_num** routine returns the 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 team number of the calling thread. The 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 2.10.8 on page 129.
- **omp_get_num_teams** routine, see Section 3.2.32 on page 295.

3.2.34 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 2.10.5 on page 116

3.2.35 omp_get_initial_device

Summary

The **omp_get_initial_device** routine returns a device number representing the host device.

Format

C / C++

```
int omp_get_initial_device(void);
```

C / C++

Fortran

```
integer function omp_get_initial_device()
```

Fortran

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 2.10.5 on page 116
- Device memory routines, see Section 3.5 on page 317.

3.2.36 omp_get_max_task_priority

Summary

The **omp_get_max_task_priority** routine returns the maximum value that can be specified in the **priority** clause.

Format

C / C++

```
int omp_get_max_task_priority(void);
```

C / C++

Fortran

```
integer function omp_get_max_task_priority()
```

Fortran

Binding

The binding thread set for an **omp_get_max_task_priority** 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_max_task_priority** routine returns the value of the *max-task-priority-var* ICV, which determines the maximum value that can be specified in the **priority** clause.

Cross References

- *max-task-priority-var*, see Section 2.3 on page 39.
- **task** construct, see Section 2.9.1 on page 91.

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 *lock hint* that contains information about the intended usage of the lock by the application code. The effect of the lock hint is implementation defined. An OpenMP implementation can use this hint to select a usage-specific lock, but lock hints do not change the mutual exclusion semantics of locks. A conforming implementation can safely ignore the lock 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_lock_init_t**. The callbacks occur in the task encountering the routine. The callback receives **omp_lock_hint_none** as *hint* argument and **ompt_mutex_lock** or **ompt_mutex_nest_lock** as *kind* argument as appropriate.

Cross References

- **ompt_callback_lock_init_t**, see Section 4.6.2.13 on page 379.

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_lock_hint_t hint);
void omp_init_nest_lock_with_hint(omp_nest_lock_t *lock,
                                  omp_lock_hint_t hint);
```

C / C++

Fortran

```
subroutine omp_init_lock_with_hint(svar, hint)
integer (kind=omp_lock_kind) svar
integer (kind=omp_lock_hint_kind) hint

subroutine omp_init_nest_lock_with_hint(nvar, hint)
integer (kind=omp_nest_lock_kind) nvar
integer (kind=omp_lock_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 this routine (*hint*) can be one of the valid OpenMP lock hints below or any implementation-defined hint. 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 lock hint constants. The valid constants must include the following, which can be extended with implementation-defined values:

C / C++

```
typedef enum omp_lock_hint_t {
  omp_lock_hint_none = 0,
  omp_lock_hint_uncontended = 1,
  omp_lock_hint_contended = 2,
  omp_lock_hint_nonspeculative = 4,
  omp_lock_hint_speculative = 8
} omp_lock_hint_t;
```

C / C++

Fortran

```fortran
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
```

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_lock_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_lock_hint_uncontended** and **omp_lock_hint_contended** cannot be combined,
- the hints **omp_lock_hint_nonspeculative** and **omp_lock_hint_speculative** cannot be combined.

Note – Future OpenMP specifications may add additional hints to the **omp_lock_hint_t** type and the **omp_lock_hint_kind** kind. Implementers are advised to add implementation-defined hints starting from the most significant bit of the **omp_lock_hint_t** type and **omp_lock_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.

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_lock_init_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_lock_init_t**, see Section 4.6.2.13 on page 379.

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_init_destroy** or **omp_init_nest_destroy** 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_lock_destroy_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_lock_destroy_t**, see Section 4.6.2.14 on page 380.

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 4.6.2.15 on page 381.
- **`ompt_callback_mutex_t`**, see Section 4.6.2.16 on page 383.
- **`ompt_callback_nest_lock_t`**, see Section 4.6.2.17 on page 384.

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 4.6.2.16 on page 383.
- **ompt_callback_nest_lock_t**, see Section 4.6.2.17 on page 384.

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 4.6.2.15 on page 381.
- **ompt_callback_mutex_t**, see Section 4.6.2.16 on page 383.
- **ompt_callback_nest_lock_t**, see Section 4.6.2.17 on page 384.

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

C / C++

```
double omp_get_wtime(void);
```

C / C++

Fortran

```
double precision function omp_get_wtime()
```

Fortran

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.

Effect

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.

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

3.4.2 `omp_get_wtick`

Summary

The **`omp_get_wtick`** routine returns the precision of the timer used by **`omp_get_wtime`**.

Format

C / C++

```
double omp_get_wtick(void);
```

C / C++

Fortran

```
double precision function omp_get_wtick()
```

Fortran

Binding

The binding thread set for an **`omp_get_wtick`** region is the encountering thread. The routine's return value is not guaranteed to be consistent across any set of threads.

Effect

The **`omp_get_wtick`** routine returns a value equal to the number of seconds between successive clock ticks of the timer used by **`omp_get_wtime`**.

C / C++

3.5 Device Memory Routines

This section describes routines that support allocation of memory and management of pointers in the data environments of target devices.

3.5.1 omp_target_alloc

Summary

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 2.10.5 on page 116.

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 2.10.5 on page 116
- **omp_get_num_devices** routine, see Section 3.2.31 on page 295
- **omp_get_initial_device** routine, see Section 3.2.35 on page 298
- **omp_target_free** routine, see Section 3.5.2 on page 318
- **ompt_callback_target_data_op_t**, see Section 4.6.2.21 on page 388.

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 2.10.5 on page 116
- **omp_get_num_devices** routine, see Section 3.2.31 on page 295
- **omp_get_initial_device** routine, see Section 3.2.35 on page 298
- **omp_target_alloc** routine, see Section 3.5.1 on page 317
- **ompt_callback_target_data_op_t**, see Section 4.6.2.21 on page 388.

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(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 *true* if the specified pointer would be found present on device *device_num* by a **map** clause; otherwise, it returns *false*.

When called from within a **target** region the effect of this routine is unspecified.

Cross References

- **target** construct, see Section 2.10.5 on page 116
- **map** clause, see Section 2.15.6.1 on page 245.
- **omp_get_num_devices** routine, see Section 3.2.31 on page 295
- **omp_get_initial_device** routine, see Section 3.2.35 on page 298

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, 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-transfer* 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-transfer* 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 2.10.5 on page 116
- **omp_get_initial_device** routine, see Section 3.2.35 on page 298
- **omp_target_alloc** routine, see Section 3.5.1 on page 317
- **ompt_callback_target_data_op_t**, see Section 4.6.2.21 on page 388.

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, 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-transfer* 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-transfer* 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 2.10.5 on page 116
- **omp_get_initial_device** routine, see Section 3.2.35 on page 298
- **omp_target_alloc** routine, see Section 3.5.1 on page 317
- **ompt_callback_target_data_op_t**, see Section 4.6.2.21 on page 388.

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(void * host_ptr, 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 2.10.5 on page 116
- **map** clause, see Section 2.15.6.1 on page 245.
- **omp_target_alloc** routine, see Section 3.5.1 on page 317
- **omp_target_disassociate_ptr** routine, see Section 3.5.6 on page 324
- **ompt_callback_target_data_op_t**, see Section 4.6.2.21 on page 388.

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(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**.

C/C++ (cont.)

Cross References

- **target** construct, see Section 2.10.5 on page 116
- **omp_target_associate_ptr** routine, see Section 3.5.6 on page 324
- **ompt_callback_target_data_op_t**, see Section 4.6.2.21 on page 388.

C / C++

3.6 Tool Control Routines

Summary

The **omp_control_tool** routine enables a program to pass commands to an active tool.

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

Description

An OpenMP program may use **omp_control_tool** to pass commands to a tool. Using **omp_control_tool**, an application can request that a tool start or restart data collection when a code region of interest is encountered, pause data collection when leaving the region of interest, flush any data that it has collected so far, or end data collection. Additionally, **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.

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.

TABLE 3.1: Standard tool control commands.

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

Tool-specific values for *command* must be $\geq$ 64. Tools must ignore *command* values that they are not explicitly designed to handle. Other values accepted by a tool for *command*, and any values for *modifier* and *arg* are tool-defined.

Events

The *tool-control* event occurs in the thread encountering a call to **omp_control_tool** at a point inside its associated OpenMP region.

Tool Callbacks

An OpenMP implementation dispatches a registered **ompt_callback_control_tool** callback for each occurrence of a *tool-control* event. The callback executes in the context of the call that occurs in the 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 331
- **ompt_callback_control_tool_t**, see Section 4.6.2.26 on page 394

CHAPTER 4

Tool Support

This chapter describes OMPT—a tool interface for the OpenMP API. The chapter begins with an overview of the OMPT interface in Section 4.1. Next, it describes how to initialize (Section 4.2) and finalize (Sections 4.3) a tool. Subsequent sections describe details of the interface, including data types shared between an OpenMP implementation and a tool (Section 4.4), an interface that enables an OpenMP implementation to determine that a tool is available (Section 4.5), type signatures for tool callbacks that an OpenMP implementation may dispatch for OpenMP events (Section 4.6), and *runtime entry points*—function interfaces provided by an OpenMP implementation for use by a tool (Section 4.7).

4.1 Overview

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.

4.2 Activating a 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.

4.2.1 Determining Whether a Tool Should be Initialized

A tool indicates its interest in using the OMPT interface by providing a non-**NULL** pointer to an **ompt_fns_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 implmentation 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 5.16, 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_fns_t** structure, the OpenMP implementation will know that a tool is present that wants to use the OMPT interface.

Next, the OpenMP implementation will initialize itself. If a tool provided a non-**NULL** pointer to an **ompt_fns_t** structure, the OpenMP runtime will prepare itself for use of the OMPT interface by a tool.

Cross References

- *tool-var* ICV, see Section 2.3 on page 39.
- *tool-libraries-var* ICV, see Section 2.3 on page 39.

- **ompt_fns_t**, see Section 4.4.1 on page 342.
- **ompt_start_tool**, see Section 4.5.1 on page 363.

