
OpenMP Application Program Interface

Version 2.5 May 2005

Copyright © 1997-2005 OpenMP Architecture Review Board.
Permission to copy without fee all or part of this material is granted,
provided the OpenMP Architecture Review Board copyright notice and the
title of this document appear. Notice is given that copying is by permission
of OpenMP Architecture Review Board.

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29

1. Introduction	1
1.1 Scope	1
1.2 Glossary	2
1.2.1 Threading Concepts	2
1.2.2 OpenMP language terminology	2
1.2.3 Data Terminology	7
1.2.4 Implementation Terminology	8
1.3 Execution Model	9
1.4 Memory Model	10
1.4.1 Structure of the OpenMP Memory Model	10
1.4.2 The Flush Operation	11
1.4.3 OpenMP Memory Consistency	12
1.5 OpenMP Compliance	13
1.6 Normative References	14
1.7 Organization of this document	14
2. Directives	17
2.1 Directive Format	18
2.1.1 Fixed Source Form Directives	19
2.1.2 Free Source Form Directives	20
2.2 Conditional Compilation	21
2.2.1 Fixed Source Form Conditional Compilation Sentinels	22
2.2.2 Free Source Form Conditional Compilation Sentinel	23
2.3 Internal Control Variables	24
2.4 parallel Construct	26
2.4.1 Determining the Number of Threads for a parallel Region	29
2.5 Work-sharing Constructs	32
2.5.1 Loop Construct	33

1	2.5.1.1	Determining the Schedule of a	
2		Work-sharing Loop	38
3	2.5.2	sections Construct	39
4	2.5.3	single Construct	42
5	2.5.4	workshare Construct	44
6	2.6	Combined Parallel Work-sharing Constructs	46
7	2.6.1	Parallel loop construct	47
8	2.6.2	parallel sections Construct	48
9	2.6.3	parallel workshare Construct	50
10	2.7	Master and Synchronization Constructs	51
11	2.7.1	master Construct	51
12	2.7.2	critical Construct	52
13	2.7.3	barrier Construct	54
14	2.7.4	atomic Construct	55
15	2.7.5	flush Construct	58
16	2.7.6	ordered Construct	61
17	2.8	Data Environment	63
18	2.8.1	Sharing Attribute Rules	63
19	2.8.1.1	Sharing Attribute Rules for Variables Referenced	
20		in a Construct	63
21	2.8.1.2	Sharing Attribute Rules for Variables Referenced	
22		in a Region, but not in a Construct	65
23	2.8.2	threadprivate Directive	66
24	2.8.3	Data-Sharing Attribute Clauses	70
25	2.8.3.1	default clause	71
26	2.8.3.2	shared clause	72
27	2.8.3.3	private clause	73
28	2.8.3.4	firstprivate clause	75
29	2.8.3.5	lastprivate clause	77
30	2.8.3.6	reduction clause	79

1	2.8.4	Data Copying Clauses	83
2	2.8.4.1	<code>copyin</code> clause	84
3	2.8.4.2	<code>copyprivate</code> clause	85
4	2.9	Nesting of Regions	87
5	3.	Runtime Library Routines	89
6	3.1	Runtime Library Definitions	90
7	3.2	Execution Environment Routines	91
8	3.2.1	<code>omp_set_num_threads</code>	91
9	3.2.2	<code>omp_get_num_threads</code>	93
10	3.2.3	<code>omp_get_max_threads</code>	94
11	3.2.4	<code>omp_get_thread_num</code>	95
12	3.2.5	<code>omp_get_num_procs</code>	96
13	3.2.6	<code>omp_in_parallel</code>	96
14	3.2.7	<code>omp_set_dynamic</code>	97
15	3.2.8	<code>omp_get_dynamic</code>	99
16	3.2.9	<code>omp_set_nested</code>	100
17	3.2.10	<code>omp_get_nested</code>	101
18	3.3	Lock Routines	102
19	3.3.1	<code>omp_init_lock</code> and <code>omp_init_nest_lock</code>	104
20	3.3.2	<code>omp_destroy_lock</code> and <code>omp_destroy_nest_lock</code>	105
21	3.3.3	<code>omp_set_lock</code> and <code>omp_set_nest_lock</code>	105
22	3.3.4	<code>omp_unset_lock</code> and <code>omp_unset_nest_lock</code>	106
23	3.3.5	<code>omp_test_lock</code> and <code>omp_test_nest_lock</code>	107
24	3.4	Timing Routines	108
25	3.4.1	<code>omp_get_wtime</code>	109
26	3.4.2	<code>omp_get_wtick</code>	110
27	4.	Environment Variables	113
28	4.1	<code>OMP_SCHEDULE</code>	114
29	4.2	<code>OMP_NUM_THREADS</code>	115

