OpenMP
Application Programming Interface

Version ~~4.0~~4.1rev0, ~~July~~November, ~~2013~~2014

This page intentionally left blank in published version.

This is Revision 0 (15 Oct 2014) and includes the following tickets applied to the 4.0 LaTeX sources: 267, 268, 271, 279, 282, 284, 285, 288, 290, 298, 300, 302, 309-311, 314, 316, 318, 323-326, 328, 330, 332-336, 338, 341, 343, 344, 352, 355.

This is a draft - DO NOT DISTRIBUTE

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 shared-memory parallelism in C, C++ and Fortran programs.

This specification provides a model for parallel programming that is portable across shared memory architectures from different vendors. Compilers from numerous vendors support the OpenMP API. 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 manage parallel programs while permitting portability. The directives extend the C, C++ and Fortran base languages with single program multiple data (SPMD) constructs, tasking constructs, device constructs, worksharing constructs, and synchronization constructs, and they provide support for sharing and privatizing data. The functionality to control the runtime environment is provided by library routines and environment variables. Compilers that support the OpenMP API often include a command line option to the compiler that activates and allows interpretation of all OpenMP directives.

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.

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 1.6 on page 20 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.
- 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*.
- 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 25 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 the 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.

loop directive An OpenMP *executable directive* whose 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`** clause then there may be more than one *associated loop*.

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, in 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`** *directive* is encountered is a part of the *region* of the *encountering thread*, but the *explicit task region* associated with the **`task`** *directive* is not. 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 The *thread* that encounters a `parallel` *construct*, creates a *team*, generates a set of *implicit tasks*, 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 `target` construct or a `teams` construct.

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

implicit parallel region An *inactive parallel region* that generates an *initial task region*. *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* encountered during the execution of another *region*.

> COMMENT: Some nestings are *conforming* and some are not. See Section 2.16 on page 197 for the restrictions on nesting.

closely nested region A *region nested* inside another *region* with no **`parallel`** *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*. Note that 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 whose 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*, 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* whose *binding thread* set is *all threads* or the *encountering thread*, nor is it defined for *regions* whose *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*.

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* whose *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`**.

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

place Unordered set of *processors* 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.

SIMD instruction A single machine instruction that can 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*.

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

1.2.3 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 2.13.1 on page 148

1.2.4 Tasking Terminology

task A specific instance of executable code and its *data environment*, generated when a *thread* encounters a **`task`** *construct* or a **`parallel`** *construct*.

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*.

explicit task A *task* generated when a **`task`** *construct* is encountered during execution.

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

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* 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 2.9.3 on page 84.

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* whose *data environment*, inclusive of ICVs, is the same as that of its generating *task region*.

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*.

1.2.5 Data Terminology

variable A named data storage block, whose 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.

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

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

private variable With respect to a given set of *task regions* or *SIMD lanes* that bind to the same **`parallel`** *region*, a *variable* whose 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* whose 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 A *data environment* defined by a **`target data`** or **`target`** construct.

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.
- All data members must be non-static.
- A *mappable type* cannot contain virtual members.

For Fortran: The type must be definable.

In addition, 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

1.2.6 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 D.

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.

1.3 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.

The initial thread that executes the implicit parallel region that surrounds the **`target`** region may execute on a *target device*. An initial thread executes sequentially, as if enclosed in an implicit task region, called an initial task region, that is defined by an implicit inactive **`parallel`** region that surrounds the entire **`target`** region.

When a **`target`** construct is encountered, the **`target`** region is executed by the implicit device task. The task that encounters the **`target`** construct waits at the end of the construct until execution of the region completes. If a target device does not exist, or the target device is not supported by the implementation, or the target device cannot execute the **target** construct then the **target** region is executed by the host device.

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** construct, a new explicit task is 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 2.14.3.3 on page 168 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 2.14 on page 154.

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, each device has an initial device data environment. The initial device data environment for the host device is the data environment associated with the initial task region. Directives that accept data-mapping attribute clauses determine how an original variable is mapped to a corresponding variable in a device data environment. The original variable is the variable with the same name that exists in the data environment of the task that encounters the directive.

If a corresponding variable is present in the enclosing device data environment, the new device data environment ~~inherits~~ uses the corresponding variable from the enclosing device data environment. If a corresponding variable is not present in the enclosing device data environment, a new corresponding variable (of the same type and size) is created in the new device data environment. In the latter case, 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 19. 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 2.14.5 on page 187.

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 2.12 on page 129 and in Section 3.3 on page 232, 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 17 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 2.12.7 on page 142 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 16

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 OpenMP Compliance

An implementation of the OpenMP API is compliant if and only if it compiles and executes all conforming programs according to the syntax and semantics laid out in Chapters 1, 2, 3 and 4. Appendices A, B, C, D, E and F and sections designated as Notes (see Section 1.7 on page 22) 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 D 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 D.

1.6 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
  - ~~Allocatable enhancement~~

    Parameterized derived types
  - ~~Finalization~~

    ~~Procedures bound by name to a type~~

    The **PASS** attribute
  - Procedures bound to a type as operators
  - ~~Type extension~~

    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.

1.7 Organization of this document

The remainder of this document is structured as follows:

- Chapter 2 “Directives”
- Chapter 3 “Runtime Library Routines”
- Chapter 4 “Environment Variables”
- Appendix A “Stubs for Runtime Library Routines”
- Appendix B “OpenMP C and C++ Grammar”
- Appendix C “Interface Declarations”
- Appendix D “OpenMP Implementation-Defined Behaviors”
- 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 whose base language is C or C++ is shown as follows:

C / C++

C/C++ specific text...

C / C++

Text that applies only to programs whose base language is C only is shown as follows:

C

C specific text...

C

Text that applies only to programs whose base language is C90 only is shown as follows:

C90

C90 specific text...

C90

Text that applies only to programs whose base language is C99 only is shown as follows:

C99

C99 specific text...

C99

Text that applies only to programs whose base language is C++ only is shown as follows:

C++

C++ specific text...

C++

Text that applies only to programs whose 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 25)
- Mechanisms to control conditional compilation (Section 2.2 on page 31)
- How to specify and to use array sections for all base languages (Section 2.4 on page 42)
- Control of OpenMP API ICVs (Section 2.3 on page 34)
- Details of each OpenMP directive (Section 2.5 on page 43 to Section 2.16 on page 197)

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 may not appear in **PURE** or **ELEMENTAL** 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 formally specified by the grammar in Appendix B, and informally 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 27 and Section 2.1.2 on page 28.

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 105). 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 2.14.3 on page 164), data copying clauses (Section 2.14.4 on page 183), the **threadprivate** directive (Section 2.14.2 on page 159) and the **flush** directive (Section 2.12.7 on page 142) accept a *list*. A *list* consists of a comma-separated collection of one or more *list items*.

C / C++

A *list item* is a variable or array section, subject to the restrictions specified in Section 2.4 on page 42 and in each of the sections describing clauses and directives for which a *list* appears.

C / C++

Fortran

A *list item* is a variable, array section or common block name (enclosed in slashes), subject to the restrictions specified in Section 2.4 on page 42 and in each of the sections describing clauses and directives for which a *list* appears.