4.2.2 Tool Initialization

If a tool-provided implementation of **ompt_start_tool** returns a non-**NULL** pointer to an **ompt_fns_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 4.6.1.1 on page 364 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 4.2.2.1 on page 334. After obtaining a pointer to the OpenMP runtime entry point known as known as **ompt_callback_set** with type signature **ompt_callback_set_t**, the tool initializer should use it to register tool callbacks for OpenMP events, as described in Section 4.2.3 on page 335.

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. The descriptions of the enumeration runtime entry point type signatures show how to use them to determine what thread states and mutual exclusion mechanisms an OpenMP implementation supports.

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_initialize_t**, see Section 4.6.1.1 on page 364.
- **ompt_callback_thread_begin_t**, see Section 4.6.2.1 on page 366.
- **ompt_enumerate_states_t**, see Section 4.7.1.1 on page 398.
- **ompt_enumerate_mutex_impls_t**, see Section 4.7.1.2 on page 400.
- **ompt_callback_set_t**, see Section 4.7.1.3 on page 402.
- **ompt_function_lookup_t**, see Section 4.7.3.1 on page 430.

4.2.2.1 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 inadvertant 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 4.7.1.1 on page 398.
- **`ompt_enumerate_mutex_impls_t`**, see Section 4.7.1.2 on page 400.
- **`ompt_callback_set_t`**, see Section 4.7.1.3 on page 402.
- **`ompt_callback_get_t`**, see Section 4.7.1.4 on page 404.
- **`ompt_get_thread_data_t`**, see Section 4.7.1.5 on page 405.
- **`ompt_get_num_places_t`**, see Section 4.7.1.6 on page 406.
- **`ompt_get_place_proc_ids_t`**, see Section 4.7.1.7 on page 407.
- **`ompt_get_place_num_t`**, see Section 4.7.1.8 on page 408.

- **ompt_get_partition_place_nums_t**, see Section 4.7.1.9 on page 409.
- **ompt_get_procid_t**, see Section 4.7.1.10 on page 410.
- **ompt_get_state_t**, see Section 4.7.1.11 on page 411.
- **ompt_get_parallel_info_t**, see Section 4.7.1.12 on page 412.
- **ompt_get_task_info_t**, see Section 4.7.1.13 on page 414.
- **ompt_get_target_info_t**, see Section 4.7.1.14 on page 416.
- **ompt_get_num_devices_t**, see Section 4.7.1.15 on page 417.
- **ompt_function_lookup_t**, see Section 4.7.3.1 on page 430.

4.2.3 Monitoring Activity on the Host

To monitor execution of an OpenMP program on the host device, a tool's initializer must register to receive notification of events that occur as an OpenMP program executes. A tool can register callbacks for OpenMP events using the runtime entry point known as **ompt_callback_set**. The possible return codes for **ompt_callback_set** and their meanings are shown in Table 4.5. If the **ompt_callback_set** 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_callback_set** or returned by **ompt_callback_get** 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

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_callback_set"	ompt_callback_set_t
"ompt_callback_get"	ompt_callback_get_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_target_info"	ompt_get_target_info_t

TABLE 4.2: Valid return codes of **ompt_callback_set** for each callback.

	ompt_set_never	ompt_set_sometimes	ompt_set_sometimes_paired	ompt_set_always
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_idle	*	*		*
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_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_callback_set_t**, see Section 4.7.1.3 on page 402.
- **ompt_callback_get_t**, see Section 4.7.1.4 on page 404.

4.2.4 Tracing Activity on Target Devices

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 performance monitoring interface for tracing activity on target devices. Tracing activity on a target device involves the following steps:

- To prepare to trace activity on a target device, when a tool initializer executes, it must register a tool **ompt_callback_device_initialize** 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

TABLE 4.3: OMPT tracing interface runtime entry point names and their type signatures.

Entry Point String Name	Type Signature
"**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**

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.
- 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.
- 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. The 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 implementatin 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 the OpenMP implementation while device tracing is in progress.
- When the OpenMP implementation is shut down, any device tracing in progress will be stopped and all trace records collected on each device will be flushed. For each target device, the OpenMP implementation will present the tool with the trace records for the device using the buffer completion callback associated with that device.

Cross References

- **ompt_callback_device_initialize_t**, see Section 4.6.2.28 on page 396.
- **ompt_get_device_time**, see Section 4.7.2.1 on page 418.
- **ompt_translate_time**, see Section 4.7.2.2 on page 419.
- **ompt_set_trace_ompt**, see Section 4.7.2.3 on page 420.
- **ompt_set_trace_native**, see Section 4.7.2.4 on page 421.
- **ompt_start_trace**, see Section 4.7.2.5 on page 422.
- **ompt_pause_trace**, see Section 4.7.2.6 on page 423.
- **ompt_stop_trace**, see Section 4.7.2.7 on page 424.
- **ompt_advance_buffer_cursor**, see Section 4.7.2.8 on page 425.
- **ompt_get_record_type**, see Section 4.7.2.9 on page 426.
- **ompt_get_record_ompt**, see Section 4.7.2.10 on page 427.
- **ompt_get_record_native**, see Section 4.7.2.11 on page 428.
- **ompt_get_record_abstract**, see Section 4.7.2.12 on page 429.

4.3 Finalizing a 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 4.6.1.2 on page 365

4.4 Data Types

4.4.1 Tool Initialization and Finalization

Summary

A tool's implementation of **ompt_start_tool** returns a pointer to an **ompt_fns_t** structure that contains pointers to the tool's initializer and finalizer functions.

C / C++

```
typedef struct ompt_fns_t {
  ompt_initialize_t initialize;
  ompt_finalize_t finalize;
} ompt_fns_t;
```

C / C++

Restrictions

Both the *initialize* and *finalize* function pointers in an **ompt_fns_t** structure returned by **ompt_start_tool** must be non-**NULL**.

Cross References

- **ompt_start_tool**, see Section 4.5.1 on page 363.

4.4.2 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_e {
  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.

4.4.2.1 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**.

4.4.2.2 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_explicit_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_explicit_parallel** for an implicit barrier that occurs at the end of a parallel region. As explained in Section 4.6.2.12 on page 378, 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_explicit_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_explicit_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.

4.4.2.3 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.

4.4.2.4 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.

4.4.2.5 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.

4.4.2.6 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.

4.4.3 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_e {
  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_idle                     = 13,
  ompt_callback_sync_region_wait         = 14,
  ompt_callback_mutex_released           = 15,
  ompt_callback_task_dependences         = 16,
  ompt_callback_task_dependence          = 17,
  ompt_callback_work                     = 18,
  ompt_callback_master                   = 19,
  ompt_callback_target_map               = 20,
  ompt_callback_sync_region              = 21,
  ompt_callback_lock_init                = 22,
  ompt_callback_lock_destroy             = 23,
  ompt_callback_mutex_acquire            = 24,
  ompt_callback_mutex_acquired           = 25,
  ompt_callback_nest_lock                = 26,
  ompt_callback_flush                    = 27,
  ompt_callback_cancel                   = 28
} ompt_callbacks_t;
```

C / C++

4.4.4 Frames

C / C++

```
typedef struct ompt_frame_s {
  void *exit_frame;
  void *enter_frame;
} ompt_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 **ompt_frame_t** object that indicates a contiguous sequence of procedure frames associated with the task. Each **ompt_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 **ompt_frame_t** object.

An **ompt_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 **ompt_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 **ompt_frame_t** object associated with an *initial task* is **NULL**.

The *enter_frame* field of a task's **ompt_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 **ompt_frame_t** object associated with the task may contain **NULL**. The value of *enter_frame* in a task's **ompt_frame_t** is reset to **NULL** just before a suspended task resumes execution.

An **ompt_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 **ompt_frame_t** object is passed to some callbacks; a pointer to a task's **ompt_frame_t** object can also be retrieved by a tool at any time,

TABLE 4.4: Meaning of various states of an **ompt_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

including in a signal handler, by invoking the **ompt_get_task_info** runtime entry point (described in Section 4.7.1.13).

Table 4.4 describes various states in which an **ompt_frame_t** object may be observed and their meaning. In the presence of nested parallelism, a tool may observe a sequence of **ompt_frame_t** objects for a thread. Appendix D illustrates use of **ompt_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 **ompt_frame_t** objects observed just prior to when their field values should be set or reset.

4.4.5 Tracing Support

4.4.5.1 Record Kind

C / C++

```
typedef enum ompt_record_kind_e {
  ompt_record_ompt                = 1,
  ompt_record_native              = 2,
  ompt_record_invalid             = 3
} ompt_record_kind_t;
```

C / C++

4.4.5.2 Native Record Kind

C / C++

```
typedef enum ompt_record_native_kind_e {
  ompt_record_native_info  = 1,
  ompt_record_native_event = 2
} ompt_record_native_kind_t;
```

C / C++

4.4.5.3 Native Record Abstract Type

C / C++

```
typedef struct ompt_record_abstract_s {
  ompt_record_native_class_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**.

4.4.5.4 Record Type

C / C++

```
typedef struct ompt_record_ompt_s {
  ompt_callbacks_t type;
  ompt_target_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_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++

4.4.6 Miscellaneous Type Definitions

This section describes miscellaneous types and enumerations used by the tool interface.

4.4.6.1 ompt_callback_t

Pointers to tool callback functions with many different type signatures are passed to the **ompt_callback_set** runtime entry point and returned by the **ompt_callback_get** runtime entry point. For convenience, these runtime entry points expect all type signatures to be cast to a dummy type **ompt_callback_t**.

C / C++

```
typedef void (*ompt_callback_t)(void);
```

C / C++

4.4.6.2 ompt_id_t

When tracing asynchronous activity on OpenMP devices, tools need identifiers to correlate target regions and operations initiated by the host with associated activities on a target device. In addition, tools need identifiers to refer to parallel regions and tasks that execute on a device. OpenMP implementations use identifiers of type **ompt_id_t** type for each of these purposes. The value **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++

Identifiers created on each device must be unique from the time an OpenMP implementation is initialized until it is shut down. Specifically, this means that (1) identifiers for each target region and target operation instance initiated by the host device must be unique over time on the host, and (2) identifiers for parallel and task region instances that execute on a device must be unique over time within that device.

Tools should not assume that **ompt_id_t** values are small or densely allocated.