1	4.3	<code>OMP_DYNAMIC</code>	116
2	4.4	<code>OMP_NESTED</code>	116
3	A.	Examples	119
4	A.1	A Simple Parallel Loop	119
5	A.2	The OpenMP Memory Model	120
6	A.3	Conditional Compilation	122
7	A.4	The <code>parallel</code> Construct	123
8	A.5	The <code>num_threads</code> Clause	125
9	A.6	Fortran Restrictions on the <code>do</code> Construct	125
10	A.7	Fortran Private Loop Iteration Variables	127
11	A.8	The <code>nowait</code> clause	128
12	A.9	The <code>parallel sections</code> Construct	129
13	A.10	The <code>single</code> Construct	130
14	A.11	The <code>workshare</code> Construct	132
15	A.12	The <code>master</code> Construct	136
16	A.13	The <code>critical</code> Construct	138
17	A.14	Work-Sharing Constructs Inside a <code>critical</code> Construct	139
18	A.15	Binding of <code>barrier</code> Regions	140
19	A.16	The <code>atomic</code> Construct	142
20	A.17	Restrictions on the <code>atomic</code> Construct	144
21	A.18	The <code>flush</code> Construct with a List	147
22	A.19	The <code>flush</code> Construct without a List	150
23	A.20	Placement of <code>flush</code> and <code>barrier</code> Directives	153
24	A.21	The <code>ordered</code> Clause and the <code>ordered</code> Construct	154
25	A.22	The <code>threadprivate</code> Directive	158
26	A.23	Fortran Restrictions on <code>shared</code> and <code>private</code> Clauses with Common Blocks	163
27			
28	A.24	The <code>default(none)</code> Clause	165
29	A.25	Race Conditions Caused by Implied Copies of Shared Variables in Fortran	167
30			

1	A.26	The <code>private</code> Clause	168
2	A.27	Reprivatization	170
3	A.28	Fortran Restrictions on Storage Association with the	
4		<code>private</code> Clause	171
5	A.29	C/C++ Arrays in a <code>firstprivate</code> Clause	174
6	A.30	The <code>lastprivate</code> Clause	175
7	A.31	The <code>reduction</code> Clause	176
8	A.32	The <code>copyin</code> Clause	180
9	A.33	The <code>copyprivate</code> Clause	181
10	A.34	Nested Loop Constructs	185
11	A.35	Restrictions on Nesting of Regions	187
12	A.36	The <code>omp_set_dynamic</code> and <code>omp_set_num_threads</code>	
13		Routines	193
14	A.37	The <code>omp_get_num_threads</code> Routine	195
15	A.38	The <code>omp_init_lock</code> Routine	197
16	A.39	Simple Lock Routines	198
17	A.40	Nestable Lock Routines	200
18	B.	Stubs for Runtime Library Routines	203
19	B.1	C/C++ Stub routines	204
20	B.2	Fortran Stub Routines	209
21	C.	OpenMP C and C++ Grammar	215
22	C.1	Notation	215
23	C.2	Rules	216
24	D.	Interface Declarations	223
25	D.1	Example of the <code>omp.h</code> Header File	223
26	D.2	Example of an Interface Declaration <code>include</code> File	225
27	D.3	Example of a Fortran 90 Interface Declaration <code>module</code>	227
28	D.4	Example of a Generic Interface for a Library Routine	232

1

E. Implementation Defined Behaviors in OpenMP233

2

F. Changes from Version 2.0 to Version 2.5237

3

Introduction

This document specifies a collection of compiler directives, library routines, and environment variables that can be used to specify shared-memory parallelism in C, C++ and Fortran programs. This functionality collectively defines the specification of the *OpenMP Application Program Interface (OpenMP API)*. 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 OpenMP 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, work-sharing constructs, and synchronization constructs, and they provide support for the sharing and privatization of 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 user 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 dependencies, conflicts, deadlocks, race conditions, or other problems that result from non-conforming programs. The user is responsible for using OpenMP in his application to produce a conforming program. OpenMP 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 having a serial flow of control and an associated stack.
- thread-safe routine** A routine that performs the intended function even when executed concurrently (by more than one *thread*).

1.2.2 OpenMP language terminology

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

COMMENT: Current *base languages* for OpenMP are C90, C99, C++, Fortran 77, Fortran 90, and Fortran 95.

original 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.

For Fortran, a block of executable statements with a single entry at the top and a single exit at the bottom.

COMMENTS:

For both languages, the point of entry cannot be a labeled statement and the point of exit cannot be a branch of any type.

For C/C++:

- The point of entry cannot 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*.

1 For Fortran:

- 2 • **STOP** statements are allowed in a *structured block*.

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

5 COMMENT: See Section 2.1 on page 18 for a description of OpenMP
6 directive syntax.

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

8 **OpenMP program** A program that consists of an *original program*, annotated with *OpenMP*
9 *directives*.

10 **declarative directive** An *OpenMP directive* that may only be placed in a declarative context. A
11 *declarative directive* has no associated executable user code, but instead has
12 one or more associated user declarations.

13 COMMENT: Only the **threadprivate** *directive* is a *declarative directive*.

14 **executable directive** An *OpenMP directive* that is not declarative, i.e., it may be placed in an
15 executable context.

16 COMMENT: All *directives* except the **threadprivate** *directive* are
17 *executable directives*.

18 **standalone directive** An OpenMP *executable directive* that has no associated executable user code.