Fortran

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

Fortran (cont.)

```
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 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 task group
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 teams
target teams distribute
target teams distribute parallel do
target teams distribute parallel do simd
target teams distribute simd
target update
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`**. See Appendix B for the formal grammar.

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. Continued 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 35.

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 the 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 the ICV for the whole program (the scope is global).
- *default-device-var* - controls the default target device. There is one copy of this ICV per data environment

2.3.2 ICV Initialization

The following table shows the ICVs, associated environment variables, and initial values:

ICV	Environment Variable	Initial value
dyn-var	**`OMP_DYNAMIC`**	See comments 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

table continued on next page

table continued from previous page

ICV	Environment Variable	Initial value
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 comments 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

Comments

- 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 1.2.6 on page 12 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 4.1 on page 244.
- **OMP_NUM_THREADS** environment variable, see Section 4.2 on page 245.
- **OMP_DYNAMIC** environment variable, see Section 4.3 on page 246.
- **OMP_PROC_BIND** environment variable, see Section 4.4 on page 246.

- **OMP_PLACES** environment variable, see Section 4.5 on page 247.
- **OMP_NESTED** environment variable, see Section 4.6 on page 249.
- **OMP_STACKSIZE** environment variable, see Section 4.7 on page 249.
- **OMP_WAIT_POLICY** environment variable, see Section 4.8 on page 250.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 4.9 on page 251.
- **OMP_THREAD_LIMIT** environment variable, see Section 4.10 on page 251.
- **OMP_CANCELLATION** environment variable, see Section 4.11 on page 252.
- **OMP_DEFAULT_DEVICE** environment variable, see Section 4.13 on page 253.

2.3.3 Modifying and Retrieving ICV Values

The following table shows the method for modifying and retrieving the values of ICVs through OpenMP API routines:

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_levels()**
levels-var	(none)	**omp_get_level()**
place-partition-var	(none)	(none)

table continued on next page

table continued from previous page

ICV	Ways to modify value	Ways to retrieve value
cancel-var	(none)	**omp_get_cancellation()**
default-device-var	**omp_set_default_device()**	**omp_get_default_device()**

Comments

- 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.

Cross References

- **thread_limit** clause of the **teams** construct, see Section 2.10.5 on page 95.
- **omp_set_num_threads** routine, see Section 3.2.1 on page 200.
- **omp_get_max_threads** routine, see Section 3.2.3 on page 202.
- **omp_set_dynamic** routine, see Section 3.2.7 on page 206.
- **omp_get_dynamic** routine, see Section 3.2.8 on page 208.
- **omp_get_cancellation** routine, see Section 3.2.9 on page 209.
- **omp_set_nested** routine, see Section 3.2.10 on page 209.
- **omp_get_nested** routine, see Section 3.2.11 on page 211.
- **omp_set_schedule** routine, see Section 3.2.12 on page 212.
- **omp_get_schedule** routine, see Section 3.2.13 on page 214.
- **omp_get_thread_limit** routine, see Section 3.2.14 on page 215.
- **omp_set_max_active_levels** routine, see Section 3.2.15 on page 216.
- **omp_get_max_active_levels** routine, see Section 3.2.16 on page 217.
- **omp_get_level** routine, see Section 3.2.17 on page 218.
- **omp_get_active_level** routine, see Section 3.2.20 on page 222.
- **omp_get_proc_bind** routine, see Section 3.2.22 on page 224.
- **omp_set_default_device** routine, see Section 3.2.23 on page 226.
- **omp_get_default_device** routine, see Section 3.2.24 on page 227.

2.3.4 How ICVs are Scoped

The following table shows the ICVs and their scope:

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	device
default-device-var	data environment

Comments

- 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 device construct is encountered, the new device data environment inherits the values of the data environment scoped ICVs from the enclosing device data environment 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 new device data environment is not inherited but instead is 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

The override relationships among construct clauses and ICVs are shown in the following table:

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

table continued on next page

table continued from previous page

ICV	construct clause, if used
levels-var	(none)
place-partition-var	(none)
cancel-var	(none)
default-device-var	(none)

Comments

- 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 elements of the *bind-var* ICV; otherwise, the **`proc_bind`** clause has no effect.

Cross References

- **`parallel`** construct, see Section 2.5 on page 43.
- **`proc_bind`** clause, Section 2.5 on page 43.
- **`num_threads`** clause, see Section 2.5.1 on page 47.
- Loop construct, see Section 2.7.1 on page 54.
- **`schedule`** clause, see Section 2.7.1.1 on page 60.

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 *lower-bound* and *length* must evaluate to non-negative integers.

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

Fortran has built-in support for array sections but the following restrictions apply for OpenMP constructs:

- A stride expression may not be specified.
- The upper bound for the last dimension of an assumed-size dummy array must be specified.

Fortran

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

C / C++

C++

- An array section cannot be used in a C++ user-defined `[]`-operator.

C++

2.5 `parallel` Construct

Summary

This fundamental construct starts parallel execution. See Section 1.3 on page 13 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(*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 (*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 47 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 49, 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 78).

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 47, 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.

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

- **`default`**, **`shared`**, **`private`**, **`firstprivate`**, and **`reduction`** clauses, see Section 2.14.3 on page 164.
- **`copyin`** clause, see Section 2.14.4 on page 183.
- **`omp_get_thread_num`** routine, see Section 3.2.4 on page 204.

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.

Note that 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 34.

2.5.2 Controlling OpenMP Thread Affinity

~~When creating a team for a parallel region, the~~

When a thread encounters a parallel directive without a **`proc_bind`** clause~~specifies a~~, 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 SUBSCRIPTNBbind`** clause then the binding policy specified by the **`proc SUBSCRIPTNBbind`** 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 ~~master thread executes on the parent's place and the remaining~~ 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 ~~execute on places from the place list consecutive from the parent's position in the list,~~ to places is as follows:

- $T \le P$. 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 ~~parent thread's implicit task. The place partition is not changed by this policy, and each implicit task inherits the *place-partition-var* ICV of the parent implicit task.~~ master thread.

- $T > P$ Each place P will contain S_p threads with consecutive thread numbers, where $\lfloor (T/P) \rfloor \le Sp \le \lceil (T/P) \rceil$. 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. ~~It accomplishes this~~ A sparse distribution is achieved by first subdividing the parent partition into T subpartitions if ~~T is less than or equal to P~~$T \le P$, or P subpartitions if ~~T is greater than P. Then it assigns one thread ($T \le P$)~~ $T > P$. Then one thread ($T \le P$) or a set of threads ~~($T > P$)~~ ($T > P$) 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 ~~'s~~ thread's place partition is split into T subpartitions, where each subpartition contains ~~at least $S = floor(P/T)$~~ $\lfloor (P/T) \rfloor$ 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. ~~For the other threads, assignment is~~ The thread with the next smallest thread number is assigned to the first place in the ~~corresponding subpartition. When T does not divide P evenly, the assignment of the remaining $P - T * S$ places into subpartitions is implementation defined~~next subpartition, and so on, with wrap around with respect to the original place partition of the master thread.

- $T > P$. The parent ~~'s~~ thread's place partition is split into P ~~unit-sized subpartitions. Each place is assigned $S = floor(T/P)$ threads . When P does not divide T evenly, the assignment of the remaining $T - P * S$ threads into places is implementation defined.~~

  ~~For the **`close`** and **`spread`** thread affinity policies, the threads with the smallest threadnumbers execute on the~~ 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 ~~master thread, then the threads with the next smaller thread numbers execute on the next place in the partition;~~ parent thread. The next S_1 threads are assigned to the next subpartition, and so on, with wrap around with respect to the ~~encountering thread's place partition .~~ 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 ~~number~~ affinity of threads in the team ~~and their mapping to places are implementation defined.~~ 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** on the loop construct, its value after the loop is unspecified.

continued on next page

continued from previous page

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

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

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

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

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

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

Restrictions

The following restrictions also apply:

- If *test-expr* is of the form *var relational-op b* and *relational-op* is < or <= then *incr-expr* must cause *var* to increase on each iteration of the loop. If *test-expr* is of the form *var relational-op b* and *relational-op* is > or >= then *incr-expr* must cause *var* to decrease on each iteration of the loop.
- If *test-expr* is of the form *b relational-op var* and *relational-op* is < or <= then *incr-expr* must cause *var* to decrease on each iteration of the loop. If *test-expr* is of the form *b relational-op var* and *relational-op* is > or >= then *incr-expr* must cause *var* to increase on each iteration of the loop.
- For C++, in the **`simd`** construct the only random access iterator types that are allowed for *var* are pointer types.

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(*list***)**

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

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

collapse(*n***)**

ordered

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 51).

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(*list***)**

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

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

collapse(*n***)**

ordered

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

All associated *do-loops* must be *do-constructs* 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. Unless the loop iteration variables are specified **lastprivate** or **linear** on the loop construct, their values after the loop are unspecified.

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 no **`collapse`** clause is present, the only loop that is associated with the loop construct is the one that immediately follows the loop directive.

If more than one loop is associated with the loop construct, then the iterations of all associated loops are collapsed into one larger iteration space that is then divided according to the **`schedule`** clause. 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 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 the associated loop(s) were executed by a single thread. The **`schedule`** clause specifies how iterations of the 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 *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 2-1. 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 60 for details of how the schedule for a worksharing loop is determined.

The schedule *kind* can be one of those specified in Table 2-1.

TABLE 2-1 **`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. Note that the size of the chunks is unspecified in this case. A compliant implementation of the **`static`** schedule must ensure that the same assignment of logical iteration numbers to threads will be used in two loop regions if the following conditions are satisfied: 1) both loop regions have the same number of loop iterations, 2) both loop regions have the same value of *chunk_size* specified, or both loop regions have no *chunk_size* specified, 3) both loop regions bind to the same parallel region, and 4) neither loop is associated with a SIMD construct. A data dependence between the same logical iterations in two such loops is guaranteed to be satisfied allowing safe use of the **`nowait`** clause.
`dynamic`	When **`schedule(dynamic,`** *chunk_size*) is specified, the iterations are distributed to threads in the team in chunks as the threads request them. 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 last chunk to be distributed, 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 as the executing threads request them. 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 last chunk to be assigned, which may have fewer than k iterations).

table continued on next page

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

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 $2 * p * k$.

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 the 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.
- 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 loop iteration variable may not appear in a **threadprivate** directive.

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 2.14.3 on page 164.
- **OMP_SCHEDULE** environment variable, see Section 4.1 on page 244.
- **ordered** construct, see Section 2.12.8 on page 146.

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 34 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 34

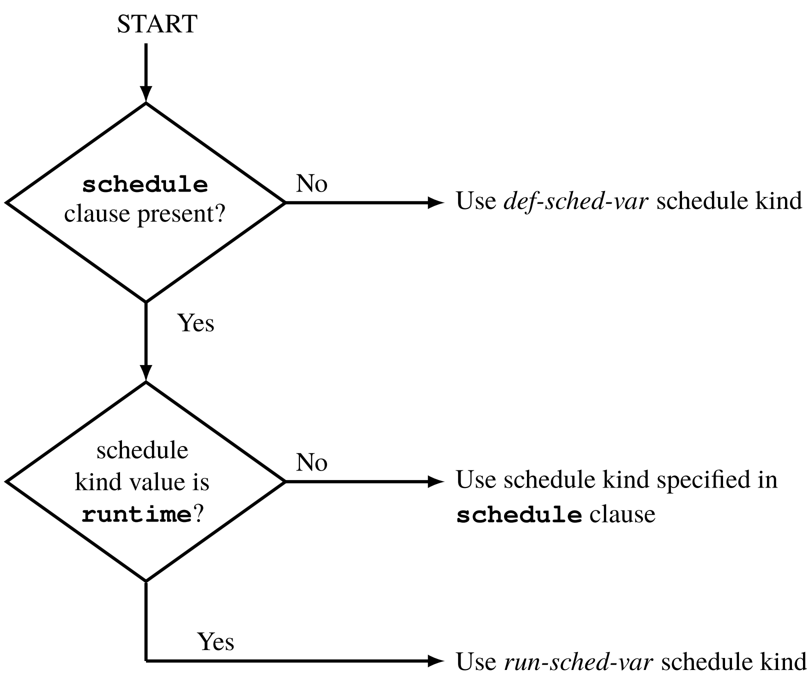

FIGURE 2-1 Determining the schedule for a worksharing loop.

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(`*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(*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.

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 2.14.3 on page 164.

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.

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 2.14.3 on page 164.
- **`copyprivate`** clause, see Section 2.14.4.2 on page 185.

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.

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

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***)**

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

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

private(*list***)**

lastprivate(*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 51).

C / C++

Fortran

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

where *clause* is one of the following:

safelen(*length***)**

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

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

private(*list***)**

lastprivate(*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*.

All associated *do-loops* must be *do-constructs* 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. 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

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.

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.
- No OpenMP construct can appear in the **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.
- 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

- **private**, **lastprivate**, **linear** and **reduction** clauses, see Section 2.14.3 on page 164.

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 from 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(`*argument-list[* **`:`** *constant-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(*argument-list*[**:** *constant-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 from 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 from a SIMD loop concurrently.

Fortran

If a **declare simd** directive contains multiple SIMD declarations, then one or more SIMD versions will be created for each declaration.

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.

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 function will always be called from inside a conditional statement of a SIMD loop. The **notinbranch** clause specifies that the function will never be called from inside a conditional statement of a SIMD loop. If neither clause is specified, then 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 *constant-linear-step* expression is specified in a **linear** clause it must be a constant positive integer expression.
- 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.
- 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.
- 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 2.14.3.6 on page 176.
- **linear** clause, see Section 2.14.3.7 on page 182.

2.8.3 Loop SIMD construct

Summary

The loop SIMD construct specifies ~~a loop that can be executed concurrently using SIMD instructions and that those iterations will also be executed in parallel by threads~~ 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-loop~~*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.

Restrictions

All restrictions to the loop construct and the **simd** construct apply to the loop SIMD construct. In addition, the following restriction applies:

- No **ordered** clause 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 54.
- **simd** construct, see Section 2.8.1 on page 68.
- Data attribute clauses, see Section 2.14.3 on page 164.

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(*scalar-expression* **)**

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

untied

default(shared | none)

mergeable

private(*list* **)**

firstprivate(*list* **)**

shared(*list* **)**

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

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

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

untied

default(private | firstprivate | shared | none)

mergeable

private(*list* **)**

firstprivate(*list* **)**

shared(*list* **)**

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

Fortran

Binding

The binding thread set of the **task** region is the current team. A **task** region binds to the innermost enclosing **parallel** region.

Description

When a thread encounters a **task** construct, a 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. A **task** construct may be nested inside an outer task, but the **task** region of the inner task is not a part of the **task** region of the outer 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. Note that 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 a **mergeable** clause is present on a **task** construct, and the generated task is an undeferred task or an included task, the implementation may generate a merged task instead.

Note – When storage is shared by an explicit **task** region, it is the programmer's responsibility to ensure, by adding proper synchronization, that the storage does not reach the end of its lifetime before the explicit **task** region completes its execution.

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.

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

2.9.1.1 depend Clause

Summary

The **depend** clause enforces additional constraints on the scheduling of tasks. These constraints establish dependences only between sibling tasks. The clause consists of a *dependence-type* with one or more list items.

Syntax

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

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

Description

Task dependences are derived from the *dependence-type* of a **depend** clause and its list items, where *dependence-type* is one of the following:

The **in** *dependence-type*. The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an **out** or **inout** *dependence-type* list.

The **out** and **inout** *dependence-types*. The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an **in**, **out**, or **inout** *dependence-type* list.

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

Note – The enforced task dependence establishes a synchronization of memory accesses performed by a dependent task with respect to accesses performed by the predecessor tasks. However, it is the responsibility of the programmer to synchronize properly with respect to other concurrent accesses that occur outside of those tasks.

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 or disjoint storage.
- List items used in **depend** clauses cannot be zero-length array sections.
- A variable that is part of another variable (such as a field of a structure) but is not an array element or an array section cannot appear in a **depend** clause.

Cross References

- Array sections, Section 2.4 on page 42.
- Task scheduling constraints, Section 2.9.3 on page 84.

2.9.2 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 2.9.3 on page 84.

2.9.3 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

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.

2.10 Device Constructs

2.10.1 target data Construct

Summary

Create 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:

device (*integer-expression***)**

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

if (*scalar-expression***)**

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:

device (*scalar-integer-expression***)**

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

if (*scalar-logical-expression***)**

The **end target data** directive denotes the end of the **target data** construct.

Fortran

Binding

The binding task region for a **`target data`** construct is the encountering task. The target region binds to the enclosing parallel or task region.

Description

When a **`target data`** construct is encountered, a new device data environment is created, and the encountering task executes the **`target data`** region. If there is no **`device`** clause, the default device is determined by the *default-device-var* ICV. The new device data environment is constructed from the enclosing device data environment, the data environment of the encountering task and any data-mapping clauses on the construct. When an **`if`** clause is present and the **`if`** clause expression evaluates to *false*, the device is the host.

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.
- At most one **`if`** clause can appear on the directive.

Cross References

- **`map`** clause, see Section 2.14.5 on page 187.
- *default-device-var*, see Section 2.3 on page 34.

2.10.2 `target` Construct

Summary

Create a device data environment and execute the construct on the same 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:

device(*integer-expression***)**

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

if(*scalar-expression***)**

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:

device(*scalar-integer-expression***)**

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

if(*scalar-logical-expression***)**

The **end target** directive denotes the end of the **target** construct

Fortran

Binding

The binding task for a **target** construct is the encountering task. The target region binds to the enclosing parallel or task region.

Description

The **target** construct provides a superset of the functionality and restrictions provided by the **target data** directive. 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 encountering task waits for the device to complete the target region. When an **if** clause is present and the **if** clause expression evaluates to *false*, the target region is executed by the host device.

Restrictions

- If a **target**, **target update**, or **target data** construct appears within a target region then 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.
- A variable referenced in a **target** construct that is not declared in the construct is implicitly treated as if it had appeared in a **map** clause with a *map-type* of **tofrom**.
- 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.

C++

- 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++

Cross References

- **target data** construct, see Section 2.10.1 on page 86.
- *default-device-var*, see Section 2.3 on page 34.
- **map** clause, see Section 2.14.5 on page 187.

2.10.3 `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 where *motion-clause* *clause* is either *motion-clause* or one of the following:

to(**device(** *list* *integer-expression* **)**

from(**if(** *list* *scalar-expression* **)**

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

device(**to(** *integer-expression* *list* **)**

if(**from(** *scalar-expression* *list* **)**

C / C++

Fortran

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

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

where *motion-clause* *clause* is either *motion-clause* or one of the following:

to(**device(** *list* *scalar-integer-expression* **)**

from(**if(** *list* *scalar-logical-expression* **)**

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

device(**to(** *scalar-integer-expression* *list* **)**

if(**from(** *scalar-logical-expression* *list* **)**

Fortran

Binding

The binding task for a **target update** construct is the encountering task.The **target update** directive is a stand-alone directive.

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 , the behavior is unspecified 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 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.

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 variable that is part of another variable (such as a field of a structure) but is not an array element or an array section cannot appear as a list item in a clause of a **`target update`** construct.
- 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.
- At most one **`if`** clause can appear on the directive.

Cross References

- *default-device-var*, see Section 2.3 on page 34.
- **`target data`**, see Section 2.10.1 on page 86.
- Array sections, Section 2.4 on page 42

2.10.4 `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 is as follows:

```
#pragma omp declare target new-line
declarations-definition-seq declaration-definition-seq
#pragma omp end declare target new-line
```

C / C++

Fortran

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

For variables, functions and subroutines:

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

where *list* is a comma-separated list of named variables, procedure names and named common blocks. Common block names must appear between slashes.

For functions and subroutines:

```
!$omp declare target
```

Fortran

Description

C / C++

Variable and routine declarations that appear between the **`declare target`** and **`end declare target`** directives form an implicit list where each list item is the variable or function name.

C / C++

Fortran

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

Fortran

If a list item is a function (C, C++, Fortran) or subroutine (Fortran) then a device-specific version of the routine is created that can be called from a target region.

If a list item is a variable then the original variable is mapped to a corresponding variable in the initial device data environment for all devices. If the original variable is initialized, the corresponding variable in the device data environment is initialized with the same value.

Restrictions

- A threadprivate variable cannot appear in a **`declare target`** directive.
- A variable declared in a **`declare target`** directive must have a mappable type

C / C++

- A variable declared in a **`declare target`** directive must be at file or namespace scope.
- A function declared in a **`declare target`** directive must be at file, namespace, or class scope.
- All declarations and definitions for a function must have a **`declare target`** directive if one is specified for any of them. Otherwise, the result is unspecified

C / 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 a list can only appear in a specification part of a subroutine subprogram, function subprogram, program or module.
- Any **declare target** directive without a list can only 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** directive 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 **declare target** variable or a **declare target** common block 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

2.10.5 `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.

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.

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.

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.

The threads other than the master thread do not begin execution until the master thread encounters a **parallel** region.

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 or directives outside of the **teams** construct.
- **distribute**, **parallel**, **parallel sections**, **parallel workshare**, and the parallel loop and parallel loop SIMD constructs are the only OpenMP constructs that can be closely nested in the **teams** region.

Cross References

num SUBSCRIPTNB t eams SUBSCRIPTNB v ar , see Section 2.3.5 on page 40.
default, **shared**, **private**, **firstprivate**, and **reduction** clauses, see Section 2.14.3 on page 164. **omp_get_num_teams** routine, see Section 3.2.26 on page 228. **omp_get_team_num** routine, see Section 3.2.27 on page 230.

2.10.6 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***)**

collapse(*n***)**

dist_schedule(*kind[***,** *chunk_size]***)**

All associated for-loops *for-loops* must have the canonical form described in Section 2.6 on page 51.

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***)**

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-loop* *do-loops* .

All associated *do-loops* must be *do-constructs* 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.

Fortran

Binding

The binding thread set for a **distribute** region is the set of master threads created by a **teams** construct. A **distribute** region binds to the innermost enclosing **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.

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.

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. Note that the size of the chunks is unspecified in this case.

When no **dist_schedule** clause is specified, the schedule is implementation defined.

Restrictions

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

- The **distribute** construct inherits the restrictions of the loop construct.
- A **distribute** construct must be closely nested in a **teams** region.

Cross References

- loop construct, see Section 2.7.1 on page 54.
- **teams** construct, see Section 2.10.5 on page 95

2.10.7 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] ... ]
    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 .

Restrictions

- The restrictions for the **distribute** and **simd** constructs apply.
- A list item may appear in a **linear** or **firstprivate** clause but not both.
- A list item may appear in a **linear** or **lastprivate** clause but not both.

Cross References

- **simd** construct, see Section 2.8.1 on page 68.
- **distribute** construct, see Section 2.10.6 on page 98.
- Data attribute clauses, see Section 2.14.3 on page 164.

2.10.8 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] ... ]
    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. The resulting loops Each of these chunks will form a loop. Each resulting loop will then be distributed across the threads contained within the **teams** 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 the **distribute** and parallel loop constructs is as if it were applied to both constructs separately except the **collapse** clause, which is applied once .

Restrictions

- The restrictions for the **distribute** and parallel loop constructs apply.
- A list item may appear in a **linear** or **firstprivate** clause but not both.
- A list item may appear in a **linear** or **lastprivate** clause but not both.

Cross References

- **distribute** construct, see Section 2.10.6 on page 98.
- Parallel loop construct, see Section 2.11.1 on page 105.
- Data attribute clauses, see Section 2.14.3 on page 164.

2.10.9 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] ... ]
    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 the **distribute** and parallel loop SIMD constructs is as if it were applied to both constructs separately except the **collapse** clause, which is applied once.

Restrictions

- The restrictions for the **distribute** and parallel loop SIMD constructs apply.
- A list item may appear in a **linear** or **firstprivate** clause but not both.
- A list item may appear in a **linear** or **lastprivate** clause but not both.

Cross References

- **distribute** construct, see Section 2.10.6 on page 98.
- Parallel loop SIMD construct, see Section 2.11.4 on page 109.
- Data attribute clauses, see Section 2.14.3 on page 164.

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.

2.11.1 Parallel Loop Construct

Summary

The parallel loop construct is a shortcut for specifying a **`parallel`** construct containing 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-loop
```

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:

```
!$omp parallel do [clause[ [,] clause] ... ]
    do-loop   do-loops
[!$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-loop* *do-loops* . **nowait** may not be specified on an **end parallel do** directive.

Fortran

Description

C / C++

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

C / C++

Fortran

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

Fortran

Restrictions

- The restrictions for the **parallel** construct and the loop construct apply

Cross References

- **parallel** construct, see Section 2.5 on page 43.
- loop construct, see Section 2.7.1 on page 54.
- Data attribute clauses, see Section 2.14.3 on page 164.

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 43.
- **`sections`** construct, see Section 2.7.2 on page 61.
- Data attribute clauses, see Section 2.14.3 on page 164.

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 43.
- **workshare** construct, see Section 2.7.4 on page 65.
- Data attribute clauses, see Section 2.14.3 on page 164.

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++

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

where *clause* can be any of the clauses accepted by the **parallel**, **for** or **simd** directives, except the **nowait** clause, with identical meanings and restrictions.

C / C++

Fortran

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

where *clause* can be any of the clauses accepted by the **parallel**, **do** or **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-loop* *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 43.
- loop SIMD construct, see Section 2.8.3 on page 76.
- Data attribute clauses, see Section 2.14.3 on page 164.

2.11.5 `target teams` construct

Summary

The **`target teams`** construct is a shortcut for specifying a **`target`** construct containing a **`teams`** construct.

Syntax

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

C / C++

```
#pragma omp target teams [clause[ [ , ] clause] ... ]
    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

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

C / C++

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

C / C++

Fortran

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

Fortran

Restrictions

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

Cross References

- **target** construct, see Section 2.10.2 on page 87.
- **teams** construct, see Section 2.10.5 on page 95.
- Data attribute clauses, see Section 2.14.3 on page 164.

2.11.6 `teams distribute` Construct

Summary

The **teams distribute** construct is a shortcut for specifying a **teams** construct containing a **distribute** construct.

Syntax

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

C / C++

```
#pragma omp teams distribute [clause[ [ , ] clause] ... ]
    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

```
!$omp teams distribute [clause[ [ , ] clause] ... ]
    do-loops
[!$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. Some clauses are permitted on both constructs.

Restrictions

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

Cross References

- **teams** construct, see Section 2.10.5 on page 95.
- **distribute** construct, see Section 2.10.6 on page 98.
- Data attribute clauses, see Section 2.14.3 on page 164.

2.11.7 teams distribute simd Construct

Summary

The **teams distribute simd** construct is a shortcut for specifying a **teams** construct containing a **distribute simd** construct.

Syntax

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

C / C++

```
#pragma omp teams distribute simd [clause[ [,] clause] ... ]
    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

```
!$omp teams distribute simd [clause[ [,] clause] ... ]
    do-loops
[!$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** 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 simd** directive. Some clauses are permitted on both constructs.

Restrictions

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

Cross References

- **teams** construct, see Section 2.10.5 on page 95.
- **distribute simd** construct, see Section 2.10.7 on page 100.
- Data attribute clauses, see Section 2.14.3 on page 164.

2.11.8 target teams distribute construct

Summary

The **target teams distribute** construct is a shortcut for specifying a **target** construct containing a **teams distribute** construct.

Syntax

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

C / C++

```
#pragma omp target teams distribute [clause[ [,] clause] ... ]
    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

```
!$omp target teams distribute [clause[ [,] clause] ... ]
    do-loops
[!$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 2.10.1 on page 86.
- **teams distribute** construct, see Section 2.11.6 on page 113.
- Data attribute clauses, see Section 2.14.3 on page 164.

2.11.9 `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.

Syntax

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

C / C++

```
#pragma omp target teams distribute simd [clause[ [ , ] clause] ... ]
    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

```
!$omp target teams distribute simd [clause[ [ , ] clause] ... ]
    do-loops
[!$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 2.10.1 on page 86.
- **teams distribute simd** construct, see Section 2.11.7 on page 114.
- Data attribute clauses, see Section 2.14.3 on page 164.

2.11.10 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.

Syntax

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

C / C++

```
#pragma omp teams distribute parallel for [clause[ [,] clause] ... ]
    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

```
!$omp teams distribute parallel do [clause[ [,] clause] ... ]
    do-loops
[ !$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 2.10.5 on page 95.
- Distribute parallel loop construct, see Section 2.10.8 on page 102.
- Data attribute clauses, see Section 2.14.3 on page 164.

2.11.11 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.

Syntax

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

C / C++

```
#pragma omp target teams distribute parallel for [clause[ [,] clause] ... ]
    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

```
!$omp target teams distribute parallel do [clause[ [,] clause] ... ]
    do-loops
[!$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.

Cross References

- **`target`** construct, see Section 2.10.2 on page 87.
- Distribute parallel loop construct, see Section 2.11.10 on page 118.
- Data attribute clauses, see Section 2.14.3 on page 164.

2.11.12 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.

Syntax

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

C / C++

```
#pragma omp teams distribute parallel for simd [clause[ [,] clause] ... ]
    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

```
!$omp teams distribute parallel do simd [clause[ [,] clause] ... ]
    do-loops
[!$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 2.10.5 on page 95.
- Distribute parallel loop SIMD construct, see Section 2.10.9 on page 103.
- Data attribute clauses, see Section 2.14.3 on page 164.

2.11.13 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.

Syntax

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

C / C++

```
#pragma omp target teams distribute parallel for simd [clause[ [,] clause] ... ]
    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

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

Cross References

- **target** construct, see Section 2.10.2 on page 87.
- Teams distribute parallel loop SIMD construct, see Section 2.11.12 on page 120.
- Data attribute clauses, see Section 2.14.3 on page 164.

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(*scalar-expression*) final(*scalar-expression*) untied default(shared | none) mergeable private(*list*) firstprivate(*list*) shared(*list*) depend(*dependence-type* : *list*)

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(*scalar-logical-expression*) final(*scalar-logical-expression*) untied default(private | firstprivate | shared | none) mergeable private(*list*) firstprivate(*list*) shared(*list*) depend(*dependence-type* : *list*)

Binding

The binding thread set of the **task** region is the current team. A **task** region binds to the innermost enclosing **parallel** region.

Description

When a thread encounters a **`task`** construct, a 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. A **`task`** construct may be nested inside an outer task, but the **`task`** region of the inner task is not a part of the **`task`** region of the outer 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. Note that 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 a **`mergeable`** clause is present on a **`task`** construct, and the generated task is an undeferred task or an included task, the implementation may generate a merged task instead.

– When storage is shared by an explicit **`task`** region, it is the programmer's responsibility to ensure, by adding proper synchronization, that the storage does not reach the end of its lifetime before the explicit **`task`** region completes its execution.

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.

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.

Fortran

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

Summary

The **`depend`** clause enforces additional constraints on the scheduling of tasks. These constraints establish dependences only between sibling tasks. The clause consists of a *dependence-type* with one or more list items.

Syntax

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

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

Description

Task dependences are derived from the *dependence-type* of a **`depend`** clause and its list items, where *dependence-type* is one of the following:

The **`in`** *dependence-type*. The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an **`out`** or **`inout`** *dependence-type* list.

The **`out`** and **`inout`** *dependence-types*. The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an **`in`**, **`out`**, or **`inout`** *dependence-type* list.

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

– 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.

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 or disjoint storage.

List items used in **`depend`** clauses cannot be zero-length array sections.

A variable that is part of another variable (such as a field of a structure) but is not an array element or an array section cannot appear in a **`depend`** clause.

Cross References

Array sections, Section 2.4 on page 42.

Task scheduling constraints, Section 2.9.3 on page 84.

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
```

Fortran

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

```
!$omp taskyield
```

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 2.9.3 on page 84.