4.4.6.3 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_u {
  uint64_t value;
  void *ptr;
} ompt_data_t;

const ompt_data_t ompt_data_none = {.value=0};
```

C / C++

4.4.6.4 ompt_wait_id_t

Each thread instance maintains a *wait identifier* of type **ompt_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 ompt_wait_id_t;
```

C / C++

When a thread is not in a wait state, the value of the thread's wait identifier is undefined.

4.4.6.5 ompt_device_t

ompt_device_t is an opaque object representing a device.

C / C++

```
typedef void ompt_device_t;
```

C / C++

4.4.6.6 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++

4.4.6.7 ompt_buffer_t

ompt_buffer_t is an opaque object handle for a target buffer.

C / C++

```
typedef void ompt_buffer_t;
```

C / C++

4.4.6.8 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++

4.4.6.9 ompt_task_dependence_t

ompt_task_dependence_t is a task dependence.

C / C++

```
typedef struct ompt_task_dependence_s {
  void *variable_addr;
  unsigned int dependence_flags;
} ompt_task_dependence_t;
```

C / C++

4.4.6.10 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_e {
  ompt_thread_initial                = 1,
  ompt_thread_worker                 = 2,
  ompt_thread_other                  = 3,
  ompt_thread_unknown                = 4
} ompt_thread_type_t;
```

C / C++

4.4.6.11 ompt_scope_endpoint_t

ompt_scope_endpoint_t is an enumeration that defines valid scope endpoint values.

C / C++

```
typedef enum ompt_scope_endpoint_e {
  ompt_scope_begin                   = 1,
  ompt_scope_end                     = 2
} ompt_scope_endpoint_t;
```

C / C++

4.4.6.12 `ompt_sync_region_kind_t`

`ompt_sync_region_kind_t` is an enumeration that defines the valid sync region kind values.

C / C++

```
typedef enum ompt_sync_region_kind_e {
  ompt_sync_region_barrier              = 1,
  ompt_sync_region_taskwait             = 2,
  ompt_sync_region_taskgroup            = 3
} ompt_sync_region_kind_t;
```

C / C++

4.4.6.13 `ompt_target_data_op_t`

`ompt_target_data_op_t` is an enumeration that defines the valid target data operation values.

C / C++

```
typedef enum ompt_target_data_op_e {
  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_op_t;
```

C / C++

4.4.6.14 `ompt_work_type_t`

`ompt_work_type_t` is an enumeration that defines the valid work type values.

C / C++

```
typedef enum ompt_work_type_e {
  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++

4.4.6.15 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_e {
  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++

4.4.6.16 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_e {
  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++

1 ### 4.4.6.17 `ompt_task_type_t`

2 **`ompt_task_type_t`** is an enumeration that defines the valid task type values.

C / C++

```
typedef enum ompt_task_type_e {
  ompt_task_initial                   = 1,
  ompt_task_implicit                  = 2,
  ompt_task_explicit                  = 3,
  ompt_task_target                    = 4
} ompt_task_type_t;
```

C / C++

3 ### 4.4.6.18 `ompt_task_status_t`

4 **`ompt_task_status_t`** is an enumeration that explains the reasons for switching a task that
5 reached a task scheduling point.

C / C++

```
typedef enum ompt_task_status_e {
  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 cancelation of that task. The **ompt_task_others** is used in the remaining cases.

4.4.6.19 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_e {
  ompt_target                            = 1,
  ompt_target_enter_data                 = 2,
  ompt_target_exit_data                  = 3,
  ompt_target_update                     = 4
} ompt_target_type_t;
```

C / C++

4.4.6.20 ompt_invoker_t

ompt_invoker_t is an enumeration that defines the valid invoker values.

C / C++

```
typedef enum ompt_invoker_e {
  ompt_invoker_program = 1, /* program invokes master task */
  ompt_invoker_runtime = 2  /* runtime invokes master task */
} ompt_invoker_t;
```

C / C++

4.4.6.21 ompt_target_map_flag_t

ompt_target_map_flag_t is an enumeration that defines the valid target map flag values.

C / C++

```
typedef enum ompt_target_map_flag_e {
  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_t;
```

C / C++

4.4.6.22 ompt_task_dependence_flag_t

ompt_task_dependence_flag_t is an enumeration that defines the valid task dependence flag values.

C / C++

```
typedef enum ompt_task_dependence_flag_e {
  ompt_task_dependence_type_out        = 1,
  ompt_task_dependence_type_in         = 2,
  ompt_task_dependence_type_inout      = 3
} ompt_task_dependence_flag_t;
```

C / C++

4.4.6.23 `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_e {
  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_flag_t;
```

C / C++

Cross References

- **`ompt_cancel_t`** data type, see Section 4.6.2.27 on page 395.

4.4.6.24 `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++

4.5 Tool Interface Routine

4.5.1 `ompt_start_tool`

Summary

If a tool wants to use the OMPT interface provided by an OpenMP implementation, the tool must implement **`ompt_start_tool`** to announce its interest.

Format

C / C++

```
ompt_fns_t *ompt_start_tool(
  unsigned int omp_version,
  const char *runtime_version
);
```

C / C++

Description

For a tool to use the OMPT interface provided by an OpenMP implementation, the tool must define a globally-visible implementation of the function **`ompt_start_tool`**.

A tool may indicate its intent to use the OMPT interface provided by an OpenMP implementation by having **`ompt_start_tool`** return a non-**`NULL`** pointer to an **`ompt_fns_t`** structure, which contains pointers to a tool's initializer and finalizer functions.

A tool may use its argument *omp_version* to determine whether it is compatible with the OMPT interface provided by an OpenMP implementation.

If a tool implements **`ompt_start_tool`** but has no interest in using the OMPT interface in a particular execution, **`ompt_start_tool`** should return **`NULL`**.

Description of Arguments

The argument *omp_version* is the value of the **`_OPENMP`** version macro associated with the OpenMP API implementation. This value identifies the OpenMP API version supported by an OpenMP implementation, which specifies the version of the OMPT interface that it supports.

The argument *runtime_version* is a version string that unambiguously identifies the OpenMP implementation.

Constraints on Arguments

The argument *runtime_version* must be an immutable string that is defined for the lifetime of a program execution.

Effect

If a tool returns a non-**NULL** pointer, an OpenMP implementation will call the tool initializer specified by the *finalize* field in this structure but before beginning execution of any OpenMP construct or completing execution of any environment routine invocation; the OpenMP implementation will call the tool finializer when the OpenMP implementation shuts down.

Cross References

- **ompt_fns_t**, see Section 4.4.1 on page 342.

4.6 Tool Callback Signatures and Trace Records

Restrictions

Tool callbacks may not use OpenMP directives or call any runtime library routines described in Section 3.

4.6.1 Initialization and Finalization Callback Signature

4.6.1.1 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 interace.

Format

C / C++

```
typedef int (*ompt_initialize_t) (
  ompt_function_lookup_t lookup,
  struct ompt_fns_t *fns
);
```

C / C++

Description

For a tool to initialize the OMPT interface of an OpenMP implementation, the tool's implementation of **ompt_start_tool** must return a pointer to a tool initializer with type signature **ompt_initialize_t**. An OpenMP implementation will call the tool initializer returned by **ompt_start_tool** 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 *fns* is the value returned by **ompt_start_tool**. The actions of a tool initializer are described in Section 4.2.2 on page 333.

Cross References

- **ompt_function_lookup_t**, see Section 4.7.3.1 on page 430.

4.6.1.2 ompt_finalize_t

Summary

A tool implements an 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) (
  struct ompt_fns_t *fns
);
```

C / C++

Description

The finalizer for an OpenMP implementation is invoked by an OpenMP implementation as it shuts down.

Description of Arguments

The argument *fns* is the value returned by **ompt_start_tool**.

Cross References

- **ompt_fns_t**, see Section 4.4.1 on page 342.

4.6.2 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.

4.6.2.1 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_s {
  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.

Cross References

- **ompt_data_t** type, see Section 4.4.6.3 on page 353.
- **ompt_thread_type_t** type, see Section 4.4.6.10 on page 356.

4.6.2.2 ompt_callback_thread_end_t

Format

C / C++

```
typedef void (*ompt_callback_thread_end_t) (
  ompt_data_t *thread_data
);
```

C / C++

Description of Arguments

The binding of argument *thread_data* is the thread that is terminating.

Cross References

- **ompt_data_t** type, see Section 4.4.6.3 on page 353.

4.6.2.3 `ompt_callback_idle_t`

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_s {
  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 4.4.6.11 on page 356.

4.6.2.4 `ompt_callback_parallel_begin_t`

Format

C / C++

```
typedef void (*ompt_callback_parallel_begin_t) (
  ompt_data_t *parent_task_data,
  const ompt_frame_t *parent_frame,
  ompt_data_t *parallel_data,
  unsigned int requested_team_size,
  unsigned int actual_team_size,
  ompt_invoker_t invoker,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_parallel_begin_s {
  ompt_id_t parent_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 *parent_task_data* is the encountering task.

The argument *parent_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 *actual_team_size* indicates the number of threads in the team.

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 4.4.6.3 on page 353.
- **ompt_frame_t** type, see Section 4.4.4 on page 349.
- **ompt_invoker_t** type, see Section 4.4.6.20 on page 360.

4.6.2.5 ompt_callback_parallel_end_t

Format

C / C++

```
typedef void (*ompt_callback_parallel_end_t) (
  ompt_data_t *parallel_data,
  ompt_data_t *task_data,
  ompt_invoker_t invoker,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_parallel_end_s {
  ompt_id_t parallel_id;
  ompt_id_t 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 *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 4.4.6.3 on page 353.
- **ompt_invoker_t** type signature, see Section 4.4.6.20 on page 360.

4.6.2.6 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_s {
  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 4.4.6.3 on page 353.
- **ompt_scope_endpoint_t** type, see Section 4.4.6.11 on page 356.

4.6.2.7 `ompt_callback_task_create_t`

Format

C / C++

```
typedef void (*ompt_callback_task_create_t) (
  ompt_data_t *parent_task_data,
  const ompt_frame_t *parent_frame,
  ompt_data_t *new_task_data,
  ompt_task_type_t type,
  int has_dependences,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_task_create_s {
  ompt_id_t parent_task_id;
  ompt_id_t new_task_id;
  ompt_task_type_t type;
  int has_dependences;
  const void *codeptr_ra;
} ompt_record_task_create_t;
```

C / C++

Description of Arguments

The binding of argument *parent_task_data* is the encountering task. This parameter is **`NULL`** for an initial task.