19 COMMENT: Only the **barrier** and **flush** *directives* are *standalone*
20 *directives*.

21 **simple directive** An OpenMP *executable directive* whose associated user code must be a
22 simple (single, non-compound) executable statement.

23 COMMENT: Only the **atomic** *directive* is a *simple directive*.

24 **loop directive** An OpenMP *executable directive* whose associated user code must be a loop
25 that is a *structured block*.

26 COMMENTS:

27 For C/C++, only the **for** *directive* is a *loop directive*.

28 For Fortran, only the **do** *directive* and the optional **end do** *directive*
29 are *loop directives*.

1 **structured directive** An OpenMP *executable directive* that is neither a *standalone directive*, a
2 *simple directive* nor a *loop directive*.

3 For C/C++, all *structured directives* have associated user code that is the
4 following *structured block*.

5 For Fortran, all *structured directives* are paired with an associated **end**
6 *directive* except **section**, whose end is marked either by the next **section**
7 or by the **end sections**. These structured directives bracket the associated
8 user code that forms a *structured block*.

9 **construct** An OpenMP *executable directive* (and for Fortran, the paired **end directive**, if
10 any) and the associated statement, loop or *structured block*, if any, not
11 including the code in any called routines, i.e., the lexical extent of an
12 *executable directive*.

13 **region** All code encountered during a specific instance of the execution of a given
14 *construct* or OpenMP library routine. A *region* includes any code in called
15 routines as well as any implicit code introduced by the OpenMP
16 implementation.

17 COMMENTS:

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

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

22 **sequential part** All code encountered during the execution of an *OpenMP program* that is not
23 enclosed by a **parallel region** corresponding to an explicit **parallel**
24 *construct*.

25 COMMENTS:

26 The *sequential part* executes as if it were enclosed by an *inactive*
27 **parallel region** called the *implicit parallel region*.

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

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

32 **nested region** A *region* (dynamically) enclosed by another *region*, i.e., a *region* executed in
33 its entirety during another *region*.

34 COMMENT: Some nestings are *conforming* and some are not. See Section 2.9
35 on page 87 for the rules specifying the *conforming* nestings.

1 **binding region** For a *region* whose *binding thread set* is the current *team*, the enclosing *region*
2 that determines the execution context and limits the scope of the effects of the
3 bound *region*.

4 *Binding region* is not defined for *regions* whose *binding thread set* is all
5 *threads* or the encountering *thread*.

6 COMMENTS:

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

9 For all other *regions* with whose *binding thread set* is the current *team*,
10 the *binding region* is the innermost enclosing **parallel** *region*.
11 When such a *region* is encountered outside of any explicit **parallel**
12 *region*, the *binding region* is the *implicit parallel* *region* enclosing
13 the *sequential part*.

14 A **parallel** *region* need not be *active* to be a *binding region*.

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

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

20 **worksharing**
21 **construct**

22 A *construct* that defines units of work, each of which is executed exactly once
23 by a *thread* in the *team* executing the *construct*.

24 For C, *worksharing constructs* are **for**, **sections**, and **single**.

25 For Fortran, *worksharing constructs* are **do**, **sections**, **single** and
26 **workshare**.

27 **active parallel region** A **parallel** *region* whose **if** clause evaluates to true.

28 COMMENT: A missing **if** clause is equivalent to an **if** clause that evaluates
29 to true.

30 **inactive parallel**
31 **region**

32 A **parallel** *region* that is not an *active parallel region*, i.e., a serialized
33 **parallel** *region*.

34 An *inactive parallel region* is always executed by a *team* of only one *thread*.

35 **implicit parallel**
36 **region**

 The *inactive parallel region* that encloses the *sequential part* of an *OpenMP*
 program.

1	initial thread	The <i>thread</i> that executes the <i>sequential part</i> .
2	master thread	A <i>thread</i> that encounters (the start of) a parallel <i>region</i> and creates a <i>team</i> .
3	team	A set of one or more <i>threads</i> participating in the execution of a parallel
4		<i>region</i> .
5		For an <i>active parallel region</i> , the team comprises the <i>master thread</i> and
6		additional <i>threads</i> that may be launched.
7		For an <i>inactive parallel region</i> , the <i>team</i> only includes the <i>master thread</i> .
8	barrier	A point in the execution of a program encountered by a <i>team</i> , beyond which
9		no <i>thread</i> in the team may execute until all <i>threads</i> in the <i>team</i> have reached
10		that point.

1.2.3 Data Terminology

12	variable	A named data object, whose value can be defined and redefined during the
13		execution of a program.
14		Only an object that is not part of another object is considered a <i>variable</i> . For
15		example, array elements, structure components, array sections and substrings
16		are not considered <i>variables</i> .
17	private variable	A <i>variable</i> whose name provides access to a different block of storage for
18		each <i>thread</i> in a <i>team</i> .
19	shared variable	A <i>variable</i> whose name provides access to the same block of storage for all
20		<i>threads</i> in a <i>team</i> .
21	global-lifetime	
22	memory	Memory locations that persist during the entire execution of the <i>original</i>
23		<i>program</i> , according to the <i>base language</i> specification.
24	threadprivate	
25	memory	<i>Global-lifetime memory</i> locations that are replicated, one per <i>thread</i> , by the
26		OpenMP implementation.