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

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.

– 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.

2.12 Master and Synchronization Constructs

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.

2.12.1 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.

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++

2.12.2 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)] new-line
    structured-block
```

C / C++

Fortran

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

```
!$omp critical [(name)]
    structured-block
!$omp end critical [(name)]
```

Fortran

Binding

The binding thread set for a **critical** region is all threads in the contention group. Region execution is restricted to a single thread at a time among all threads in the contention group, 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. A thread waits at the beginning of a **critical** region until no thread in the contention group is executing a **critical** region with the same name. The **critical** construct enforces exclusive access 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.

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

Restrictions

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

2.12.3 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.

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.

2.12.4 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 newline
```

C / C++

Fortran

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

```
!$omp taskwait
```

Fortran

Binding

The **taskwait** region binds to the current task region. The binding thread set of the **taskwait** region is the current team.

Description

The **taskwait** region includes an implicit task scheduling point in the current task region. The current task region is suspended at the task scheduling point until all child tasks that it generated before the **taskwait** region complete execution.

2.12.5 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 new-line
    structured-block
```

C / C++

Fortran

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

```
!$omp taskgroup
    structured-block
!$omp end taskgroup
```

Fortran

Binding

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

Description

When a thread encounters a **taskgroup** construct, it starts executing the 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 child tasks that it generated in the **taskgroup** region and all of their descendent tasks complete execution.

Cross References

- Task scheduling, see Section 2.9.3 on page 84.

2.12.6 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

C / C++

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

```
#pragma omp atomic [read | write | update |
 capture] [seq_cst] new-line
    expression-stmt
```

or

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

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

- If clause is **read**:

```
v = x;
```

- If clause is **write**:

```
x = expr;
```

- If clause is **update** or not present:

```
x++;
x--;
++x;
--x;
x binop= expr;
x = x binop expr;
x = expr binop x;
```

- If 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:

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

or

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

or

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

or

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

or

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

or

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

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

x **=** *expr*

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

v **=** *x*

and where *update-statement* has one of the following forms (if 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 be an allocatable variable. 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 mathematically 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

- In all **atomic** construct forms, the **seq_cst** clause and the clause that denotes the type of the atomic construct can appear in any order. In addition, an optional comma may be used to separate the clauses

Binding

The binding thread set for an 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 contention group 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 19 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. 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.

Restrictions

C / C++

The following restriction applies to the **atomic** construct:

- All atomic accesses to the storage locations designated by x throughout the program are required to have a compatible type.

C / C++

Fortran

The following restriction applies to the **atomic** construct:

- All atomic accesses to the storage locations designated by x throughout the program are required to have the same type and type parameters.

Fortran

Cross References

- **`critical`** construct, see Section 2.12.2 on page 130.
- **`barrier`** construct, see Section 2.12.3 on page 132.
- **`flush`** construct, see Section 2.12.7 on page 142.
- **`ordered`** construct, see Section 2.12.8 on page 146.
- **`reduction`** clause, see Section 2.14.3.6 on page 176.
- lock routines, see Section 3.3 on page 232.

2.12.7 `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 16 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 currently 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. Notice that 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**, and **critical**, **ordered**, **target** and **target data** regions.
- 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.

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 2.12.6 on page 135.

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.

2.12.8 `ordered` Construct

Summary

The **ordered** construct specifies a structured block in a loop region that will be executed in the order of the loop iterations. This sequentializes and orders the code within an **ordered** region 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**

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**

Fortran

Binding

The binding thread set for an **ordered** region is the current team. An **ordered** region binds to the innermost enclosing loop region. **ordered** regions that bind to different loop regions execute independently of each other.

Description

The threads 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. 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** region, it waits at the beginning of that **ordered** region until execution of all the **ordered** regions belonging to all previous iterations have completed. 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.

Restrictions

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

- The loop region to which an **ordered** region with a **threads** clause binds must have an **ordered** clause specified on the corresponding loop (or parallel loop) construct.
- During execution of an iteration of a loop or a loop nest within a loop region, a thread must not execute more than one **ordered** region with the **threads** clause that binds to the same loop region.

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.
- An **ordered** region with the **simd** clause must be closely nested to a **simd** or **declare simd** construct.

C++

Fortran

- An **ordered** region with the **simd** clause must be closely nested to a **simd** construct or to a function or subroutine that is the target of a **declare simd** construct.

Fortran

Cross References

- loop construct, see Section 2.7.1 on page 54.
- parallel loop construct, see Section 2.11.1 on page 105.

2.13 Cancellation Constructs

2.13.1 `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 (*scalar-expression***)**

C / C++

Fortran

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

```
!$omp cancel construct-type-clause[ [ , ] if-clause] new-line
```

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

parallel

sections

do

taskgroup

and *if-clause* is

if (*scalar-logical-expression***)**

Fortran

Binding

The binding thread set of the **cancel** region is the current team. The **cancel** region binds to the innermost enclosing construct of the type corresponding to the *type-clause* specified in the directive (that is, the innermost **parallel**, **sections**, **do** loop , or **taskgroup** construct).

Description

The **cancel** construct activates cancellation of the binding construct only if *cancel-var* is **true**, in which case the construct causes the encountering task to continue execution at the end of the canceled construct. If *cancel-var* is **false**, the **cancel** construct is ignored.

Threads check for active cancellation only at cancellation points. Cancellation points are implied at the following locations:

- implicit barriers;
- **barrier** regions;
- **cancel** regions;
- **cancellation point** regions;

When a thread reaches one of the above cancellation points and if *cancel-var* is *true*, the thread immediately checks for active cancellation (that is, if cancellation has been activated by a **cancel** construct). If cancellation is active, the encountering thread continues execution at the end of the canceled construct.

Note – If one thread activates cancellation and another thread encounters a cancellation point, the absolute 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 innermost enclosing **taskgroup** 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 **taskgroup** **task** region, which implies its 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**, **for**, or **do** 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**, **for**, or **do** 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

Note – The user programmer is responsible for releasing locks and similar 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 synchronization to avoid deadlocks that might arise from cancellation of OpenMP regions that contain OpenMP synchronization constructs.

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.

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. 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.
- If *construct-type-clause* is **taskgroup** and the **cancel** construct is not nested inside a **taskgroup** region, then the behavior is unspecified.
- A worksharing construct that is canceled must not have a **nowait** clause.
- A loop construct that is canceled must not have an **ordered** clause.
- A construct that may be subject to cancellation 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*, see Section 2.3.1 on page 34.
- **cancellation point** construct, see Section 2.13.2 on page 152.
- **omp_get_cancellation** routine, see Section 3.2.9 on page 209.

2.13.2 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

A **cancellation point** region binds to the current task 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 a synchronization between threads or tasks.

When an implicit or explicit task reaches a user-defined cancellation point and if *cancel-var* is **true** the task immediately checks whether cancellation of the region specified in the clause has been activated. If so, the encountering task continues execution at the end of the canceled construct.

Restrictions

- A **cancellation point** construct for which *construct-type-clause* is **taskgroup** must be closely nested inside a **task** construct. A **cancellation point** construct for which *construct-type-clause* is not **taskgroup** must be closely nested inside an OpenMP construct that matches the type specified in *construct-type-clause*.
- An OpenMP program with orphaned **cancellation point** constructs is non-conforming.

Cross References

- *cancel-var*, see Section 2.3.1 on page 34.
- **cancel** construct, see Section 2.13.1 on page 148.
- **omp_get_cancellation** routine, see Section 3.2.9 on page 209.

2.14 Data Environment

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

- Section 2.14.1 on page 154 describes how the data-sharing attributes of variables referenced in **parallel**, **task**, **simd**, and worksharing regions are determined.
- The **threadprivate** directive, which is provided to create threadprivate memory, is described in Section 2.14.2 on page 159.
- Clauses that may be specified on directives to control the data-sharing attributes of variables referenced in **parallel**, **task**, **simd** or worksharing constructs are described in Section 2.14.3 on page 164
- 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 2.14.4 on page 183.
- Clauses that may be specified on directives to map variables to devices are described in Section 2.14.5 on page 187.

2.14.1 Data-sharing Attribute Rules

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

- Section 2.14.1.1 on page 154 describes the data-sharing attribute rules for variables referenced in a construct.
- Section 2.14.1.2 on page 158 describes the data-sharing attribute rules for variables referenced in a region, but outside any construct.

2.14.1.1 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`** or **`parallel for`** 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 *constant-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.

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`** or **`parallel do`** 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 *constant-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`** construct is private in the innermost such construct that encloses the loop.
- Implied-do indices and **`forall`** indices are private.
- Cray pointees inherit the have the same the data-sharing attribute of 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** or **parallel for** 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 *constant-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** or **parallel do** 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 *constant-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** 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 2.14.3 on page 164.

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** or **task** construct, the data-sharing attributes of these variables are determined by the **default** clause, if present (see Section 2.14.3.1 on page 165).
- In a **parallel** construct, if no **default** clause is present, these variables are shared.
- For constructs other than **task**, if no **default** clause is present, these variables inherit their data-sharing attributes from the reference the variables with the same names that exist in the enclosing context.
- In a **task** construct, if no **default** clause is present, a variable that in the enclosing context is determined to be shared by all implicit tasks bound to the current team is shared.

Fortran

- In an orphaned **task** construct, if no **default** clause is present, dummy arguments are firstprivate.

Fortran

- In a **task** construct, if no **default** clause is present, a variable whose 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** construct are described in Section 2.14.3.4 on page 171.

2.14.1.2 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.
- Variables with **`const`** -qualified type having no mutable member, and that are declared in called routines, 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.
- Formal In C++, formal arguments of called routines in the region that are passed by reference inherit the have the same data-sharing attributes of as the associated actual argument 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 declared in modules 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 are passed by reference inherit the have the same data-sharing attributes of as the associated actual argument arguments .
- Cray pointees inherit the have the same data-sharing attribute of 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

2.14.2 `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 13 and Section 2.9 on page 78.

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 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 is 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 not currently allocated.
- 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 34.
- number of threads used to execute a **parallel** region, see Section 2.5.1 on page 47.
- **copyin** clause, see Section 2.14.4.1 on page 184.

2.14.3 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 25). 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.

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

A named common block may be specified in a list by enclosing the name in slashes. 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 clause must be declared to be a common block in the same scoping unit in which the data-sharing attribute clause appears.

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**, or **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

2.14.3.1 default clause

Summary

The **default** clause explicitly determines the data-sharing attributes of variables that are referenced in a **parallel**, **task** or **teams** construct and would otherwise be implicitly determined (see Section 2.14.1.1 on page 154).

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`**, or **`teams`** directive.

2.14.3.2 `shared` clause

Summary

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

It is the programmer's responsibility to 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**, **task** or **teams** construct if it is associated with a target or a subobject of a target that is in a **private**, **firstprivate**, **lastprivate**, or **reduction** clause inside the construct.

Under certain conditions, passing a shared variable to a non-intrinsic procedure may result in the value of the shared variable being copied into temporary storage before the procedure reference, and back out of the temporary storage into the actual argument storage after the procedure reference. It is implementation defined when this situation occurs

Note – Use of intervening temporary storage may occur when the following three conditions hold regarding an actual argument in a reference to a non-intrinsic procedure:

a The actual argument is one of the following:

- A shared variable.
- A subobject of a shared variable.
- An object associated with a shared variable.
- An object associated with a subobject of a shared variable.

b The actual argument is also one of the following:

- An array section.
- An array section with a vector subscript.
- An assumed-shape array.
- A pointer array.

c The associated dummy argument for this actual argument is an explicit-shape array or an assumed-size array.

These conditions effectively result in references to, and definitions of, the temporary storage during the procedure reference. 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 / C++

- A variable that is part of another variable (as an array or structure element) cannot appear in a **shared** clauseunless it is a static data member of a C++ class shared clause .

Fortran

A variable that is part of another variable (as an array or structure element) cannot appear in a **shared** clause.

2.14.3.3 `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. 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`, `task`, or worksharing, or `simd` construct.

List items that appear in a `private` or `firstprivate` clause in a `task` construct may also appear in a `private` clause in an enclosed `parallel` or `task` 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` or `task` construct.

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.

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/or 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++

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: once for each implicit task in the **parallel** construct; once for each task generated by a **task** construct; and once for each SIMD lane used by a **simd** construct. The initial value of the new list item is undefined. Within a **parallel**, **worksharing**, **task**, **teams**, or **simd** region, 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 "not currently allocated", the new list item or the subobject of the new list item will have an initial allocation status of "not currently allocated";
- if the allocation status is "currently allocated", the new list item or the subobject of the new list item will have an initial allocation status of "currently 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 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 **parallel**, **task**, **simd**, or **teams** 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:

- A variable that is part of another variable (as an array or structure element) cannot appear in a **`private`** clause.

C / C++

- A variable of class type (or array thereof) that appears in a **`private`** clause requires an accessible, unambiguous default constructor for the class type.
- 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 a reference type. be a reference to an incomplete type.
- If a list item is 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 **`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

2.14.3.4 `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 2.14.3.3 on page 168, 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**, 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 allocation status of not currently allocated, 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 variable that is part of another variable (as an array or structure element) cannot appear in a `firstprivate` clause.
- 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 or `task` construct if any of the worksharing or `task` regions arising from the worksharing or `task` 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 in 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 type or a reference type. C/C++ type or be a reference to an incomplete type.
- If a list item is 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, and in expressions for statement function definitions, may not appear in a `firstprivate` clause.

Fortran

2.14.3.5 `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(`*list***`)`**

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 2.14.3.3 on page 168. In addition, when a **`lastprivate`** clause 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.

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

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 to which **nowait** is applied, accesses to the original list item may create a data race. To avoid this, synchronization must be inserted to ensure that the sequentially last iteration or lexically last section construct has stored and flushed that list item.

If a list item the **lastprivate** clause is used on a **distribute simd** , distribute parallel loop, or distribute parallel loop SIMD, accesses to the original list item may create a data race. To avoid this, synchronization must be inserted to ensure that the sequentially last iteration has stored and flushed that list item.

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 variable that is part of another variable (as an array or structure element) cannot appear in a **lastprivate** clause.
- 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.

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 type or a reference type. C/C++ type or be a reference to an incomplete type.
- If a list item is 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.
- An If the original list item with has the **ALLOCATABLE** attribute, the corresponding list item in the sequentially last iteration or lexically last section must have an allocation status of allocated upon exit from that iteration or section.
- Variables that appear in namelist statements, in variable format expressions, and in expressions for statement function definitions, may not appear in a **lastprivate** clause.

Fortran

2.14.3.6 `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

C / C++

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

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

where:

C

reduction-identifier is either an *identifier* or one of the following operators: `+`, `-`, `*`, `&`, `|`, `^`, `&&` and `||`

C

C++

reduction-identifier is either an *id-expression* or one of the following operators: `+`, `-`, `*`, `&`, `|`, `^`, `&&` and `||`

C++

The following table 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.

Identifier	Initializer	Combiner
`+`	`omp_priv = 0`	`omp_out += omp_in`
`*`	`omp_priv = 1`	`omp_out *= omp_in`
`-`	`omp_priv = 0`	`omp_out += omp_in`
`&`	`omp_priv =  0`	`omp_out &= omp_in`
`|`	`omp_priv = 0`	`omp_out |= omp_in`
`^`	`omp_priv = 0`	`omp_out ^= omp_in`
`&&`	`omp_priv = 1`	`omp_out = omp_in && omp_out`
`||`	`omp_priv = 0`	`omp_out = omp_in || omp_out`
`max`	`omp_priv =` *Least representable number in the reduction list item type*	`omp_out = omp_in > omp_out ? omp_in : omp_out`

table continued on next page

table continued from previous page

Identifier	Initializer	Combiner
min	**omp_priv = ** *Largest representable number in the reduction list item type*	**omp_out = omp_in < omp_out ? omp_in : omp_out**

where **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.

C / C++

Fortran

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

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

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**.

The following table 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.

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**
.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**

table continued on next page

table continued from previous page

Identifier	Initializer	Combiner
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

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

The reduction clause can be used to perform some forms of recurrence calculations (involving mathematically associative and commutative operators) in parallel.

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 **teams** construct, a private copy of each list item is created, one for each team in the league as if the **private** clause had been used. The private copy is then initialized as specified above. 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*.

The *reduction-identifier* specified in the **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.

C++

If the type of a The list items that appear in the **reduction** clause may include array sections.

If the list item is a reference to a type *T* then the type will be considered to be *T* for all purposes of this clause an array or an array section it will be treated as if a **reduction** clause would be applied to each separate element of the array section. The elements of each private array section will be allocated contiguously .

If the type is a derived class, then any *reduction-identifier* that matches its base classes are 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 **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.

The location in the OpenMP program at which the values are combined and the order in which the 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.

Restrictions

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

- 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.

- Any number of **reduction** clauses can be specified on the directive, but a list item can appear only once in the **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, its lower-bound must be zero.
- 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**.
- Arrays may not appear in a **reduction** clause.

A list item that appears in a **reduction** clause must not be **const**-qualified.

- If a list item is a reference type then it must bind to the same object for all threads of the team.
- 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 reduction operator or intrinsic *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.
- A pointer must be associated upon entry and exit to the region.
- A pointer must not have its association status changed within the region.
- An original list item with the **POINTER** attribute must be associated at entry to the construct containing the **reduction** clause. Additionally, the list item must not be deallocated, allocated, or pointer assigned within the region.

- An original list item with the **ALLOCATABLE** attribute must be in the allocated state at entry to the construct containing the **reduction** clause. Additionally, the list item must not be deallocated and/or 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

2.14.3.7 `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

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

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

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 2.14.3.3 on page 168 except as noted. In addition, 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*. If *linear-step* is not specified it is assumed to be 1. The value corresponding to the sequentially last iteration of the associated loops is assigned to the original list item.

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 / C++

- A *list-item* that appears in a **linear** clause must be of integral or pointer type, or must be a reference to an integral or pointer type .

C / C++

Fortran

- A *list-item* that appears in a **linear** clause must be of type **integer**.
- 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.

Fortran

2.14.4 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 25). 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

2.14.4.1 `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 not currently allocated, 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

2.14.4.2 `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 63), 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 whose thread 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

2.14.5 map Clause

Summary

The **map** clause maps a variable from the current task's data environment to the device data environment associated with the construct.

Syntax

The syntax of the map clause is as follows:

```
map([map-type : ] list)
```

Description

The list items that appear in a **map** clause may include array sections.

For list items that appear in a **map** clause, corresponding new list items are created in the device data environment associated with the construct.

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 a corresponding list item of the original list item is in the enclosing device data environment, the new device data environment uses the corresponding list item from the enclosing device data environment. No additional storage is allocated in the new device data environment and neither initialization nor assignment is performed, regardless of the *map-type* that is specified.

If a corresponding list item is not in the enclosing device data environment, a new list item with language-specific attributes is derived from the original list item and created in the new device data environment. This new list item becomes the corresponding list item to the original list item in the new device data environment. Initialization and assignment are performed if specified by the *map-type*.

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 storage and thus lifetime of this list item lasts until the block in which it is created exits. The size and alignment of the new list item are determined by the type of the variable. This allocation occurs if the region references the list item in any statement.

If the type of the variable appearing in an array section is pointer, reference to array, or reference to pointer then the variable is implicitly treated as if it had appeared in a **map** clause with a *map-type* of **alloc**. The corresponding variable is assigned the address of the storage location of the corresponding array section in the new device data environment. If the variable appears in a **to** or **from** clause in a **target update** region enclosed by the new device data environment but not as part of the specification of an array section, the behavior is unspecified.

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.

The **alloc** *map-type* declares that on entry to the region each new corresponding list item has an undefined initial value.

The **to** *map-type* declares that on entry to the region each new corresponding list item is initialized with the original list item's value.

The **from** *map-type* declares that on exit from the region the corresponding list item's value is assigned to each original list item.

The **tofrom** *map-type* declares that on entry to the region each new corresponding list item is initialized with the original list item's value and that on exit from the region the corresponding list item's value is assigned to each original list item.

If a *map-type* is not specified, the *map-type* defaults to **tofrom**.

Restrictions

- 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 has corresponding storage in the enclosing device data environment, all of the original storage must have corresponding storage in the enclosing device data environment.
- A variable that is part of another variable (such as a field of a structure) but is not an array element or an array section cannot appear in a **map** clause.
- 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 / C++

- Initialization and assignment are through bitwise copy.
- A variable for which the type is pointer, reference to array, or reference to pointer and an array section derived from that variable must not appear as list items of **`map`** clauses of the same construct.
- A variable for which the type is pointer, reference to array, or reference to pointer must not appear as a list item if the enclosing device data environment already contains an array section derived from that variable.
- An array section derived from a variable for which the type is pointer, reference to array, or reference to pointer must not appear as a list item if the enclosing device C/C++ data environment already contains that variable.

C / C++

Fortran

- The value of the new list item becomes that of the original list item in the map Fortran initialization and assignment.

Fortran

2.15 `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 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 a base language identifier or one of the following operators: **+**, **−**, *****, **&**, **|**, **^**, **&&** and **||**
- *typename-list* is 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 *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 *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** 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 or 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, the or the same as a generic name that is the same as one of the allowed intrinsic procedures, the interface for that operator or the generic name must be defined in the same subprogram specification of the same module , or must be accessible by host or use association.

- If the **declare reduction** directive appears in a module, any user-defined operators Any subroutine, or function used in the combiner must be defined in the same subprogram **initializer** clause or *combiner* expression must be an intrinsic function , or must be accessible by host or use association. The user-defined operators must also be accessible by host or use association in the subprogram in which the corresponding **reduction** clause appears. have an accessible interface.

- Any subroutine or function user-defined operator, or extended operator used in the **initializer** clause or *combiner* expression must be an intrinsic function, or must have an explicit interface in the same subprogram or have an accessible interface.

- If any subroutine, function, user-defined operator or extended operator used in the **initializer** clause or *combiner* expression, it must be accessible by host or use association 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** directives with character type Fortran character type of the same kind parameter and the same *reduction-identifier* are 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 2.14.3.6 on page 176.

2.16 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, explicit **task**, **critical**, **ordered**, **atomic**, or **master** region.
- A **barrier** region may not be closely nested inside a worksharing, explicit **task**, **critical**, **ordered**, **atomic**, or **master** region.
- A **master** region may not be closely nested inside a worksharing, **atomic**, or explicit **task** region.
- An **ordered** region may not be closely nested inside a **critical**, **atomic**, or explicit **task** region.
- An **ordered** region must be closely nested inside a loop region (or parallel loop region) with an **ordered** clause.
- A **critical** region may not be nested (closely or otherwise) inside a **critical** region with the same name. Note that this restriction is not sufficient to prevent deadlock.
- OpenMP constructs may not be nested inside an **atomic** region.
- OpenMP constructs may not be nested inside a **simd** region.
- If a **target**, **target update**, or **target data** construct appears within a **target** region then the behavior is unspecified.
- If specified, a **teams** construct must be contained within a **target** construct. That **target** construct must contain no statements or directives outside of the **teams** construct.
- **distribute**, **parallel**, **parallel sections**, **parallel workshare**, and the parallel loop and parallel loop SIMD constructs are the only OpenMP constructs that can be closely nested in the **teams** region.
- A **distribute** construct must be closely nested in a **teams** region.
- If *construct-type-clause* is **taskgroup**, the **cancel** construct must be closely nested inside a **task** construct and the **cancel** construct must be nested inside a **taskgroup** region. 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 nested inside a **task** construct. A **cancellation point** construct for which *construct-type-clause* is not **taskgroup** 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 is divided into the following sections:

- Runtime library definitions (Section 3.1 on page 199).
- Execution environment routines that can be used to control and to query the parallel execution environment (Section 3.2 on page 200).
- Lock routines that can be used to synchronize access to data (Section 3.3 on page 232).
- Portable timer routines (Section 3.4 on page 238).

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_sched_t**.
- The type **omp_proc_bind_t**.

See Section Section C.1 on page 289 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_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 31).

See Section C.1 on page 291 and Section C.3 on page 294 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 C.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.

See Section 2.5.1 on page 47 for the rules governing the number of threads used to execute a **parallel** region.

Cross References

- *nthreads-var* ICV, see Section 2.3 on page 34.
- **OMP_NUM_THREADS** environment variable, see Section 4.2 on page 245.
- **omp_get_max_threads** routine, see Section 3.2.3 on page 202.
- **parallel** construct, see Section 2.5 on page 43.
- **num_threads** clause, see Section 2.5 on page 43.

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.

See Section 2.5.1 on page 47 for the rules governing the number of threads used to execute a **parallel** region.

Cross References

- **parallel** construct, see Section 2.5 on page 43.
- **omp_set_num_threads** routine, see Section 3.2.1 on page 200.
- **OMP_NUM_THREADS** environment variable, see Section 4.2 on page 245.

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.

See Section 2.5.1 on page 47 for the rules governing the number of threads used to execute a **parallel** region.

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 34.
- **parallel** construct, see Section 2.5 on page 43.
- **num_threads** clause, see Section 2.5 on page 43.
- **omp_set_num_threads** routine, see Section 3.2.1 on page 200.
- **OMP_NUM_THREADS** environment variable, see Section 4.2 on page 245.

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 201.

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. Note that 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.

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 34.
- **omp_get_active_level** routine, see Section 3.2.20 on page 222.

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*.

See Section 2.5.1 on page 47 for the rules governing the number of threads used to execute a **parallel** region.

Cross References

- *dyn-var* ICV, see Section 2.3 on page 34.
- **omp_get_num_threads** routine, see Section 3.2.2 on page 201.
- **omp_get_dynamic** routine, see Section 3.2.8 on page 208.
- **OMP_DYNAMIC** environment variable, see Section 4.3 on page 246.

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*.

See Section 2.5.1 on page 47 for the rules governing the number of threads used to execute a **parallel** region.

Cross References

- *dyn-var* ICV, see Section 2.3 on page 34.
- **omp_set_dynamic** routine, see Section 3.2.7 on page 206.
- **OMP_DYNAMIC** environment variable, see Section 4.3 on page 246.

3.2.9 omp_get_cancellation

Summary

The **omp_get_cancellation** routine returns the value of the *cancel-var* ICV, which controls the behavior of the **cancel** construct and cancellation points.

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 activated. It returns *false* otherwise.

Cross References

- *cancel-var* ICV, see Section 2.3.1 on page 34.
- **OMP_CANCELLATION** environment variable, see Section 4.11 on page 252

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*.

See Section 2.5.1 on page 47 for the rules governing the number of threads used to execute a **parallel** region.

Cross References

- *nest-var* ICV, see Section 2.3 on page 34.
- **omp_set_max_active_levels** routine, see Section 3.2.15 on page 216.
- **omp_get_max_active_levels** routine, see Section 3.2.16 on page 217.
- **omp_get_nested** routine, see Section 3.2.11 on page 211.
- **OMP_NESTED** environment variable, see Section 4.6 on page 249.

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.

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*.

See Section 2.5.1 on page 47 for the rules governing the number of threads used to execute a **`parallel`** region.

Cross References

- *nest-var* ICV, see Section 2.3 on page 34.
- **`omp_set_nested`** routine, see Section 3.2.10 on page 209.
- **`OMP_NESTED`** environment variable, see Section 4.6 on page 249.

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 modifier     kind
, int     modifier   );
```

C / C++

Fortran

```
subroutine omp_set_schedule( kind, modifier     kind ,     modifier   )
integer (kind=omp_sched_kind)   kind
integer modifier
   kind
integer    modifier
```

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** *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 34.
- **omp_get_schedule** routine, see Section 3.2.13 on page 214.
- **OMP_SCHEDULE** environment variable, see Section 4.1 on page 244.
- Determining the schedule of a worksharing loop, see Section 2.7.1.1 on page 60.