The argument *parent_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.

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 4.4.6.3 on page 353.
- **ompt_frame_t** type, see Section 4.4.4 on page 349.
- **ompt_task_type_t** type, see Section 4.4.6.17 on page 359.

4.6.2.8 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++

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.

Cross References

- **ompt_data_t** type, see Section 4.4.6.3 on page 353.
- **ompt_task_dependence_t** type, see Section 4.4.6.9 on page 355.

4.6.2.9 `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_s {
  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 4.4.6.3 on page 353.

4.6.2.10 `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++

Trace Record

C / C++

```
typedef struct ompt_record_task_schedule_s {
  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++

Description of Arguments

The argument *prior_task_status* indicates the status of the task that arrived at a task scheduling point.

The binding of argument *prior_task_data* is the task that arrived at the scheduling point.

The binding of argument *next_task_data* is the task that will resume at the scheduling point.

Cross References

- **`ompt_data_t`** type, see Section 4.4.6.3 on page 353.
- **`ompt_task_status_t`** type, see Section 4.4.6.18 on page 359.

4.6.2.11 `ompt_callback_implicit_task_t`

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 thread_num
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_implicit_s {
  ompt_scope_endpoint_t endpoint;
  ompt_id_t parallel_id;
  ompt_id_t task_id;
  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.

The binding of argument *task_data* is the implicit task executing the parallel region's structured block.

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 4.4.6.3 on page 353.
- **`ompt_scope_endpoint_t`** enumeration type, see Section 4.4.6.11 on page 356.

4.6.2.12 `ompt_callback_sync_region_t`

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++

Trace Record

C / C++

```
typedef struct ompt_record_sync_region_s {
  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++

Description of Arguments

The argument *kind* indicates the kind of synchronization 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 *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 4.4.6.3 on page 353.
- **ompt_sync_region_kind_t** type, see Section 4.4.6.12 on page 357.
- **ompt_scope_endpoint_t** type, see Section 4.4.6.11 on page 356.

4.6.2.13 ompt_callback_lock_init_t

Format

C / C++

```
typedef void (*ompt_callback_lock_init_t) (
  ompt_mutex_kind_t kind,
  unsigned int hint,
  unsigned int impl,
  ompt_wait_id_t wait_id,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_lock_init_s {
  ompt_mutex_kind_t kind;
  unsigned int hint;
  unsigned int impl;
  ompt_wait_id_t wait_id;
  const void *codeptr_ra;
} ompt_record_lock_init_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.

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

- **`ompt_wait_id_t`** type, see Section 4.4.6.4 on page 354.

4.6.2.14 `ompt_callback_lock_destroy_t`

Format

C / C++

```
typedef void (*ompt_callback_lock_destroy_t) (
  ompt_mutex_kind_t kind,
  ompt_wait_id_t wait_id,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_lock_destroy_s {
  ompt_mutex_kind_t kind;
  ompt_wait_id_t wait_id;
  const void *codeptr_ra;
} ompt_record_lock_destroy_t;
```

C / C++

Description of Arguments

The argument *kind* indicates the kind of the lock.

The argument *wait_id* identifies the lock.

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_wait_id_t** type, see Section 4.4.6.4 on page 354.

4.6.2.15 ompt_callback_mutex_acquire_t

Format

C / C++

```
typedef void (*ompt_callback_mutex_acquire_t) (
  ompt_mutex_kind_t kind,
  unsigned int hint,
  unsigned int impl,
  ompt_wait_id_t wait_id,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_mutex_acquire_s {
  ompt_mutex_kind_t kind;
  unsigned int hint;
  unsigned int impl;
  ompt_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

- **ompt_wait_id_t** type, see Section 4.4.6.4 on page 354.
- **ompt_mutex_kind_t** type, see Section 4.4.6.15 on page 358.

4.6.2.16 `ompt_callback_mutex_t`

Format

C / C++

```
typedef void (*ompt_callback_mutex_t) (
  ompt_mutex_kind_t kind,
  ompt_wait_id_t wait_id,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_mutex_s {
  ompt_mutex_kind_t kind;
  ompt_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

- **ompt_wait_id_t** type signature, see Section 4.4.6.4 on page 354.
- **ompt_mutex_kind_t** type signature, see Section 4.4.6.15 on page 358.

4.6.2.17 `ompt_callback_nest_lock_t`

Format

C / C++

```
typedef void (*ompt_callback_nest_lock_t) (
  ompt_scope_endpoint_t endpoint,
  ompt_wait_id_t wait_id,
  const void *codeptr_ra
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_nest_lock_s {
  ompt_scope_endpoint_t endpoint;
  ompt_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

- **ompt_wait_id_t** type signature, see Section 4.4.6.4 on page 354.
- **ompt_scope_endpoint_t** type signature, see Section 4.4.6.11 on page 356.

4.6.2.18 `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_s {
  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.

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 2.7 on page 61.
- **ompt_data_t** type signature, see Section 4.4.6.3 on page 353.
- **ompt_scope_endpoint_t** type signature, see Section 4.4.6.11 on page 356.
- **ompt_work_type_t** type signature, see Section 4.4.6.14 on page 357.

4.6.2.19 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_s {
  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 4.4.6.3 on page 353.

4.6.2.20 ompt_callback_target_t

Format

C / C++

```
typedef void (*ompt_callback_target_t) (
  ompt_target_type_t kind,
  ompt_scope_endpoint_t endpoint,
  int device_id,
  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_s {
  ompt_target_type_t kind;
  ompt_scope_endpoint_t endpoint;
  int device_id;
  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_id* 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 4.4.6.2 on page 353.
- **ompt_data_t** type signature, see Section 4.4.6.3 on page 353.
- **ompt_scope_endpoint_t** type signature, see Section 4.4.6.11 on page 356.
- **ompt_target_type_t** type signature, see Section 4.4.6.19 on page 360.

4.6.2.21 ompt_callback_target_data_op_t

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 *host_addr,
  void *device_addr,
  size_t bytes
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_target_data_op_s {
  ompt_id_t host_op_id;
  ompt_target_data_op_t optype;
  void *host_addr;
  void *device_addr;
  size_t bytes;
  ompt_device_time_t end_time;
} ompt_record_target_data_op_t;
```

C / C++

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 *host_addr* indicates the address of data on host side.

The argument *device_addr* indicates the address of data on device side.

The argument *bytes* indicates the size of data.

Cross References

- **ompt_id_t** type, see Section 4.4.6.2 on page 353.
- **ompt_target_data_op_t** type signature, see Section 4.4.6.13 on page 357.

4.6.2.22 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
);
```

C / C++

Trace Record

C / C++

```
typedef struct ompt_record_target_map_s {
  ompt_id_t target_id;
  unsigned int nitems;
  void **host_addr;
  void **device_addr;
  size_t *bytes;
  unsigned int *mapping_flags;
} ompt_record_target_map_t;
```

C / C++

Description of Arguments

The binding of argument *target_id* is the target region.

The argument *nitems* indicates the number of data mappings.

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.

Cross References

- **`ompt_id_t`** type, see Section 4.4.6.2 on page 353.

4.6.2.23 `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
);
```

C / C++

Description

This callback is invoked 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.

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_s {
  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++

Cross References

- **ompt_id_t** type, see Section 4.4.6.2 on page 353.

4.6.2.24 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) (
  int device_id,
  ompt_buffer_t **buffer,
  size_t *bytes
);
```

C / C++

Description

The 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_id* 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 4.4.6.7 on page 355.

4.6.2.25 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) (
  int device_id,
  const ompt_buffer_t *buf,
  size_t bytes,
  ompt_buffer_cursor_t begin,
  int buffer_owned
);
```

C / C++

Description

The 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_id* 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 4.4.6.7 on page 355.
- **ompt_buffer_cursor_t** type, see Section 4.4.6.8 on page 355.

4.6.2.26 ompt_callback_control_tool_t

Format

C / C++

```
typedef int (*ompt_callback_control_tool_t) (
  uint64_t command,
  uint64_t modifier,
  void *arg
);
```

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 3.6 on page 327.

The argument *modifier* passes a command modifier from an application to a tool.

The 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**.

Constraints on Arguments

Tool-specific values for *command* must be ≥ 64.

Cross References

- **omp_control_tool_t** enumeration type, see Section 3.6 on page 327.

4.6.2.27 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 4.4.6.23 on page 362.

4.6.2.28 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) (
  int device_id,
  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_id* 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 4.2.4 on page 338.

Cross References

- **ompt_function_lookup_t**, see Section 4.7.3.1 on page 430.

4.7 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.

4.7.1 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.

4.7.1.1 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 4.4.2 on page 342.

4.7.1.2 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++

4.7.1.3 ompt_callback_set_t

Summary

A runtime entry point known as **ompt_callback_set** with type signature **ompt_callback_set_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_callback_set_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_callback_set** with type signature **ompt_callback_set_t**.

The return value of the **ompt_callback_set** 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_callback_set** 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.

TABLE 4.5: Return codes for **ompt_callback_set** and **ompt_set_trace_ompt**.

```
typedef enum ompt_set_result_e {
  ompt_set_error              = 0,
  ompt_set_none               = 1,
  ompt_set_sometimes          = 2,
  ompt_set_sometimes_paired = 3,
  ompt_set_always             = 4
} ompt_set_result_t;
```

Cross References

- **ompt_callbacks_t** enumeration type, see Section 4.4.3 on page 347.
- **ompt_callback_t** type, see Section 4.4.6.1 on page 353.
- **ompt_callback_get_t** host callback type signature, see Section 4.7.1.4 on page 404.