1 **defined** For *variables*, the property of having a valid value.
2 For C:
3 For the contents of *variables*, the property of having a valid value.
4 For C++:
5 For the contents of *variables* of POD (plain old data) type, the property of
6 having a valid value.
7 For *variables* of non-POD class type, the property of having been constructed
8 but not subsequently destructed.
9 For Fortran:
10 For the contents of *variables*, the property of having a valid value. For the
11 allocation or association status of *variables*, the property of having a valid
12 status.
13 COMMENT: Programs that rely upon *variables* that are not *defined* are *non-*
14 *conforming programs*.

15 1.2.4 Implementation Terminology

16 **supporting n levels of**
17 **parallelism** Implies allowing an *active parallel region* to be enclosed by $n-1$ *active*
18 *parallel regions*, where the *team* associated with each *active parallel region*
19 has more than one *thread*.
20 **supporting OpenMP** Supporting at least one level of parallelism.
21 **supporting nested**
22 **parallelism** Supporting more than one level of parallelism.
23 **conforming program** An *OpenMP program* that follows all the rules and restrictions of the OpenMP
24 specification.
25 **compliant**
26 **implementation** An implementation of the OpenMP specification that compiles and executes
27 any *conforming program* as defined by the specification.
28 COMMENT: A *compliant implementation* may exhibit *unspecified behavior*
29 when compiling or executing a *non-conforming program*.

1 **parallel** regions may be arbitrarily nested inside each other. If nested parallelism is
2 disabled, or is not supported by the OpenMP implementation, then the new team that is
3 created by a thread encountering a **parallel** construct inside a **parallel** region
4 will consist only of the encountering thread. However, if nested parallelism is supported
5 and enabled, then the new team can consist of more than one thread.

6 When any team encounters a work-sharing construct, the work inside the construct is
7 divided among the members of the team and executed co-operatively instead of being
8 executed by every thread. There is an optional barrier at the end of work-sharing
9 constructs. Execution of code by every thread in the team resumes after the end of the
10 work-sharing construct.

11 Synchronization constructs and library routines are available in OpenMP to co-ordinate
12 threads and data in **parallel** and work-sharing constructs. In addition, library
13 routines and environment variables are available to control or query the runtime
14 environment of OpenMP programs.

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

20 1.4 Memory Model

21 1.4.1 Structure of the OpenMP Memory Model

22 OpenMP provides a relaxed-consistency, shared-memory model. All OpenMP threads
23 have access to a place to store and retrieve variables, called the *memory*. In addition,
24 each thread is allowed to have its own *temporary view* of the memory. The temporary
25 view of memory for each thread is not a required part of the OpenMP memory model,
26 but can represent any kind of intervening structure, such as machine registers, cache, or
27 other local storage, between the thread and the memory. The temporary view of memory
28 allows the thread to cache variables and thereby avoid going to memory for every
29 reference to a variable. Each thread also has access to another type of memory that must
30 not be accessed by other threads, called *threadprivate memory*.

31 A **parallel** directive determines two kinds of access to variables used in the
32 associated structured block: shared and private. Each variable referenced in the
33 structured block has an original variable, which is the variable by the same name that

1 exists in the program immediately outside the **parallel** construct. Each reference to a
2 shared variable in the structured block becomes a reference to the original variable. For
3 each private variable referenced in the structured block, a new version of the original
4 variable (of the same type and size) is created in memory for each thread of the team
5 formed to execute the **parallel** region associated with the **parallel** directive,
6 except possibly for the master thread of the team. References to a private variable in the
7 structured block refer to the current thread's private version of the original variable.

8 If multiple threads write to the same shared variable without synchronization, the
9 resulting value of the variable in memory is unspecified. If at least one thread reads from
10 a shared variable and at least one thread writes to it without synchronization, the value
11 seen by any reading thread is unspecified.

12 It is implementation defined as to whether, and in what sizes, memory accesses by
13 multiple threads to the same variable without synchronization are atomic with respect to
14 each other.

15 A private variable in an outer **parallel** region belonging to, or accessible from, a
16 thread that eventually becomes the master thread of an inner nested **parallel** region,
17 is permitted to be accessed by any of the threads of the team executing the inner
18 **parallel** region, unless the variable is also private with respect to the inner
19 **parallel** region. Any other access by one thread to the private variables of another
20 thread results in unspecified behavior.

21 1.4.2 The Flush Operation

22 The memory model has relaxed-consistency because a thread's temporary view of
23 memory is not required to be consistent with memory at all times. A value written to a
24 variable can remain in the thread's temporary view until it is forced to memory at a later
25 time. Likewise, a read from a variable may retrieve the value from the thread's
26 temporary view, unless it is forced to read from memory. The OpenMP flush operation
27 enforces consistency between the temporary view and memory.

28 The flush operation is applied to a set of variables called the *flush-set*. The flush
29 operation restricts reordering of memory operations that an implementation might
30 otherwise do. Implementations must not reorder the code for a memory operation for a
31 given variable, or the code for a flush operation for the variable, with respect to a flush
32 operation that refers to the same variable.