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     kind
 , int * modifier     modifier  );
```

C / C++

Fortran

```
subroutine omp_get_schedule( kind, modifier     kind ,    modifier   )
integer (kind=omp_sched_kind)  kind
integer modifier
  kind
integer    modifier
```

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`** *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 212, 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 212.

Cross References

- *run-sched-var* ICV, see Section 2.3 on page 34.
- **omp_set_schedule** routine, see Section 3.2.12 on page 212.
- **OMP_SCHEDULE** environment variable, see Section 4.1 on page 244.
- Determining the schedule of a worksharing loop, see Section 2.7.1.1 on page 60.

3.2.14 omp_get_thread_limit

Summary

The **omp_get_thread_limit** routine returns the maximum number of OpenMP threads available on the device 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 maximum number of OpenMP threads available on the device as stored in the ICV value of the *thread-limit-var* ICV .

Cross References

- *thread-limit-var* ICV, see Section 2.3 on page 34.
- **OMP_THREAD_LIMIT** environment variable, see Section 4.10 on page 251.

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  %DIFDELCMD <
%%%
SUBSCRIPTNB  %DIFDELCMD < %%%
l  %DIFDELCMD < %%%
evels     max
SUBSCRIPTNB l evels  );
```

C / C++

Fortran

```
subroutine omp_set_max_active_levels( max  %DIFDELCMD <
%%%
SUBSCRIPTNB  %DIFDELCMD < %%%
l  %DIFDELCMD < %%%
evels)
integer max  %DIFDELCMD <
%%%
SUBSCRIPTNB  %DIFDELCMD < %%%
l  %DIFDELCMD < %%%
evels
  max
SUBSCRIPTNB l evels )
integer   max
SUBSCRIPTNB l evels
```

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 34.
- **omp_get_max_active_levels** routine, see Section 3.2.16 on page 217.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 4.9 on page 251.

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 34.
- **omp_set_max_active_levels** routine, see Section 3.2.15 on page 216.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 4.9 on page 251.

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 34.
- **omp_get_active_level** routine, see Section 3.2.20 on page 222.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 4.9 on page 251.

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  level );
```

C / C++

Fortran

```
integer function omp_get_ancestor_thread_num( level)
integer level
  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_level** routine, see Section 3.2.17 on page 218.
- **omp_get_thread_num** routine, see Section 3.2.4 on page 204.
- **omp_get_team_size** routine, see Section 3.2.19 on page 221.

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     level   );
```

C / C++

Fortran

```
integer function omp_get_team_size( level)
integer level
  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 201.
- **omp_get_level** routine, see Section 3.2.17 on page 218.
- **omp_get_ancestor_thread_num** routine, see Section 3.2.18 on page 220.

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 34.
- **omp_get_level** routine, see Section 3.2.17 on page 218.

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*.

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 49 for the rules governing the thread affinity policy.

Cross References

- *bind-var* ICV, see Section 2.3 on page 34.
- **OMP_PROC_BIND** environment variable, see Section 4.4 on page 246.
- Controlling OpenMP thread affinity, see Section 2.5.2 on page 49.

3.2.23 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 34.
- **omp_get_default_device**, see Section 3.2.24 on page 227.
- **OMP_DEFAULT_DEVICE** environment variable, see Section 4.13 on page 253

3.2.24 omp_get_default_device

Summary

The **omp_get_default_device** routine returns the default target device.

Format

C / C++

```c
int omp_get_default_device(void);
```

C / C++

Fortran

```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 34.
- **omp_set_default_device**, see Section 3.2.23 on page 226.
- **OMP_DEFAULT_DEVICE** environment variable, see Section 4.13 on page 253.

3.2.25 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.

3.2.26 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 2.10.5 on page 95.

3.2.27 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 2.10.5 on page 95.
- **omp_get_num_teams** routine, see Section 3.2.26 on page 228.

3.2.28 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 2.10.2 on page 87

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.

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 in such a way 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`** is a data type capable of representing 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_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** is a data type capable of representing a nestable lock. For the following routines, a nested 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 nested 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_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 provide the only means of initializing an OpenMP lock.

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.

3.3.2 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.

3.3.3 omp_set_lock and omp_set_nest_lock

Summary

These routines provide a means of setting an OpenMP lock. The calling task region is suspended until the lock is set.

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 causes suspension of the task executing the routine until the specified lock is available and then sets the lock.

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.

3.3.4 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 suspended because the lock was unavailable, the effect is that one task is chosen and given ownership of the lock.

3.3.5 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.

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 the 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**.

CHAPTER 4

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 34). 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 the OpenMP program 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.

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
```

4.1 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 54 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 212.

Example:

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

Cross References

- *run-sched-var* ICV, see Section 2.3 on page 34.
- Loop construct, see Section 2.7.1 on page 54.
- Parallel loop construct, see Section 2.11.1 on page 105.
- **omp_set_schedule** routine, see Section 3.2.12 on page 212.
- **omp_get_schedule** routine, see Section 3.2.13 on page 214.

4.2 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 34 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 47 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 34.
- **num_threads** clause, Section 2.5 on page 43.
- **omp_set_num_threads** routine, see Section 3.2.1 on page 200.
- **omp_get_num_threads** routine, see Section 3.2.2 on page 201.
- **omp_get_max_threads** routine, see Section 3.2.3 on page 202.
- **omp_get_team_size** routine, see Section 3.2.19 on page 221.

4.3 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 34.
- **omp_set_dynamic** routine, see Section 3.2.7 on page 206.
- **omp_get_dynamic** routine, see Section 3.2.8 on page 208.

4.4 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.

The behavior of the program is implementation defined if any of the values 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.

Example:

```
setenv OMP_PROC_BIND false
setenv OMP_PROC_BIND "spread, spread, close"
```

Cross References

- *bind-var* ICV, see Section 2.3 on page 34.
- **proc_bind** clause, see Section 2.5.2 on page 49.
- **omp_get_proc_bind** routine, see Section 3.2.22 on page 224.

4.5 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 4-1 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 4-1 List of 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.

Example:

```
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 34.
- Controlling OpenMP thread affinity, Section 2.5.2 on page 49.

4.6 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 34.
- **omp_set_nested** routine, see Section 3.2.10 on page 209.
- **omp_get_team_size** routine, see Section 3.2.19 on page 221.

4.7 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 34.

4.8 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 34.

4.9 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 34.
- **omp_set_max_active_levels** routine, see Section 3.2.15 on page 216.
- **omp_get_max_active_levels** routine, see Section 3.2.16 on page 217.

4.10 OMP_THREAD_LIMIT

The **OMP_THREAD_LIMIT** environment variable sets the maximum number of OpenMP threads to use for the whole OpenMP program 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 34.
- **omp_get_thread_limit** routine, see Section 3.2.14 on page 215.

4.11 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 34.
- **cancel** construct, see Section 2.13.1 on page 148.
- **cancellation point** construct, see Section 2.13.2 on page 152.
- **omp_get_cancellation** routine, see Section 3.2.9 on page 209.

4.12 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 4, 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 4. 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**, undefined, or any other value than **TRUE** or **VERBOSE**.

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='201307'
  [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
```

4.13 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 34.
- device constructs, Section 2.10 on page 86.

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.

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 modifier)
{
}

void omp_get_schedule(omp_sched_t *kind, int *modifier)
{
    *kind = omp_sched_static;
    *modifier = 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;
}

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;
    }

    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_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_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.;
    }
```

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, modifier)
  include 'omp_lib_kinds.h'
  integer (kind=omp_sched_kind) kind
  integer modifier
  return
end subroutine

subroutine omp_get_schedule(kind, modifier)
  include 'omp_lib_kinds.h'
  integer (kind=omp_sched_kind) kind
  integer modifier
  kind = omp_sched_static
  modifier = 0
  return
end subroutine

integer function omp_get_thread_limit()
  omp_get_thread_limit = 1
  return
end function

subroutine omp_set_max_active_levels( level )
  integer 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 omp_get_proc_bind

   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

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_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_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
```

APPENDIX B

OpenMP C and C++ Grammar

B.1 Notation

The grammar rules consist of the name for a non-terminal, followed by a colon, followed by replacement alternatives on separate lines.

The syntactic expression *term*$_{opt}$ indicates that the term is optional within the replacement.

The syntactic expression *term*$_{optseq}$ is equivalent to *term-seq*$_{opt}$ with the following additional rules:

term-seq :

term

term-seq term

term-seq , term

B.2 Rules

The notation is described in Section 6.1 of the C standard. This grammar appendix shows the extensions to the base language grammar for the OpenMP C and C++ directives.

C++

statement-seq:

  statement

  openmp-directive

  statement-seq statement

  statement-seq openmp-directive

C++

C90

statement-list:

  statement

  openmp-directive

  statement-list statement

  statement-list openmp-directive

C90

C99

block-item:

  declaration

  statement

  openmp-directive

C99

statement:

  /* standard statements */

  openmp-construct

  declaration-definition:

    /* Any C or C++ declaration or definition statement */

  function-statement:

/* C or C++ function definition or declaration */

declarations-definitions-seq *declaration-definition-seq* *:*

declaration-definition

declarations-definitions-seq *declaration-definition-seq* *declaration-definition*

openmp-construct:

parallel-construct

for-construct

sections-construct

single-construct

simd-construct

for-simd-construct

parallel-for-simd-construct

target-data-construct

target-construct

target-update-construct

teams-construct

distribute-construct

distribute-simd-construct

distribute-parallel-for-construct

distribute-parallel-for-simd-construct

target-teams-construct

teams-distribute-construct

teams-distribute-simd-construct

target-teams-distribute-construct

target-teams-distribute-simd-construct

teams-distribute-parallel-for-construct

target-teams-distribute-parallel-for-construct

teams-distribute-parallel-for-simd-construct

target-teams-distribute-parallel-for-simd-construct

parallel-for-construct

parallel-sections-construct

task-construct

master-construct

critical-construct

atomic-construct

ordered-construct

openmp-directive:

barrier-directive

taskwait-directive

taskyield-directive

flush-directive

structured-block:

statement

parallel-construct:

parallel-directive structured-block

parallel-directive:

 # pragma omp parallel *parallel-clause*$_{optseq}$ *new-line*

parallel-clause:

unique-parallel-clause

data-default-clause

data-privatization-clause

data-privatization-in-clause

data-sharing-clause

data-reduction-clause

unique-parallel-clause:

if-clause

num_threads (*expression* **)**

copyin (*variable-list* **)**

for-construct:

for-directive iteration-statement

for-directive:

 # pragma omp for *for-clause*$_{optseq}$ *new-line* *for-clause:*

unique-for-clause

data-privatization-clause

data-privatization-in-clause

data-privatization-out-clause

data-reduction-clause

`nowait`

unique-for-clause:

`ordered`

`schedule (` *schedule-kind* **`)`**

`schedule (` *schedule-kind* **`,`** *expression* **`)`**

`collapse (` *expression* **`)`**

schedule-kind:

`static`

`dynamic`

`guided`

`auto`

`runtime`

sections-construct:

sections-directive section-scope

sections-directive:

`# pragma omp sections` *sections-clause*$_{optseq}$ *new-line*

sections-clause:

data-privatization-clause

data-privatization-in-clause

data-privatization-out-clause

data-reduction-clause

`nowait`

section-scope:

{ section-sequence }

section-sequence:

section-directive$_{opt}$ structured-block

section-sequence section-directive structured-block

section-directive:

 # pragma omp section *new-line*

single-construct:

single-directive structured-block

single-directive:

 # pragma omp single *single-clause$_{optseq}$ new-line*

single-clause:

unique-single-clause

data-privatization-clause

data-privatization-in-clause

nowait

unique-single-clause:

copyprivate (*variable-list* **)**

simd-construct:

simd-directive iteration-statement

simd-directive:

 # pragma omp simd *simd-clause$_{optseq}$ new-line*

simd-clause:

collapse (*expression* **)**

aligned-clause

linear-clause

uniform-clause

data-reduction-clause

inbranch-clause

inbranch-clause:

inbranch

notinbranch

uniform-clause:

uniform (*variable-list* **)**

linear-clause:

`linear` **`(`** *variable-list* **`)`**

`linear` **`(`** *variable-list* **`:`** *expression* **`)`**

aligned-clause:

`aligned` **`(`** *variable-list* **`)`**

`aligned` **`(`** *variable-list* **`:`** *expression* **`)`**

declare-simd-construct:

declare-simd-directive-seq function-statement

declare-simd-directive-seq:

declare-simd-directive

declare-simd-directive-seq declare-simd-directive

declare-simd-directive:

`# pragma omp declare simd` *declare-simd-clause*$_{optseq}$ *new-line*

declare-simd-clause:

`simdlen` **`(`** *expression* **`)`**

aligned-clause

linear-clause

uniform-clause

data-reduction-clause

inbranch-clause

for-simd-construct:

for-simd-directive iteration-statement

for-simd-directive:

`# pragma omp for simd` *for-simd-clause*$_{optseq}$ *new-line*

for-simd-clause:

for-clause

simd-clause

parallel-for-simd-construct:

parallel-for-simd-directive iteration-statement

parallel-for-simd-directive:

`# pragma omp parallel for simd` *parallel-for-simd-clause*$_{optseq}$ *new-line* *parallel-for-simd-clause:*

parallel-for-clause

simd-clause

target-data-construct:

target-data-directive structured-block

target-data-directive:

 # pragma omp target data *target-data-clause*$_{optseq}$ *new-line*

target-data-clause:

device-clause

map-clause

if-clause

device-clause:

device (*expression* **)**

map-clause:

map (*map-type*$_{opt}$ *variable-array-section-list* **)**

map-type:

alloc:

to:

from:

tofrom:

target-construct:

target-directive structured-block

target-directive:

 # pragma omp target *target-clause*$_{optseq}$ *new-line*

target-clause:

device-clause

map-clause

if-clause

target-update-construct:

target-update-directive structured-block

target-update-directive:

 # pragma omp target update *target-update-clause*$_{seq}$ $_{optseq}$ *new-line*

target-update-clause:

motion-clause

device-clause

if-clause

motion-clause:

to (*variable-array-section-list* **)**

from (*variable-array-section-list* **)**

declare-target-construct:

declare-target-directive declarations-definitions-seq declaration-definition-seq end-declare-target-directive

declare-target-directive:

 # pragma omp declare target *new-line*

end-declare-target-directive:

 # pragma omp end declare target *new-line*

teams-construct:

teams-directive structured-block

teams-directive:

 # pragma omp teams *teams-clause*$_{optseq}$ *new-line*

teams-clause:

num_teams (*expression* **)**

thread_limit (*expression* **)**

data-default-clause

data-privatization-clause

data-privatization-in-clause

data-sharing-clause

data-reduction-clause

distribute-construct:

distribute-directive iteration-statement

distribute-directive:

 # pragma omp distribute *distribute-clause*$_{optseq}$ *new-line* *distribute-clause:*

data-privatization-clause

data-privatization-in-clause

collapse (*expression* **)**

dist_schedule (static)

dist_schedule (static , *expression* **)**

distribute-simd-construct:

distribute-simd-directive iteration-statement

distribute-simd-directive:

#pragma omp distribute simd *distribute-simd-clause*$_{optseq}$ *new-line*

distribute-simd-clause:

distribute-clause

simd-clause

distribute-parallel-for-construct:

distribute-parallel-for-directive iteration-statement

distribute-parallel-for-directive:

#pragma omp distribute parallel for *distribute-parallel-for-clause*$_{optseq}$ *new-line*

distribute-parallel-for-clause:

distribute-clause

parallel-for-clause

distribute-parallel-for-simd-construct:

distribute-parallel-for-simd-directive iteration-statement

distribute-parallel-for-simd-directive:

#pragma omp distribute parallel for *distribute-parallel-for-simd-clause*$_{optseq}$ *new-line*

distribute-parallel-for-simd-clause:

distribute-clause

parallel-for-simd-clause

target-teams-construct:

target-teams-directive iteration-statement

target-teams-directive:

#pragma omp target teams *target-teams-clause*$_{optseq}$ *new-line*

target-teams-clause:

target-clause

teams-clause

teams-distribute-construct:

teams-distribute-directive iteration-statement

teams-distribute-directive:

#pragma omp teams distribute *teams-distribute-clause*$_{optseq}$ *new-line*

teams-distribute-clause:

teams-clause

distribute-clause

teams-distribute-simd-construct:

teams-distribute-simd-directive iteration-statement

teams-distribute-simd-directive:

#pragma omp teams distribute simd *teams-distribute-simd-clause*$_{optseq}$ *new-line*

teams-distribute-simd-clause:

teams-clause

distribute-simd-clause

target-teams-distribute-construct:

target-teams-distribute-directive iteration-statement

target-teams-distribute-directive:

#pragma omp target teams distribute *target-teams-distribute-clause*$_{optseq}$ *new-line*

target-teams-distribute-clause:

target-clause

teams-distribute-clause

target-teams-distribute-simd-construct:

target-teams-distribute-simd-directive iteration-statement

target-teams-distribute-simd-directive:

#pragma omp target teams distribute simd *target-teams-distribute-simd-clause*$_{optseq}$ *new-line*

target-teams-distribute-simd-clause:

target-clause

teams-distribute-simd-clause

teams-distribute-parallel-for-construct:

teams-distribute-parallel-for-directive iteration-statement

teams-distribute-parallel-for-directive:

#pragma omp teams distribute parallel for *teams-distribute-parallel-for-clause*$_{optseq}$ *new-line*

teams-distribute-parallel-for-clause:

teams-clause

distribute-parallel-for-clause

target-teams-distribute-parallel-for-construct:

target-teams-distribute-parallel-for-directive iteration-statement

target-teams-distribute-parallel-for-directive:

#pragma omp teams distribute parallel for *target-teams-distribute-parallel-for-clause*$_{optseq}$ *new-line*

target-teams-distribute-parallel-for-clause:

target-clause

teams-distribute-parallel-for-clause

teams-distribute-parallel-for-simd-construct:

teams-distribute-parallel-for-simd-directive iteration-statement

teams-distribute-parallel-for-simd-directive:

#pragma omp teams distribute parallel for simd *teams-distribute-parallel-for-simd-clause*$_{optseq}$ *new-line*

teams-distribute-parallel-for-simd-clause:

teams-clause

distribute-parallel-for-simd-clause

target-teams-distribute-parallel-for-simd-construct:

target-teams-distribute-parallel-for-simd-directive iteration-statement

target-teams-distribute-parallel-for-simd-directive:

#pragma omp target teams distribute parallel for simd *target-teams-distribute-parallel-for-simd-clause*$_{optseq}$ *new-line*

target-teams-distribute-parallel-for-simd-clause:

target-clause

teams-distribute-parallel-for-simd-clause

task-construct:

task-directive structured-block

task-directive:

 # pragma omp task *task-clause*$_{optseq}$ *new-line*

task-clause:

unique-task-clause

data-default-clause

data-privatization-clause

data-privatization-in-clause

data-sharing-clause

unique-task-clause:

if-clause

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

untied

mergeable

depend (*dependence-type* **:** *variable-array-section-list* **)**

dependence-type:

in

out

inout

parallel-for-construct:

parallel-for-directive iteration-statement

parallel-for-directive:

 # pragma omp parallel for *parallel-for-clause*$_{optseq}$ *new-line*

parallel-for-clause:

unique-parallel-clause

unique-for-clause

data-default-clause

data-privatization-clause

data-privatization-in-clause

data-privatization-out-clause

data-sharing-clause

data-reduction-clause

parallel-sections-construct:

parallel-sections-directive section-scope

parallel-sections-directive:

 # pragma omp parallel sections *parallel-sections-clause*$_{optseq}$ *new-line*

parallel-sections-clause:

unique-parallel-clause

data-default-clause

data-privatization-clause

data-privatization-in-clause

data-privatization-out-clause

data-sharing-clause

data-reduction-clause

master-construct:

master-directive structured-block

master-directive:

 # pragma omp master *new-line*

critical-construct:

critical-directive structured-block

critical-directive:

 # pragma omp critical *region-phrase*$_{opt}$ *new-line*

region-phrase:

(identifier)

barrier-directive:

 # pragma omp barrier *new-line*

taskwait-directive:

 # pragma omp taskwait *new-line*

taskgroup-construct:

taskgroup-directive structured-block

taskgroup-directive:

 # pragma omp taskgroup *new-line*

taskyield-directive:

 # pragma omp taskyield *new-line*

atomic-construct:

atomic-directive expression-statement

atomic-directive structured block

atomic-directive:

 # pragma omp atomic *atomic-clause*$_{opt}$ *seq_cst-clauseopt new-line*

atomic-clause:

read

write

update

capture

seq-cst-clause:

seq_cst

flush-directive:

 # pragma omp flush *flush-vars*$_{opt}$ *new-line*

flush-vars:

(variable-list)

ordered-construct:

ordered-directive structured-block

ordered-directive:

 # pragma omp ordered *new-line*

cancel-directive:

 # pragma omp cancel *construct-type-clause if-clause*$_{opt}$ *new-line*

construct-type-clause:

parallel

sections

for

taskgroup

cancellation-point-directive:

 # pragma omp cancellation point *construct-type-clause new-line*

declaration:

/* standard declarations */

threadprivate-directive

declare-simd-directive

declare-target-construct

declare-reduction-directive

threadprivate-directive:

 # pragma omp threadprivate (*variable-list* **)** *new-line*

declare-reduction-directive:

 # pragma omp declare reduction (*reduction-identifier* **:** *reduction-type-list* **:** *expression* **)** *initializer-clause$_{opt}$* *new-line*

reduction-identifier:

C

identifier

C

C++

id-expression

C++

C / C++

one of: **+ * - & ^ | && || min max**

C / C++

reduction-type-list:

type-id

reduction-type-list, type-id

initializer-clause:

C

initializer (*identifier* **=** *initializer* **)**

initializer (*identifier* **(** *argument-expression-list* **))**

C

C++

initializer (*identifier initializer* **)**

initializer (*id-expression* **(** *expression-list* **))**

C++

data-default-clause:

default (shared)

default (none)

data-privatization-clause:

private (*variable-list* **)**

data-privatization-in-clause:

firstprivate (*variable-list* **)**

data-privatization-out-clause:

lastprivate (*variable-list* **)**

data-sharing-clause:

shared (*variable-list* **)**

data-reduction-clause:

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

if-clause:

if (*scalar-expression* **)**

C

array-section:

identifier array-section-subscript

variable-list:

identifier

variable-list **,** *identifier*

variable-array-section-list:

identifier

array-section

variable-array-section-list **,** *identifier*

variable-array-section-list **,** *array-section*

C

C++

array-section:

id-expression array-section-subscript

variable-list:

id-expression

variable-list **,** *id-expression*

variable-array-section-list:

id-expression

array-section

variable-array-section-list **,** *id-expression*

variable-array-section-list **,** *array-section*

C++

array-section-subscript:

array-section-subscript [$expression_{opt}$ **:** $expression_{opt}$ *]*

array-section-subscript [expression]

[$expression_{opt}$ **:** $expression_{opt}$ *]*

[expression]

APPENDIX C

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.