4.7.1.4 ompt_callback_get_t

Summary

A runtime entry point known as **ompt_callback_get** with type signature **ompt_callback_get_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_callback_get_t)(
  ompt_callbacks_t which,
  ompt_callback_t *callback
);
```

C / C++

Description

A tool uses the runtime entry point known as **ompt_callback_get** with type signature **ompt_callback_get_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* is a pointer to a return value that will be assigned the value of 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 4.4.3 on page 347.
- **ompt_callback_t** type, see Section 4.4.6.1 on page 353.
- **ompt_callback_set_t** type signature, see Section 4.7.1.3 on page 402.

4.7.1.5 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 4.4.6.3 on page 353.

4.7.1.6 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 2.3 on page 39.
- **OMP_PLACES** environment variable, see Section 5.5 on page 437.

4.7.1.7 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.

Format

C / C++

```
typedef int (*ompt_get_place_proc_ids_t)(
    int place_num,
    int ids_size,
    int *ids
  );
```

C / C++

Binding

The binding thread set for the region of the runtime entry point known as **ompt_get_place_proc_ids** 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_place_proc_ids`** with type signature **`ompt_get_place_proc_ids_t`** returns the numerical identifiers of each processor associated with the specified place. The numerical identifiers returned are non-negative, and their meaning is implementation defined.

Description of Arguments

The argument *place_num* specifies the place being queried.

The argument *ids_size* indicates the size of the result array specified by argument *ids*.

The argument *ids* is an array where the routine can return a vector of processor identifiers in the specified place.

Effect

If the array *ids* of size *ids_size* is large enough to contain all identifiers, they are returned in *ids* and their order in the array is implementation defined.

Otherwise, if the *ids* array is too small, the values in *ids* are unchanged.

In both cases, the routine returns the number of numerical identifiers available to the execution environment in the specified place.

4.7.1.8 `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 encountering thread is bound.

Format

C / C++

```
typedef int (*ompt_get_place_num_t)(void);
```

C / C++

Binding

The binding thread set for the region of the runtime entry point known as **ompt_get_place_num** is the encountering thread.

Description

When the encountering 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 encountering thread is not bound to a place, the routine returns -1.

This runtime entry point is *async signal safe*.

4.7.1.9 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 region of the runtime entry point known as **ompt_get_partition_place_nums** is the encountering 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.

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 2.3 on page 39.
- **OMP_PLACES** environment variable, see Section 5.5 on page 437.

4.7.1.10 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 encountering thread.

Format

C / C++

```
typedef int (*ompt_get_proc_id_t)(void);
```

C / C++

Binding

The binding thread set for the region of the runtime entry point known as **ompt_get_proc_id** is the encountering thread.

Description

The runtime entry point known as **ompt_get_proc_id** returns the numerical identifier of the processor of the encountering thread. The numerical identifier is non-negative, and its meaning is implementation defined.

This runtime entry point is *async signal safe*.

4.7.1.11 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)(
  ompt_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

- **ompt_wait_id_t** type, see Section 4.4.6.4 on page 354.

4.7.1.12 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 1 if there is a parallel region at the specified ancestor level 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, 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, 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

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 4.4.6.3 on page 353.

4.7.1.13 `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,
  ompt_task_type_t *type,
  ompt_data_t **task_data,
  ompt_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 1 if there is a task region at the specified ancestor level and 0 otherwise.

If a task exists at the specified ancestor level, information will be returned in the variables passed by reference to the entry point. If no task region exists at the specified ancestor level, 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* is pointer to a task type return value or a **NULL** if no task type return value is required.

The argument *task_data* is a pointer to a task data pointer return value or a **NULL** if no task data pointer return value is required.

The argument *task_frame* is a pointer to a task frame pointer return value or a **NULL** if no task frame pointer return value is required.

The argument *parallel_data* is a pointer to a parallel data pointer return value or a **NULL** if no parallel data pointer return value is required.

The argument *thread_num* is a pointer to a return value for a thread number or a **NULL** if no thread number return value is required.

Effect

If the runtime entry point returns 0, no return values will be set. 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 **ompt_frame_t** structure associated with the task at the specified level. Appendix D discusses an example that illustrates the use of **ompt_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 4.4.6.3 on page 353.

- **ompt_frame_t** type, see Section 4.4.4 on page 349.
- **ompt_task_type_t** type, see Section 4.4.6.17 on page 359.

4.7.1.14 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)(
  int *device_id,
  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 invoking 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 invoking 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

The argument *device_id* is a pointer to a return value for the current device. If the host is in a **target** region, the target device will be returned in **device_id*.

The argument *target_id* is a pointer to a return value for the target region identifier. If the host is in a **target** region, the **target** region identifier will be returned in **target_id*.

The argument *host_op_id* is a pointer to a return value for an identifer for an operation being initiated on a **target** device. If the invoking 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) the identifier for the operation being initiated will be returned in **host_op_id*; otherwise, **host_op_id*. will be set to **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 4.4.6.2 on page 353.

4.7.1.15 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*.

4.7.2 Entry Points in the OMPT Device Tracing Interface

4.7.2.1 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 4.4.6.5 on page 354.
- **ompt_device_time_t**, see Section 4.4.6.6 on page 355.

4.7.2.2 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 4.4.6.5 on page 354.
- **ompt_device_time_t**, see Section 4.4.6.6 on page 355.

4.7.2.3 `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.

Format

C / C++

```
typedef int (*ompt_set_trace_ompt_t)(
  ompt_device_t *device,
  unsigned int enable,
  unsigned int etype
);
```

C / C++

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 *enable* indicates whether tracing should be enabled or disabled for the event or events specified by argument *etype*. A positive value for *enable* indicates that recording of one or more events specified by *etype* should be enabled; a value of 0 for *enable* indicates that recording of events should be disabled by this invocation.

An argument *etype* value 0 indicates that traces for all event types will be enabled or disabled. Passing a positive value for *etype* inidicates that recording should be enabled or disabled for the event in **ompt_callbacks_t** that matches *etype*.

Effect

Table 4.6 shows the possible return codes for **ompt_set_trace_ompt**. If a single invocation of **ompt_set_trace_ompt** is used to enable or disable more than one event (i.e., **etype**=0), the return code will be 3 if tracing is possible for one or more events but not for others.

Cross References

- **ompt_callbacks_t**, see Section 4.4.3 on page 347.
- **ompt_device_t**, see Section 4.4.6.5 on page 354.

TABLE 4.6: Meaning of return codes for **ompt_trace_set_ompt** and **ompt_set_trace_native**.

return code	meaning
0	error
1	event will never occur
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

4.7.2.4 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.

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 *enable* indicates whether recording of events should be enabled or disabled by this invocation.

The argument *flags* specifies the kinds of native device monitoring to enable or disable. Each kind of monitoring is specified by a flag bit. Flags can be composed by using logical `or` to combine enumeration values from type **`ompt_native_mon_flags_t`**. Table 4.6 shows the possible return codes for **`ompt_set_trace_native`**. If a single invocation of **`ompt_set_trace_ompt`** is used to enable/disable more than one kind of monitoring, the return code will be 3 if tracing is possible for one or more kinds of monitoring but not for others.

To start, pause, or stop tracing for a specific target device associated with the handle *device*, a tool calls the functions **`ompt_start_trace`**, **`ompt_pause_trace`**, or **`ompt_stop_trace`**.

Cross References

- **`ompt_device_t`**, see Section 4.4.6.5 on page 354.

4.7.2.5 `ompt_start_trace_t`

Summary

A runtime entry point for a device known as **`ompt_start_trace`** with type signature **`ompt_start_trace_t`** starts tracing of activity on a specific device.

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,
  ompt_callback_get_target_info_t get_info
);
```

C / C++

Description

This runtime entry point enables tracing on a device. It provides tool callbacks that the device uses to request a buffer from a tool for recording events and a second calback that the device uses to return a buffer containing events to the tool.

Under normal operating conditions, every event buffer provided to a device by the tool 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.

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.

The argument *get_info* is a function that a device can use to map device activity back to identifiers that indicate where the activity was initiated by the host.

Cross References

- **ompt_device_t**, see Section 4.4.6.5 on page 354.
- **ompt_callback_buffer_request_t**, see Section 4.6.2.24 on page 392.
- **ompt_callback_buffer_complete_t**, see Section 4.6.2.25 on page 393.

4.7.2.6 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.

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*. The entry point will return 0 if the request fails, e.g., if tracing for a device has not been started, and return a non-zero return code otherwise. Redundant pause or resume commands are idempotent and will return a non-zero value indicating success.

Cross References

- **ompt_device_t**, see Section 4.4.6.5 on page 354.

4.7.2.7 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

Each invocation returns 1 if the command succeeded and 0 otherwise. A call to **ompt_stop_trace** also implicitly requests that the device flush any buffers that it owns.

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 4.4.6.5 on page 354.

4.7.2.8 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* is a pointer to a return value for the next value of a opaque buffer cursor.

Cross References

- **ompt_device_t**, see Section 4.4.6.5 on page 354.
- **ompt_buffer_cursor_t**, see Section 4.4.6.8 on page 355.

4.7.2.9 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 4.4.6.7 on page 355.
- **ompt_buffer_cursor_t**, see Section 4.4.6.8 on page 355.

4.7.2.10 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 4.4.5.4 on page 352.
- **ompt_device_t**, see Section 4.4.6.5 on page 354.
- **ompt_buffer_cursor_t**, see Section 4.4.6.8 on page 355.

4.7.2.11 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 4.4.6.2 on page 353.
- **`ompt_buffer_t`**, see Section 4.4.6.7 on page 355.
- **`ompt_buffer_cursor_t`**, see Section 4.4.6.8 on page 355.

4.7.2.12 `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 4.4.5.3 on page 351.

4.7.3 Lookup Entry Point

4.7.3.1 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 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 336 for a list and Section 4.7.1 on page 398 for detailed definitions.
- Tool initializer for a device's OMPT tracing interface, Section 4.2.4 on page 338.
- Entry points in the OMPT tracing interface, see Table 4.3 on page 339 for a list and Section 4.7.2 on page 418 for detailed definitions.
- Tool initializer for the OMPT callback interface, Section 4.6.1.1 on page 364

CHAPTER 5

Environment Variables

This chapter describes the OpenMP environment variables that specify the settings of the ICVs that affect the execution of OpenMP programs (see Section 2.3 on page 39). The names of the environment variables must be upper case. The values assigned to the environment variables are case insensitive and may have leading and trailing white space. Modifications to the environment variables after the program has started, even if modified by the program itself, are ignored by the OpenMP implementation. However, the settings of some of the ICVs can be modified during the execution of the OpenMP program by the use of the appropriate directive clauses or OpenMP API routines.