33 If a thread has captured the value of a write in its temporary view of a variable since its
34 last flush of that variable, then when it executes another flush of the variable, the flush
35 does not complete until the value of the variable has been written to the variable in
36 memory. A flush of a variable executed by a thread also causes its temporary view of the
37 variable to be discarded, so that if its next memory operation for that variable is a read,
38 then the thread will read from memory and may capture the value in the temporary view.

1 When a thread executes a flush, no later memory operation by that thread for a variable
2 involved in that flush is allowed to start until the flush completes. The completion of a
3 flush of a set of variables executed by a thread is defined as the point at which all writes
4 to those variables done by that thread are visible in memory to all other threads and the
5 temporary view, for that thread, of all variables involved, is discarded.

6 The flush operation provides a guarantee of consistency between a thread's temporary
7 view and memory. Therefore, the flush operation can be used to guarantee that a value
8 written to a variable by one thread may be read by a second thread. To accomplish this,
9 the programmer must ensure that the second thread has not written to the variable since
10 its last flush of the variable, and that the following sequence of events happens in the
11 specified order:

- 12 1. The value is written to the variable by the first thread.
- 13 2. The variable is flushed by the first thread.
- 14 3. The variable is flushed by the second thread.
- 15 4. The value is read from the variable by the second thread.

16 The **volatile** keyword in the C and C++ languages specifies a consistency
17 mechanism that is related to the OpenMP memory consistency mechanism in the
18 following way: a reference that reads the value of an object with a volatile-qualified type
19 behaves as if there were a flush operation on that object at the previous sequence point,
20 while a reference that modifies the value of an object with a volatile-qualified type
21 behaves as if there were a flush operation on that object at the next sequence point.

22 1.4.3 OpenMP Memory Consistency

23 The type of relaxed memory consistency provided by OpenMP is similar to *weak*
24 *ordering*¹. OpenMP does not apply restrictions to the reordering of memory operations
25 executed by a single thread except for those related to a flush operation.

26 The restrictions in Section 1.4.2 on page 11 on reordering with respect to flush
27 operations guarantee the following:

- 28 • If the intersection of the flush-sets of two flushes performed by two different threads
29 is non-empty, then the two flushes must be completed as if in some sequential order,
30 seen by all threads.

31 1. Weak ordering is described in *S. V. Adve and K. Gharachorloo, "Shared Memory Consistency Models: A Tutorial", IEEE*
32 *Computer, 29(12), pp.66-76, December 1996*. Weak ordering requires that some memory operations be defined as
33 synchronization operations and that these be ordered with respect to each other. In the context of OpenMP, two flushes of
the same variable are synchronization operations. The OpenMP memory model is slightly weaker than weak ordering,
however, because flushes whose flush-sets have an empty intersection are not ordered with respect to each other.

- If the intersection of the flush-sets of two flushes performed by one thread is non-empty, then the two flushes must appear to be completed in that thread's program order.
- 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 2.7.5 on page 58 for details. For an example illustrating the memory model, see Section A.2 on page 120.

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 14) 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 (e.g., **ALLOCATE** and **DEALLOCATE** statements must be thread-safe in Fortran). Unsynchronized concurrent use of such routines by different threads must produce correct results (though not necessarily the same as serial execution results, as in the case of random number generation routines).

In both Fortran 90 and Fortran 95, 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 E 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 E.

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.

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: Examples
- Appendix B: Stubs for Runtime Library Routines
- Appendix C: OpenMP C and C++ Grammar
- Appendix D: Interface Declarations
- Appendix E: Implementation Defined Behaviors in OpenMP
- Appendix F: Changes from Version 2.0 to Version 2.5

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 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....
▲

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 18)
- Mechanisms to control conditional compilation (Section 2.2 on page 21)
- Control of OpenMP API internal control variables (Section 2.3 on page 24)
- Details of each OpenMP directive (Section 2.4 on page 26 to Section 2.9 on page 87)

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 OpenMP 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 C, 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.

Directives are case-sensitive.

An OpenMP 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 19 and Section 2.1.2 on page 20.

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

1 Only one *directive-name* can be specified per directive (note that this includes combined
2 directives, see Section 2.6 on page 46). The order in which clauses appear on directives
3 is not significant. Clauses on directives may be repeated as needed, subject to the
4 restrictions listed in the description of each clause.

5 Some data-sharing attribute clauses (Section 2.8.3 on page 70), data copying clauses
6 (Section 2.8.4 on page 83), the **threadprivate** directive (Section 2.8.2 on page 66),
7 and the **flush** directive (Section 2.7.5 on page 58) accept a *list*. A *list* consists of a
8 comma-separated collection of one or more *list items*.

9 **C/C++**

10 A list item is a variable name, subject to the restrictions specified in each of the sections
11 describing clauses and directives for which a *list* appears.

12 **C/C++**

13 **Fortran**

14 A list item is a variable name or common block name (enclosed in slashes), subject to
15 the restrictions specified in each of the sections describing clauses and directives for
16 which a *list* appears.