C.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 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;

/*
 * 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 void omp_set_nested(int nested);
extern int omp_get_cancellation(void);
extern int omp_get_nested(void);
extern void omp_set_schedule(omp_sched_t kind, int modifier);
extern void omp_get_schedule(omp_sched_t *kind, int *modifier);
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 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 void omp_init_lock(omp_lock_t *lock);
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_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);

#ifdef __cplusplus

#endif

#endif
```

C.2 Example of an Interface Declaration `include` File

omp_lib_kinds.h:

```
integer omp_lock_kind
     integer omp_nest_lock_kind
! this selects an integer that is large enough to hold a 64 bit integer
     parameter ( omp_lock_kind = selected_int_kind( 10 ) )
     parameter ( omp_nest_lock_kind = selected_int_kind( 10 ) )
     integer omp_sched_kind
! this selects an integer that is large enough to hold a 32 bit integer
     parameter ( omp_sched_kind = selected_int_kind( 8 ) )
     integer ( omp_sched_kind ) omp_sched_static
     parameter ( omp_sched_static = 1 )
     integer ( omp_sched_kind ) omp_sched_dynamic
     parameter ( omp_sched_dynamic = 2 )
     integer ( omp_sched_kind ) omp_sched_guided
     parameter ( omp_sched_guided = 3 )
     integer ( omp_sched_kind ) omp_sched_auto
     parameter ( omp_sched_auto = 4 )
     integer omp_proc_bind_kind
     parameter ( omp_proc_bind_kind = selected_int_kind( 8 ) )
     integer ( omp_proc_bind_kind ) omp_proc_bind_false
     parameter ( omp_proc_bind_false = 0 )
     integer ( omp_proc_bind_kind ) omp_proc_bind_true
     parameter ( omp_proc_bind_true = 1 )
     integer ( omp_proc_bind_kind ) omp_proc_bind_master
     parameter ( omp_proc_bind_master = 2 )
     integer ( omp_proc_bind_kind ) omp_proc_bind_close
     parameter ( omp_proc_bind_close = 3 )
     integer ( omp_proc_bind_kind ) omp_proc_bind_spread
     parameter ( omp_proc_bind_spread = 4 )
```

omp_lib.h:

```
! default integer type assumed below
! default logical type assumed below
! OpenMP API v4.0

     include 'omp_lib_kinds.h'
     integer openmp_version
     parameter ( openmp_version = 201307 )

     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
integer 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_in_final
logical omp_in_final

integer ( omp_proc_bind_kind ) omp_get_proc_bind
```

```
external omp_get_proc_bind

external omp_init_lock
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_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
```

C.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_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
        end module omp_lib_kinds

       module omp_lib

         use omp_lib_kinds

!                                    OpenMP API v4.0
         integer, parameter :: openmp_version = 201307

        interface

         subroutine omp_set_num_threads (number_of_threads_expr)
          integer, intent(in) :: number_of_threads_expr
         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 (enable_expr)
 logical, intent(in) ::enable_expr
end subroutine omp_set_dynamic

function omp_get_dynamic ()
 logical :: omp_get_dynamic
end function omp_get_dynamic

function omp_get_cancellation ()
 integer :: omp_get_cancellation
end function omp_get_cancellation

subroutine omp_set_nested (enable_expr)
 logical, intent(in) :: enable_expr
end subroutine omp_set_nested

function omp_get_nested ()
 logical :: omp_get_nested
end function omp_get_nested

subroutine omp_set_schedule (kind, modifier)
 use omp_lib_kinds
 integer(kind=omp_sched_kind), intent(in) :: kind
 integer, intent(in) :: modifier
end subroutine omp_set_schedule

subroutine omp_get_schedule (kind, modifier)
 use omp_lib_kinds
 integer(kind=omp_sched_kind), intent(out) :: kind
```

```
 integer, intent(out)::modifier
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(var)
 integer, intent(in) :: var
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( )
 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 omp_get_proc_bind

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

subroutine omp_init_lock (var)
 use omp_lib_kinds
 integer (kind=omp_lock_kind), intent(out) :: var
end subroutine omp_init_lock

subroutine omp_destroy_lock (var)
 use omp_lib_kinds
 integer (kind=omp_lock_kind), intent(inout) :: var
end subroutine omp_destroy_lock

subroutine omp_set_lock (var)
 use omp_lib_kinds
 integer (kind=omp_lock_kind), intent(inout) :: var
end subroutine omp_set_lock

subroutine omp_unset_lock (var)
 use omp_lib_kinds
 integer (kind=omp_lock_kind), intent(inout) :: var
end subroutine omp_unset_lock

function omp_test_lock (var)
 use omp_lib_kinds
 logical :: omp_test_lock
 integer (kind=omp_lock_kind), intent(inout) :: var
end function omp_test_lock

subroutine omp_init_nest_lock (var)
```

```
  use omp_lib_kinds
  integer (kind=omp_nest_lock_kind), intent(out) :: var
 end subroutine omp_init_nest_lock

 subroutine omp_destroy_nest_lock (var)
  use omp_lib_kinds
  integer (kind=omp_nest_lock_kind), intent(inout) :: var
 end subroutine omp_destroy_nest_lock

 subroutine omp_set_nest_lock (var)
  use omp_lib_kinds
  integer (kind=omp_nest_lock_kind), intent(inout) :: var
 end subroutine omp_set_nest_lock

 subroutine omp_unset_nest_lock (var)
  use omp_lib_kinds
  integer (kind=omp_nest_lock_kind), intent(inout) :: var
 end subroutine omp_unset_nest_lock

 function omp_test_nest_lock (var)
  use omp_lib_kinds
  integer :: omp_test_nest_lock
  integer (kind=omp_nest_lock_kind), intent(inout) :: var
 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

 end interface

end module omp_lib
```

C.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(number_of_threads_expr)
      use omp_lib_kinds
      integer(4), intent(in) :: number_of_threads_expr
    end subroutine omp_set_num_threads_4

    subroutine omp_set_num_threads_8(number_of_threads_expr)
      use omp_lib_kinds
      integer(8), intent(in) :: number_of_threads_expr
    end subroutine omp_set_num_threads_8

end interface omp_set_num_threads
```

APPENDIX D

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).
- **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 16).
- **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 19).
- **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 (see Section 2.3.2 on page 35).
- **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 47).
- **Thread affinity**: With $T \leq P$, when T does not divide P evenly, the assignment of the remaining $P - T * S$ places into subpartitions is implementation defined. With $T > P$, when P does not divide T evenly, the assignment of the remaining $T - P * S$ threads into places 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 49).

- **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. See Section 2.7.1 on page 54.
- **`sections` construct**: the method of scheduling the structured blocks among threads in the team is implementation defined (see Section 2.7.2 on page 61).
- **`single` construct**: the method of choosing a thread to execute the structured block is implementation defined (see Section 2.7.3 on page 63)
- **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 **`aligned`** clause is not specified, the assumed alignment is implementation defined (see Section 2.8.1 on page 68).
- **declare simd construct**: if the **`simdlen`** clause is not specified, the number of concurrent arguments for the function is implementation defined. If the **`aligned`** clause is not specified, the assumed alignment is implementation defined (see Section 2.8.2 on page 72).
- **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 2.10.5 on page 95).
- If no **`dist_schedule`** clause is specified then the schedule for the **`distribute`** construct is implementation defined (see Section 2.10.6 on page 98).
- **`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 2.12.6 on page 135).
- **`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 200).
- **`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 212).
- **`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 216).

- **`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 217).
- **`OMP_SCHEDULE` environment variable**: if the value of the variable does not conform to the specified format then the result is implementation defined (see Section 4.1 on page 244).
- **`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 4.2 on page 245).
- **`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 4.4 on page 246).
- **`OMP_DYNAMIC` environment variable**: if the value is neither **`true`** nor **`false`** the behavior is implementation defined (see Section 4.3 on page 246).
- **`OMP_NESTED` environment variable**: if the value is neither **`true`** nor **`false`** the behavior is implementation defined (see Section 4.6 on page 249).
- **`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 4.7 on page 249).
- **`OMP_WAIT_POLICY` environment variable**: the details of the **`ACTIVE`** and **`PASSIVE`** behaviors are implementation defined (see Section 4.8 on page 250).
- **`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 4.9 on page 251).
- **`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 4.10 on page 251).
- **`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 4.5 on page 247).

- **Thread affinity policy**: if the affinity request for a **parallel** construct cannot be fulfilled, the behavior of the thread affinity policy is implementation defined for that **parallel** construct.

Fortran

- **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 2.14.2 on page 159).

- **shared clause**: passing a shared variable to a non-intrinsic procedure may result in the value of the shared variable being copied into temporary storage before the procedure reference, and back out of the temporary storage into the actual argument storage after the procedure reference. Situations where this occurs other than those specified are implementation defined (see Section 2.14.3.2 on page 166).

- **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 199).

Fortran

APPENDIX E

Features History

This appendix summarizes the major changes between recent versions of the OpenMP API since version 2.5.

E.1 Version 3.1 to 4.0 Differences

- Various changes throughout the specification were made to provide initial support of Fortran 2003 (see Section 1.6 on page 20).
- C/C++ array syntax was extended to support array sections (see Section 2.4 on page 42).
- The **proc_bind** clause (see Section 2.5.2 on page 49), the **OMP_PLACES** environment variable (see Section 4.5 on page 247), and the **omp_get_proc_bind** runtime routine (see Section 3.2.22 on page 224) were added to support thread affinity policies.
- SIMD constructs were added to support SIMD parallelism (see Section 2.8 on page 68).
- Device constructs (see Section 2.10 on page 86), the **OMP_DEFAULT_DEVICE** environment variable (see Section 4.13 on page 253), 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 2.9.3 on page 84).
- The **depend** clause (see Section 2.9.1.1 on page 81) was added to support task dependences.
- The **taskgroup** construct (see Section 2.12.5 on page 134) was added to support more flexible deep task synchronization.

- The **reduction** clause (see Section 2.14.3.6 on page 176) was extended and the **declare reduction** construct (see Section 2.15 on page 190) was added to support user defined reductions.
- The **atomic** construct (see Section 2.12.6 on page 135) 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 2.13.1 on page 148), the **cancellation point** construct (see Section 2.13.2 on page 152), the **omp_get_cancellation** runtime routine (see Section 3.2.9 on page 209) and the **OMP_CANCELLATION** environment variable (see Section 4.11 on page 252) were added to support the concept of cancellation.
- The **OMP_DISPLAY_ENV** environment variable (see Section 4.12 on page 252) 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.2 Version 3.0 to 3.1 Differences

- The **final** and **mergeable** clauses (see Section 2.9.1 on page 78) were added to the **task** construct to support optimization of task data environments.
- The **taskyield** construct (see Section 2.9.2 on page 83) was added to allow user-defined task scheduling points.
- The **atomic** construct (see Section 2.12.6 on page 135) 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 2.14.3.4 on page 171).
- Data environment restrictions were changed to allow Fortran pointers in **firstprivate** (see Section 2.14.3.4 on page 171) and **lastprivate** (see Section 2.14.3.5 on page 174).
- New reduction operators **min** and **max** were added for C and C++
- The nesting restrictions in Section 2.16 on page 197 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 the code in the atomic construct.
- The **omp_in_final** runtime library routine (see Section 3.2.21 on page 223) 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 4.2 on page 245), 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 47).
- The *bind-var* ICV has been added, which controls whether or not threads are bound to processors (see Section 2.3.1 on page 34). The value of this ICV can be set with the **OMP_PROC_BIND** environment variable (see Section 4.4 on page 246).
- Descriptions of examples (see Appendix Section A on page 254) were expanded and clarified.
- Replaced incorrect use of **omp_integer_kind** in Fortran interfaces (see Section C.3 on page 294 and Section C.4 on page 299) with **selected_int_kind(8)**.

E.3 Version 2.5 to 3.0 Differences

The concept of tasks has been added to the OpenMP execution model (see Section 1.2.4 on page 8 and Section 1.3 on page 13).

- The **task** construct (see Section 2.9 on page 78) has been added, which provides a mechanism for creating tasks explicitly.
- The **taskwait** construct (see Section 2.12.4 on page 133) 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 16). 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 34). 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 200, Section 3.2.7 on page 206 and Section 3.2.10 on page 209).
- 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 47).
- 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 54).

- 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 54).
- 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 54).
- 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 54).
- Fortran assumed-size arrays now have predetermined data-sharing attributes (see Section 2.14.1.1 on page 154).
- In Fortran, **firstprivate** is now permitted as an argument to the **default** clause (see Section 2.14.3.1 on page 165).
- 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 2.14.3.3 on page 168).
- In Version 3.0, Fortran allocatable arrays may appear in **private**, **firstprivate**, **lastprivate**, **reduction**, **copyin** and **copyprivate** clauses. (see Section 2.14.2 on page 159, Section 2.14.3.3 on page 168, Section 2.14.3.4 on page 171, Section 2.14.3.5 on page 174, Section 2.14.3.6 on page 176, Section 2.14.4.1 on page 184 and Section 2.14.4.2 on page 185).
- In Version 3.0, static class members variables may appear in a **threadprivate** directive (see Section 2.14.2 on page 159).
- Version 3.0 makes clear where, and with which arguments, constructors and destructors of private and threadprivate class type variables are called (see Section 2.14.2 on page 159, Section 2.14.3.3 on page 168, Section 2.14.3.4 on page 171, Section 2.14.4.1 on page 184 and Section 2.14.4.2 on page 185).
- 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 212 and Section 3.2.13 on page 214).
- 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 34, Section 3.2.14 on page 215 and Section 4.10 on page 251).
- 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 34, Section 3.2.15 on page 216, Section 3.2.16 on page 217 and Section 4.9 on page 251).

- 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 34 and Section 4.7 on page 249).
- 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 34 and Section 4.8 on page 250).
- 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 218).
- 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 220).
- 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 221).
- 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 222).
- In Version 3.0, locks are owned by tasks, not by threads (see Section 3.3 on page 232).

Index

D