The environment variables are as follows:

- **`OMP_SCHEDULE`** sets the *run-sched-var* ICV that specifies the runtime schedule type and chunk size. It can be set to any of the valid OpenMP schedule types.
- **`OMP_NUM_THREADS`** sets the *nthreads-var* ICV that specifies the number of threads to use for parallel regions.
- **`OMP_DYNAMIC`** sets the *dyn-var* ICV that specifies the dynamic adjustment of threads to use for **`parallel`** regions.
- **`OMP_PROC_BIND`** sets the *bind-var* ICV that controls the OpenMP thread affinity policy.
- **`OMP_PLACES`** sets the *place-partition-var* ICV that defines the OpenMP places that are available to the execution environment.
- **`OMP_NESTED`** sets the *nest-var* ICV that enables or disables nested parallelism.
- **`OMP_STACKSIZE`** sets the *stacksize-var* ICV that specifies the size of the stack for threads created by the OpenMP implementation.
- **`OMP_WAIT_POLICY`** sets the *wait-policy-var* ICV that controls the desired behavior of waiting threads.
- **`OMP_MAX_ACTIVE_LEVELS`** sets the *max-active-levels-var* ICV that controls the maximum number of nested active **`parallel`** regions.

- **OMP_THREAD_LIMIT** sets the *thread-limit-var* ICV that controls the maximum number of threads participating in a contention group.
- **OMP_CANCELLATION** sets the *cancel-var* ICV that enables or disables cancellation.
- **OMP_DISPLAY_ENV** instructs the runtime to display the OpenMP version number and the initial values of the ICVs, once, during initialization of the runtime.
- **OMP_DEFAULT_DEVICE** sets the *default-device-var* ICV that controls the default device number.
- **OMP_MAX_TASK_PRIORITY** sets the *max-task-priority-var* ICV that specifies the maximum value that can be specified in the **priority** clause of the **task** construct.
- **OMP_TOOL** sets the *tool-var* ICV that controls whether or not an OpenMP will try to register a performance tool.
- **OMP_TOOL_LIBRARIES** sets the *tool-libraries-var* ICV that contains a list of tool libraries that the runtime searches to find one appropriate for use on a device where an OpenMP implementation is being initialized.

The examples in this chapter only demonstrate how these variables might be set in Unix C shell (csh) environments. In Korn shell (ksh) and DOS environments the actions are similar, as follows:

- csh:

```
setenv OMP_SCHEDULE "dynamic"
```

- ksh:

```
export OMP_SCHEDULE="dynamic"
```

- DOS:

```
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 2.7.1 on page 62 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 274.

Examples:

```
setenv OMP_SCHEDULE "guided,4"
setenv OMP_SCHEDULE "dynamic"
```

Cross References

- *run-sched-var* ICV, see Section 2.3 on page 39.
- Loop construct, see Section 2.7.1 on page 62.
- Parallel loop construct, see Section 2.11.1 on page 140.
- **omp_set_schedule** routine, see Section 3.2.12 on page 274.
- **omp_get_schedule** routine, see Section 3.2.13 on page 276.

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 2.3 on page 39 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 2.5.1 on page 55 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 2.3 on page 39.
- **num_threads** clause, Section 2.5 on page 50.
- **omp_set_num_threads** routine, see Section 3.2.1 on page 262.
- **omp_get_num_threads** routine, see Section 3.2.2 on page 263.
- **omp_get_max_threads** routine, see Section 3.2.3 on page 264.
- **omp_get_team_size** routine, see Section 3.2.19 on page 282.

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 2.3 on page 39.
- **omp_set_dynamic** routine, see Section 3.2.7 on page 268.
- **omp_get_dynamic** routine, see Section 3.2.8 on page 270.

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 2.3 on page 39.
- **proc_bind** clause, see Section 2.5.2 on page 57.
- **omp_get_proc_bind** routine, see Section 3.2.22 on page 285.

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.

$$
\begin{array}{rcl}
\langle \text{list} \rangle & \models & \langle \text{p-list} \rangle \mid \langle \text{aname} \rangle \\
\langle \text{p-list} \rangle & \models & \langle \text{p-interval} \rangle \mid \langle \text{p-list} \rangle , \langle \text{p-interval} \rangle \\
\langle \text{p-interval} \rangle & \models & \langle \text{place} \rangle : \langle \text{len} \rangle : \langle \text{stride} \rangle \mid \langle \text{place} \rangle : \langle \text{len} \rangle \mid \langle \text{place} \rangle \mid ! \langle \text{place} \rangle \\
\langle \text{place} \rangle & \models & \{ \langle \text{res-list} \rangle \} \\
\langle \text{res-list} \rangle & \models & \langle \text{res-interval} \rangle \mid \langle \text{res-list} \rangle , \langle \text{res-interval} \rangle \\
\langle \text{res-interval} \rangle & \models & \langle \text{res} \rangle : \langle \text{num-places} \rangle : \langle \text{stride} \rangle \mid \langle \text{res} \rangle : \langle \text{num-places} \rangle \mid \langle \text{res} \rangle \mid ! \langle \text{res} \rangle \\
\langle \text{aname} \rangle & \models & \langle \text{word} \rangle (\langle \text{num-places} \rangle) \mid \langle \text{word} \rangle \\
\langle \text{word} \rangle & \models & \text{sockets} \mid \text{cores} \mid \text{threads} \mid \text{<implementation-defined abstract name>} \\
\langle \text{res} \rangle & \models & \textit{non-negative integer} \\
\langle \text{num-places} \rangle & \models & \textit{positive integer} \\
\langle \text{stride} \rangle & \models & \textit{integer} \\
\langle \text{len} \rangle & \models & \textit{positive integer}
\end{array}
$$

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 2.3 on page 39.
- Controlling OpenMP thread affinity, Section 2.5.2 on page 57.
- **omp_get_num_places** routine, see Section 3.2.23 on page 287.
- **omp_get_place_num_procs** routine, see Section 3.2.24 on page 288.
- **omp_get_place_proc_ids** routine, see Section 3.2.25 on page 289.
- **omp_get_place_num** routine, see Section 3.2.26 on page 290.
- **omp_get_partition_num_places** routine, see Section 3.2.27 on page 291.
- **omp_get_partition_place_nums** routine, see Section 3.2.28 on page 292.

5.6 OMP_NESTED

The **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 2.3 on page 39.
- **omp_set_nested** routine, see Section 3.2.10 on page 271.
- **omp_get_team_size** routine, see Section 3.2.19 on page 282.

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 2.3 on page 39.

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 2.3 on page 39.

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 2.3 on page 39.
- **omp_set_max_active_levels** routine, see Section 3.2.15 on page 277.
- **omp_get_max_active_levels** routine, see Section 3.2.16 on page 279.

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 2.3 on page 39.
- **omp_get_thread_limit** routine, see Section 3.2.14 on page 277.

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 2.3.1 on page 39.
- **cancel** construct, see Section 2.14.1 on page 197.
- **cancellation point** construct, see Section 2.14.2 on page 202.
- **omp_get_cancellation** routine, see Section 3.2.9 on page 271.

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='201611'
  [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_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 2.3 on page 39.
- device constructs, Section 2.10 on page 106.

5.14 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 2.3 on page 39.
- Tasking Constructs, see Section 2.9 on page 91.
- **omp_get_max_task_priority** routine, see Section 3.2.36 on page 299.

5.15 OMP_TOOL

The **OMP_TOOL** environment variable sets the *tool-var* ICV which controls whether an OpenMP runtime will try to register a performance 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 2.3 on page 39.
- Tool Interface, see Section 4 on page 331.

5.16 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 comma-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 2.3 on page 39.
- Tool Interface, see Section 4 on page 331.
- **ompt_start_tool** routine, see Section 4.5.1 on page 363.

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

Note – 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_default_device(int device_num)
{
}

int omp_get_default_device(void)
{
    return 0;
}

int omp_get_num_devices(void)
{
    return 0;
}

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;
    if (nlock->owner == MASTER && nlock->count >= 1)
    {
        nlock->count++;
    }
    else if (nlock->owner == NOOWNER && nlock->count == 0)
    {
        nlock->owner = MASTER;
        nlock->count = 1;
    }
    else
    {
        fprintf(stderr, "error: lock corrupted or not initialized\n");
        exit(1);
    }
}

void omp_unset_nest_lock(omp_nest_lock_t *arg)
{
    struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
    if (nlock->owner == MASTER && nlock->count >= 1)
    {
        nlock->count--;
        if (nlock->count == 0)
```

```
                {
                    nlock->owner = NOOWNER;
                }
            }
            else if (nlock->owner == NOOWNER && nlock->count == 0)
            {
                fprintf(stderr, "error: lock not set\n");
                exit(1);
            }
            else
            {
                fprintf(stderr, "error: lock corrupted or not initialized\n");
                exit(1);
            }
        }

        int omp_test_nest_lock(omp_nest_lock_t *arg)
        {
            struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
            omp_set_nest_lock(arg);
            return nlock->count;
        }

        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;
    }
```