17 **Fortran**

18 **Fortran**

19 2.1.1 Fixed Source Form Directives

20 The following sentinels are recognized in fixed form source files:

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

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

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

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

```

1      c23456789
2      !$omp parallel do shared(a,b,c)
3
4      c$omp parallel do
5      c$omp+shared(a,b,c)
6
7      c$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 nonblank 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 nonblank 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 pair of keywords:

```

25          end critical
26          end do
27          end master

```

```
1         end ordered
2         end parallel
3         end sections
4         end single
5         end workshare
6         parallel do
7         parallel sections
8         parallel workshare
```

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

```
11 !23456789
12         !$omp parallel do &
13             !$omp shared(a,b,c)
14
15         !$omp parallel &
16         !$omp&do shared(a,b,c)
17
18         !$omp paralleldo shared(a,b,c)
```

17 Fortran

18 2.2 Conditional Compilation

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

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

24 For examples of conditional compilation, see Section A.3 on page 122.

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
```

1 #endif

2 2.2.2 Free Source Form Conditional Compilation Sentinel

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

```
5 !$
```

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

- 8 • The sentinel can appear in any column but must be preceded only by white space.
- 9 • The sentinel must appear as a single word with no intervening white space.
- 10 • Initial lines must have a space after the sentinel.
- 11 • Continued lines must have an ampersand as the last nonblank character on the line,
12 prior to any comment appearing on the conditionally compiled line. (Continued lines
13 can have an ampersand after the sentinel, with optional white space before and after
14 the ampersand.)

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

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

```
20 c23456789  
21 !$ iam = omp_get_thread_num() + &  
22 !$& index  
  
23 #ifdef _OPENMP  
24 iam = omp_get_thread_num() + &  
25 index  
26 #endif
```

27 Fortran

2.3 Internal Control Variables

An OpenMP implementation must act as if there were internal control variables that store the information for determining the number of threads to use for a **parallel** region and how to schedule a work-sharing loop. The control variables are given values at various times (described below) during execution of an OpenMP program. They are initialized by the implementation itself and may be given values by using OpenMP environment variables, and by calls to OpenMP API routines. The only way for the program to retrieve the values of these control variables is by calling OpenMP API routines.

For purposes of exposition, this document refers to the control variables by certain names (below), 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 Table 2.1.

The following control variables store values that affect the operation of **parallel** regions:

- *nthreads-var* - stores the number of threads requested for future **parallel** regions.
- *dyn-var* - controls whether dynamic adjustment of the number of threads to be used for future **parallel** regions is enabled.
- *nest-var* - controls whether nested parallelism is enabled for future **parallel** regions.

The following control variables store values that affect the operation of loop regions:

- *run-sched-var* - stores scheduling information to be used for loop regions using the **runtime** schedule clause.
- *def-sched-var* - stores implementation defined default scheduling information for loop regions.

Table 2-1 shows the methods for modifying and retrieving the values of each control variable, as well as their initial values.

TABLE 2-1 Control variables

Control variable	Ways to modify value	Way to retrieve value	Initial value
<i>nthreads-var</i>	OMP_NUM_THREADS omp_set_num_threads()	omp_get_max_threads()	Implementation defined
<i>dyn-var</i>	OMP_DYNAMIC omp_set_dynamic()	omp_get_dynamic()	Implementation defined
<i>nest-var</i>	OMP_NESTED omp_set_nested()	omp_get_nested()	<i>false</i>
<i>run-sched-var</i>	OMP_SCHEDULE	(none)	Implementation defined
<i>def-sched-var</i>	(none)	(none)	Implementation defined

The effect of the API routines in Table 2-1 on the internal control variables described in this specification applies only during the execution of the sequential part of the program. During execution of the sequential part, only one copy of each internal control variable may exist. The effect of these API routines on the internal control variables is implementation defined when the API routines are executed from within any explicit **parallel** region. Additionally, the number of copies of the internal control variables, and their effects, during the execution of any explicit parallel region are implementation defined.

The internal control variables are each given values before any OpenMP construct or OpenMP API routine executes. The initial values of *nthreads-var*, *dyn-var*, *run-sched-var*, and *def-sched-var* are implementation defined. The initial value of *nest-var* is *false*. After the initial values are assigned, but also before any OpenMP construct or OpenMP API routine executes, the values of any OpenMP environment variables that were set by the user are read and the associated control variables are modified accordingly. After this point, no changes to any OpenMP environment variables will be reflected in the control variables. During execution of the user's code, certain control variables can be further modified by certain OpenMP API routine calls. An OpenMP construct clause does not modify the value of any of these control variables.

Table 2-2 shows the override relationships between various construct clauses, OpenMP API routines, environment variables, and initial values.

TABLE 2-2 Override relationships

construct clause, if used	...overrides previous call to OpenMP API routine	...overrides environment variable, if set	...overrides initial value
<code>num_threads</code> clause	<code>omp_set_num_threads()</code>	<code>OMP_NUM_THREADS</code>	initial value of <i>nthreads-var</i>
(none)	<code>omp_set_dynamic()</code>	<code>OMP_DYNAMIC</code>	initial value of <i>dyn-var</i>
(none)	<code>omp_set_nested()</code>	<code>OMP_NESTED</code>	initial value of <i>nest-var</i>
(none)	(none)	<code>OMP_SCHEDULE</code> (only used when schedule kind is <code>runtime</code>)	initial value of <i>run-sched-var</i>
<code>schedule</code> clause	(none)	(none)	initial value of <i>def-sched-var</i>

Cross References:

- **parallel** construct, see Section 2.4 on page 26.
- Loop construct, see Section 2.5.1 on page 33.
- `omp_set_num_threads` routine, see Section 3.2.1 on page 91.
- `omp_set_dynamic` routine, see Section 3.2.7 on page 97.


- `omp_set_nested` routine, see Section 3.2.9 on page 100.
- `omp_get_max_threads` routine, see Section 3.2.3 on page 94.
- `omp_get_dynamic` routine, see Section 3.2.8 on page 99.
- `omp_get_nested` routine, see Section 3.2.10 on page 101.
- `OMP_NUM_THREADS` environment variable, see Section 4.2 on page 115.
- `OMP_DYNAMIC` environment variable, see Section 4.3 on page 116.
- `OMP_NESTED` environment variable, see Section 4.4 on page 116.
- `OMP_SCHEDULE` environment variable, see Section 4.1 on page 114.

2.4 parallel Construct

Summary

This is the fundamental construct that starts parallel execution. See Section 1.3 on page 9 for a general description of the OpenMP execution model.

Syntax

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)
    private(list)
    firstprivate(list)
    default(shared | none)
    shared(list)
    copyin(list)
```

```
reduction(operator: list)
num_threads(integer-expression)
```

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)
private(list)
firstprivate(list)
default(private | shared | none)
shared(list)
copyin(list)
reduction( {operator | intrinsic_procedure_name} :list)
num_threads(scalar-integer-expression)
```

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 2.4.1 on page 29 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 which encountered the **parallel** construct

1 becomes the master thread of the new team, with a thread number of zero for the
2 duration of the **parallel** region. All threads in the new team, including the master
3 thread, execute the region. Once the team is created, the number of threads in the team
4 remains constant for the duration of that **parallel** region.

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

9 The structured block of the **parallel** construct is executed by each thread, although
10 each thread can execute a path of statements that is different from the other threads.

11 There is an implied barrier at the end of a **parallel** region. Only the master thread of
12 the team continues execution after the end of a **parallel** region.

13 If a thread in a team executing a **parallel** region encounters another **parallel**
14 directive, it creates a new team, according to the rules in Section 2.4.1 on page 29, and
15 it becomes the master of that new team.



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

21 For an example of the **parallel** construct, see Section A.4 on page 123. For an
22 example of the **num_threads** clause, see Section A.5 on page 125.



23 Restrictions

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

- 25 • A program which branches into or out of a **parallel** region is non-conforming.
- 26 • A program must not depend on any ordering of the evaluations of the clauses of the
27 **parallel** directive, or on any side effects of the evaluations of the clauses.
- 28 • At most one **if** clause can appear on the directive.
- 29 • At most one **num_threads** clause can appear on the directive. The **num_threads**
30 expression must evaluate to a positive integer value.

31  C/C++ 

- 32 • A **throw** executed inside a **parallel** region must cause execution to resume
33 within the same **parallel** region, and it must be caught by the same thread that
34 threw the exception.

35  C/C++ 

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

Cross References

- **default**, **shared**, **private**, **firstprivate**, and **reduction** clauses, see Section 2.8.3 on page 70.
- **copyin** clause, see Section 2.8.4 on page 83.
- **omp_get_thread_num** routine, see Section 3.2.4 on page 95.

2.4.1 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, the number of levels of parallelism supported, and the values of the `nthreads-var`, `dyn-var` and `nest-var` internal control variables are used to determine the number of threads to use in the region. Figure 2-1 describes how the number of threads is determined. 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 executing inside an active `parallel` region encounters a `parallel` construct, the new team which is created will consist of only the encountering thread, when any of the following conditions hold:

- nested parallelism is disabled,
- the `if` clause expression evaluates to false, or
- no further levels of parallelism are supported by the OpenMP implementation.

However, if nested parallelism is enabled and additional levels of parallelism are supported, then the new team can consist of more than one thread.

The number of levels of parallelism supported is implementation defined. If only one level of parallelism is supported (i.e. nested parallelism is not supported) then the value of the `nest-var` internal control variable is always *false*.

If dynamic adjustment of the number of threads is enabled, the number of threads that are used for executing subsequent `parallel` regions may be adjusted automatically by the implementation. Once the number of threads is determined, it remains fixed for the

1 duration of that **parallel** region. If dynamic adjustment of the number of threads is
2 disabled, the number of threads that are used for executing subsequent **parallel**
3 regions may not be adjusted by the implementation.

4 It is implementation defined whether the ability to dynamically adjust the number of
5 threads is provided. If this ability is not provided, then the value of the *dyn-var* internal
6 control variable is always *false*.

7 Implementations may deliver fewer threads than indicated in Figure 2-1, in exceptional
8 situations, such as when there is a lack of resources, even if dynamic adjustment is
9 disabled. In these exceptional situations the behavior of the program is implementation
10 defined: this may, for example, include interrupting program execution.