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
```

```
    if ( level .eq. 0 ) then
       omp_get_ancestor_thread_num = 0
    else
       omp_get_ancestor_thread_num = -1
    end if
    return
  end function

  integer function omp_get_team_size(level)
    integer level
    if ( level .eq. 0 ) then
       omp_get_team_size = 1
    else
       omp_get_team_size = -1
    end if
    return
  end function

  integer function omp_get_active_level()
    omp_get_active_level = 0
    return
  end function

  logical function omp_in_final()
    omp_in_final = .true.
    return
  end function

  function omp_get_proc_bind()
    include 'omp_lib_kinds.h'
    integer (kind=omp_proc_bind_kind) omp_get_proc_bind
    omp_get_proc_bind = omp_proc_bind_false
  end function

  integer function omp_get_num_places()
    return 0
  end function

  integer function omp_get_place_num_procs(place_num)
    integer place_num
    return 0
  end function

  subroutine omp_get_place_proc_ids(place_num, ids)
    integer place_num
    integer ids(*)
    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_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_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
```

This page intentionally left blank

APPENDIX B

Interface Declarations

This appendix gives examples of the C/C++ header file, the Fortran **`include`** file and Fortran **`module`** that shall be provided by implementations as specified in Chapter 3. It also includes an example of a Fortran 90 generic interface for a library routine. This is a non-normative section, 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 lock hints
 */
typedef enum omp_lock_hint_t
{
 omp_lock_hint_none = 0,
 omp_lock_hint_uncontended = 1,
 omp_lock_hint_contended = 2,
 omp_lock_hint_nonspeculative = 4,
 omp_lock_hint_speculative = 8
/* , Add vendor specific constants for lock hints here,
   starting from the most-significant bit. */
} 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;

/*
 * 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_default_device(int device_num);
extern int omp_get_default_device(void);
extern int omp_get_num_devices(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);

#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_lock_hint_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 ) )
      parameter ( omp_lock_hint_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_lock_hint_kind ) omp_lock_hint_none
      parameter ( omp_lock_hint_none = 0 )
      integer ( omp_lock_hint_kind ) omp_lock_hint_uncontended
      parameter ( omp_lock_hint_uncontended = 1 )
      integer ( omp_lock_hint_kind ) omp_lock_hint_contended
      parameter ( omp_lock_hint_contended = 2 )
```

```
        integer ( omp_lock_hint_kind ) omp_lock_hint_nonspeculative
        parameter ( omp_lock_hint_nonspeculative = 4 )
        integer ( omp_lock_hint_kind ) omp_lock_hint_speculative
        parameter ( omp_lock_hint_speculative = 8 )

        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 )
```

omp_lib.h:

```
!     default integer type assumed below
!     default logical type assumed below
!     OpenMP API v5.0 Preview 1 (TR4)

        include 'omp_lib_kinds.h'
        integer openmp_version
        parameter ( openmp_version = 201611 )

        external omp_set_num_threads
        external omp_get_num_threads
        integer omp_get_num_threads
        external omp_get_max_threads
        integer omp_get_max_threads
        external omp_get_thread_num
        integer omp_get_thread_num
        external omp_get_num_procs
        integer omp_get_num_procs
        external omp_in_parallel
        logical omp_in_parallel
```

```
external omp_set_dynamic
external omp_get_dynamic
logical omp_get_dynamic
external omp_get_cancellation
logical omp_get_cancellation
external omp_set_nested
external omp_get_nested
logical omp_get_nested
external omp_set_schedule
external omp_get_schedule
external omp_get_thread_limit
integer omp_get_thread_limit
external omp_set_max_active_levels
external omp_get_max_active_levels
integer omp_get_max_active_levels
external omp_get_level
integer omp_get_level
external omp_get_ancestor_thread_num
integer omp_get_ancestor_thread_num
external omp_get_team_size
integer omp_get_team_size
external omp_get_active_level
integer omp_get_active_level
external omp_set_default_device
external omp_get_default_device
integer omp_get_default_device
external omp_get_num_devices
integer omp_get_num_devices
external omp_get_num_teams
integer omp_get_num_teams
external omp_get_team_num
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
```

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


      module omp_lib

        use omp_lib_kinds

!                                     OpenMP API v5.0 Preview 1 (TR4)
        integer, parameter :: openmp_version = 201611

       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_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_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

  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 11).
- **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 18).
- **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 1.4.4 on page 21).
- **Internal control variables**: the initial values of *dyn-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*, and *default-device-var* are implementation defined. The method for initializing a target device's internal control variable is implementation defined (see Section 2.3.2 on page 40).
- **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 2.5.1 on page 55).

- **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 2.5.2 on page 57).

- **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 2.7.1 on page 62).

- **`sections` construct**: the method of scheduling the structured blocks among threads in the team is implementation defined (see Section 2.7.2 on page 71).

- **`single` construct**: the method of choosing a thread to execute the structured block is implementation defined (see Section 2.7.3 on page 74)

- **`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 2.8.1 on page 80).

- **`declare simd` construct**: 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 2.8.2 on page 84).

- **`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 2.9.2 on page 95).

- **`is_device_ptr` clause**: Support for pointers created outside of the OpenMP device data management routines is implementation defined (see Section 2.10.5 on page 116).

- **`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 2.10.5 on page 116).

- **`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 2.10.8 on page 129).

- **`distribute` construct**: the integer type (or kind, for Fortran) used to compute the iteration count for the collapsed loop is implementation defined (see Section 2.10.9 on page 132).
- **`distribute` construct**: If no **`dist_schedule`** clause is specified then the schedule for the **`distribute`** construct is implementation defined (see Section 2.10.9 on page 132).
- **`critical` construct**: the effect of using a **`hint`** clause is implementation defined (see Section 2.13.2 on page 167).
- **`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 2.13.7 on page 178).

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 2.15.1.2 on page 209).
- **`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 2.15.2 on page 210).
- **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 260).

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 262).
- **`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 274).
- **`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 277).

- **`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 279).

- **`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 289).

- **`omp_get_initial_device` routine**: the value of the device number is implementation defined (see Section 3.2.35 on page 298).

- **`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 304).

- **`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 322).

- **`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 434).

- **`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 435).

- **`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 436).

- **`OMP_DYNAMIC` environment variable**: if the value is neither **`true`** nor **`false`** the behavior is implementation defined (see Section 5.3 on page 436).

- **`OMP_NESTED` environment variable**: if the value is neither **`true`** nor **`false`** the behavior is implementation defined (see Section 5.6 on page 439).

- **`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 440).

- **`OMP_WAIT_POLICY` environment variable**: the details of the **`ACTIVE`** and **`PASSIVE`** behaviors are implementation defined (see Section 5.8 on page 441).

- **`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 442).

- **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 442).
- **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 437).
- **OMPT thread states**: The set of OMPT thread states supported is implementation defined (see Section 4.4.2 on page 342).
- **ompt_callback_idle tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **ompt_callback_set**, 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 337)
- **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_callback_set**, 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 337)
- **ompt_callback_mutex_released tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **ompt_callback_set**, 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 337)
- **ompt_callback_task_dependences tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **ompt_callback_set**, 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 337)
- **ompt_callback_task_dependence tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **ompt_callback_set**, 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 337)
- **ompt_callback_work tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **ompt_callback_set**, 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 337)

- **`ompt_callback_master` tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **`ompt_callback_set`**, 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 337)
- **`ompt_callback_target_map` tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **`ompt_callback_set`**, 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 337)
- **`ompt_callback_sync_region` tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **`ompt_callback_set`**, 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 337)
- **`ompt_callback_lock_init` tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **`ompt_callback_set`**, 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 337)
- **`ompt_callback_lock_destroy` tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **`ompt_callback_set`**, 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 337)
- **`ompt_callback_mutex_acquire` tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **`ompt_callback_set`**, 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 337)
- **`ompt_callback_mutex_acquired` tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **`ompt_callback_set`**, 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 337)
- **`ompt_callback_nest_lock` tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **`ompt_callback_set`**, 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 337)
- **`ompt_callback_flush` tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **`ompt_callback_set`**, 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 337)

- **`ompt_callback_cancel` tool callback**: if a tool attempts to register a callback with this string name using the runtime entry point **`ompt_callback_set`**, 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 337)
- **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 4.2.4 on page 338).
- **`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 4.2.4 on page 338).
- **`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 4.2.4 on page 338).

APPENDIX D

Task Frame Management for the Tool Interface

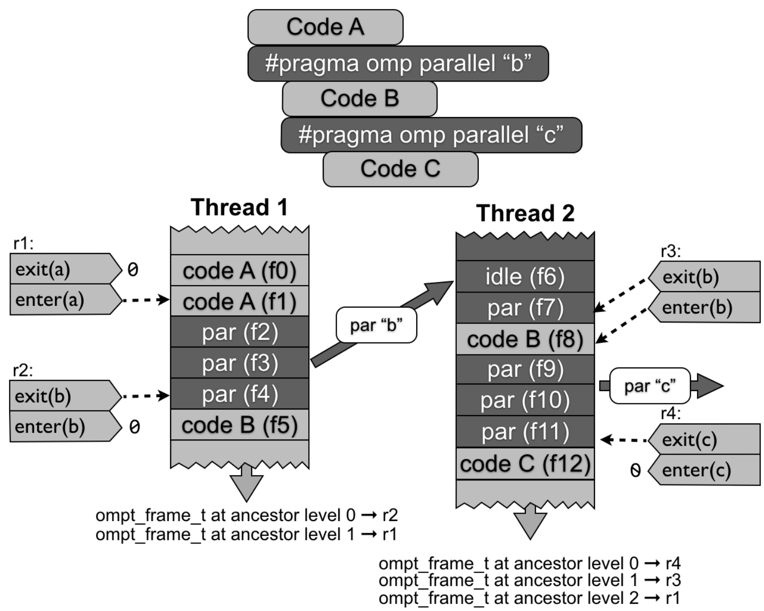

FIGURE D.1: Thread call stacks implementing nested parallelism annotated with frame information for the OMPT tool interface.

The top half of Figure D.1 illustrates a conceptualization of a program executing a nested parallel region, where code A, B, and C represent, respectively, one or more procedure frames of code associated with an initial task, an outer parallel region, and an inner parallel region. The bottom half of Figure D.1 illustrates the stacks of two threads executing the nested parallel region. In the 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 **ompt_frame_t** for the initial task executing code A to frame f1—the user frame in the initial task that calls the runtime. The **ompt_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 **ompt_frame_t** (labeled *r2*) to 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 **ompt_frame_t** (labeled *r3*) to 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 **ompt_frame_t** (labeled *r3*) to 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 **ompt_frame_t** (labeled *r4*) for the implicit task for "c" to 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 **ompt_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 **ompt_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

- **ompt_frame_t**, see Section 4.4.4 on page 349.

APPENDIX E

Features History

This appendix summarizes the major changes between recent versions of the OpenMP API since version 2.5.

E.1 Version 4.5 to 5.0 Differences

- 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 28 and Section 2.13.10 on page 194).
- To support taskloop reductions, the **reduction** and **in_reduction** clauses were added to the **taskloop** (see Section 2.9.2 on page 95) and **taskloop simd** (see Section 2.9.3 on page 100) constructs.
- The **depend** clause was added to the **taskwait** construct (see Section 2.13.5 on page 174).
- To support conditional assignment to lastprivate variables, the **conditional** modifier was added to the **lastprivate** clause (see Section 2.15.3.5 on page 225).
- To support task reductions, the **task_reduction** clause was added to the **taskgroup** construct (see Section 2.15.4.5 on page 238) and the **in_reduction** clause to the **task** construct (see Section 2.15.4.6 on page 239).
- To reduce programmer effort implicit declare target directives for some functions (C, C++, Fortran) and subroutines (Fortran) were added (see Section 2.10.5 on page 116 and Section 2.10.7 on page 124).
- Support for a tool interface was added (see Section 4 on page 331).

E.2 Version 4.0 to 4.5 Differences

- Support for several features of Fortran 2003 was added (see Section 1.7 on page 23 for features that are still not supported).
- A parameter was added to the **ordered** clause of the loop construct (see Section 2.7.1 on page 62) and clauses were added to the **ordered** construct (see Section 2.13.9 on page 190) 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 2.7.1 on page 62).
- The **simdlen** clause was added to the **simd** construct (see Section 2.8.1 on page 80) to support specification of the exact number of iterations desired per SIMD chunk.
- The **priority** clause was added to the **task** construct (see Section 2.9.1 on page 91) 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 3.2.36 on page 299) and the **OMP_MAX_TASK_PRIORITY** environment variable was added to control the maximum priority value allowed (see Section 5.14 on page 445).
- Taskloop constructs (see Section 2.9.2 on page 95 and Section 2.9.3 on page 100) 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 2.10.2 on page 107) and the **is_device_ptr** clause was added to the **target** construct (see Section 2.10.5 on page 116).
- The **nowait** and **depend** clauses were added to the **target** construct (see Section 2.10.5 on page 116) to improve support for asynchronous execution of **target** regions.
- The **private**, **firstprivate** and **defaultmap** clauses were added to the **target** construct (see Section 2.10.5 on page 116).
- 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 2.10.7 on page 124).
- To support unstructured data mapping for devices, the **target enter data** (see Section 2.10.3 on page 109) and **target exit data** (see Section 2.10.4 on page 112) constructs were added and the **map** clause (see Section 2.15.6.1 on page 245) was updated.
- To support a more complete set of device construct shortcuts, the **target parallel** (see Section 2.11.5 on page 146), target parallel loop (see Section 2.11.6 on page 148), target parallel loop SIMD (see Section 2.11.7 on page 149), and **target simd** (see Section 2.11.8 on page 151), 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 2.12 on page 164).
- The **hint** clause was addded to the **critical** construct (see Section 2.13.2 on page 167).
- The **source** and **sink** dependence types were added to the **depend** clause (see Section 2.13.10 on page 194) to support doacross loop nests.
- The implicit data-sharing attribute for scalar variables in **target** regions was changed to **firstprivate** (see Section 2.15.1.1 on page 205).
- Use of some C++ reference types was allowed in some data sharing attribute clauses (see Section 2.15.3 on page 215).
- 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 2.15.4.4 on page 236).
- The **ref**, **val**, and **uval** modifiers were added to the **linear** clause (see Section 2.15.3.6 on page 228).
- Support was added to the map clauses to handle structure elements (see Section 2.15.6.1 on page 245).
- Query functions for OpenMP thread affinity were added (see Section 3.2.23 on page 287 to Section 3.2.28 on page 292).
- 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 304).
- Device memory routines were added to allow explicit allocation, deallocation, memory transfers and memory associations (see Section 3.5 on page 317).
- C/C++ Grammar (previously Appendix B) was moved to a separate document.

E.3 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 23).
- C/C++ array syntax was extended to support array sections (see Section 2.4 on page 48).
- The **proc_bind** clause (see Section 2.5.2 on page 57), the **OMP_PLACES** environment variable (see Section 5.5 on page 437), and the **omp_get_proc_bind** runtime routine (see Section 3.2.22 on page 285) were added to support thread affinity policies.

- SIMD constructs were added to support SIMD parallelism (see Section 2.8 on page 80).
- Device constructs (see Section 2.10 on page 106), the **OMP_DEFAULT_DEVICE** environment variable (see Section 5.13 on page 444), 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 2.9.6 on page 104).
- The **depend** clause (see Section 2.13.10 on page 194) was added to support task dependences.
- The **taskgroup** construct (see Section 2.13.6 on page 176) was added to support more flexible deep task synchronization.
- The **reduction** clause (see Section 2.15.4.4 on page 236) was extended and the **declare reduction** construct (see Section 2.16 on page 250) was added to support user defined reductions.
- The **atomic** construct (see Section 2.13.7 on page 178) 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 2.14.1 on page 197), the **cancellation point** construct (see Section 2.14.2 on page 202), the **omp_get_cancellation** runtime routine (see Section 3.2.9 on page 271) and the **OMP_CANCELLATION** environment variable (see Section 5.11 on page 442) were added to support the concept of cancellation.
- The **OMP_DISPLAY_ENV** environment variable (see Section 5.12 on page 443) was added to display the value of ICVs associated with the OpenMP environment variables.
- Examples (previously Appendix A) were moved to a separate document.

E.4 Version 3.0 to 3.1 Differences

- The **final** and **mergeable** clauses (see Section 2.9.1 on page 91) were added to the **task** construct to support optimization of task data environments.
- The **taskyield** construct (see Section 2.9.4 on page 102) was added to allow user-defined task scheduling points.
- The **atomic** construct (see Section 2.13.7 on page 178) 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 2.15.3.4 on page 223).
- Data environment restrictions were changed to allow Fortran pointers in `firstprivate` (see Section 2.15.3.4 on page 223) and `lastprivate` (see Section 2.15.3.5 on page 225).
- New reduction operators `min` and `max` were added for C and C++
- The nesting restrictions in Section 2.17 on page 256 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 284) 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 435), but the algorithm for determining the number of threads used in a parallel region has been modified to handle a list (see Section 2.5.1 on page 55).
- The *bind-var* ICV has been added, which controls whether or not threads are bound to processors (see Section 2.3.1 on page 39). The value of this ICV can be set with the `OMP_PROC_BIND` environment variable (see Section 5.4 on page 436).
- 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 478 and Section B.4 on page 485) with `selected_int_kind(8)`.

E.5 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 9 and Section 1.3 on page 15).

- The `task` construct (see Section 2.9 on page 91) has been added, which provides a mechanism for creating tasks explicitly.
- The `taskwait` construct (see Section 2.13.5 on page 174) 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 18). 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 2.3 on page 39). 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 262, Section 3.2.7 on page 268 and Section 3.2.10 on page 271).
- 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 2.5.1 on page 55).
- In Version 3.0, the assignment of iterations to threads in a loop construct with a **static** schedule kind is deterministic (see Section 2.7.1 on page 62).
- 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 2.7.1 on page 62).
- 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 2.7.1 on page 62).
- 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 2.7.1 on page 62).
- Fortran assumed-size arrays now have predetermined data-sharing attributes (see Section 2.15.1.1 on page 205).
- In Fortran, **firstprivate** is now permitted as an argument to the **default** clause (see Section 2.15.3.1 on page 216).
- For list items in the **private** clause, implementations are no longer permitted to use the storage of the original list item to hold the new list item on the master thread. If no attempt is made to reference the original list item inside the **parallel** region, its value is well defined on exit from the **parallel** region (see Section 2.15.3.3 on page 218).
- In Version 3.0, Fortran allocatable arrays may appear in **private**, **firstprivate**, **lastprivate**, **reduction**, **copyin** and **copyprivate** clauses. (see Section 2.15.2 on page 210, Section 2.15.3.3 on page 218, Section 2.15.3.4 on page 223, Section 2.15.3.5 on page 225, Section 2.15.4.4 on page 236, Section 2.15.5.1 on page 240 and Section 2.15.5.2 on page 242).
- In Version 3.0, static class members variables may appear in a **threadprivate** directive (see Section 2.15.2 on page 210).

- Version 3.0 makes clear where, and with which arguments, constructors and destructors of private and threadprivate class type variables are called (see Section 2.15.2 on page 210, Section 2.15.3.3 on page 218, Section 2.15.3.4 on page 223, Section 2.15.5.1 on page 240 and Section 2.15.5.2 on page 242).
- The runtime library routines **omp_set_schedule** and **omp_get_schedule** have been added; these routines respectively set and retrieve the value of the *run-sched-var* ICV (see Section 3.2.12 on page 274 and Section 3.2.13 on page 276).
- The *thread-limit-var* ICV has been added, which controls the maximum number of threads participating in the OpenMP program. The value of this ICV can be set with the **OMP_THREAD_LIMIT** environment variable and retrieved with the **omp_get_thread_limit** runtime library routine (see Section 2.3.1 on page 39, Section 3.2.14 on page 277 and Section 5.10 on page 442).
- The *max-active-levels-var* ICV has been added, which controls the number of nested active **parallel** regions. The value of this ICV can be set with the **OMP_MAX_ACTIVE_LEVELS** environment variable and the **omp_set_max_active_levels** runtime library routine, and it can be retrieved with the omp_get_max_active_levels runtime library routine (see Section 2.3.1 on page 39, Section 3.2.15 on page 277, Section 3.2.16 on page 279 and Section 5.9 on page 442).
- The *stacksize-var* ICV has been added, which controls the stack size for threads that the OpenMP implementation creates. The value of this ICV can be set with the **OMP_STACKSIZE** environment variable (see Section 2.3.1 on page 39 and Section 5.7 on page 440).
- The *wait-policy-var* ICV has been added, which controls the desired behavior of waiting threads. The value of this ICV can be set with the **OMP_WAIT_POLICY** environment variable (see Section 2.3.1 on page 39 and Section 5.8 on page 441).
- The **omp_get_level** runtime library routine has been added, which returns the number of nested **parallel** regions enclosing the task that contains the call (see Section 3.2.17 on page 280).
- 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 281).
- 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 282).
- 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 283).
- In Version 3.0, locks are owned by tasks, not by threads (see Section 3.3 on page 301).

Index