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

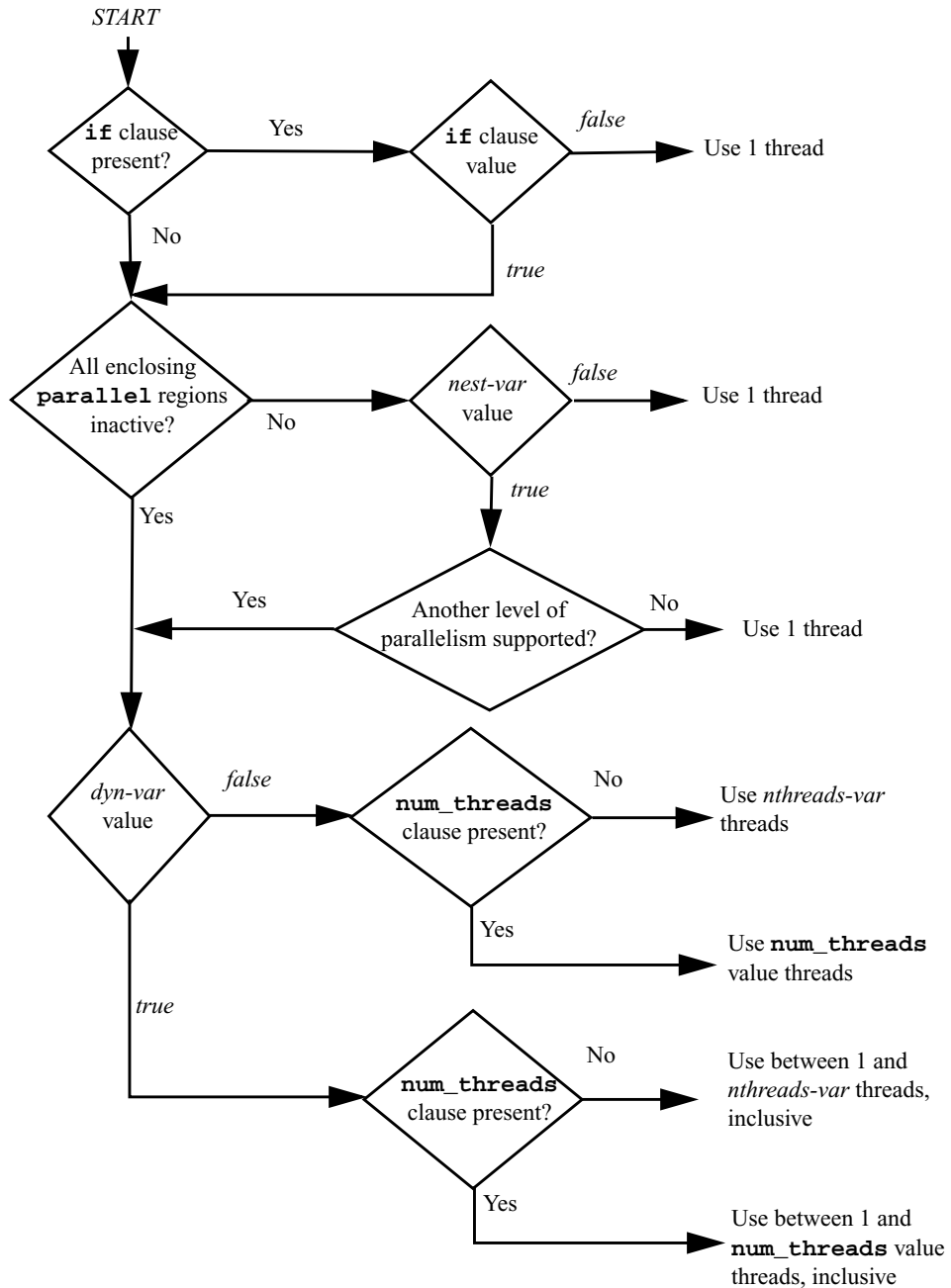


FIGURE 2-1 Determining the number of threads for a parallel region. Note that no ordering of evaluation of the `if` and `num_threads` clauses is implied.

2.5 Work-sharing Constructs

A work-sharing construct distributes the execution of the associated region among the members of the team that encounters it. A work-sharing region must bind to an active **parallel** region in order for the work-sharing region to execute in parallel. If execution encounters a work-sharing region in the sequential part, it is executed by the initial thread.

A work-sharing construct does not launch new threads, and a work-sharing region has no barrier on entry. However, an implied barrier exists at the end of the work-sharing region, unless a **nowait** clause is specified. If a **nowait** clause is present, an implementation may omit code to synchronize the threads at the end of the work-sharing region. In this case, threads that finish early may proceed straight to the instructions following the work-sharing region without waiting for the other members of the team to finish the work-sharing region, and without performing a flush operation (see Section A.8 on page 128 for an example.)

OpenMP defines the following work-sharing constructs, and these are described in the sections that follow:

- loop construct
- **sections** construct
- **single** construct

- **workshare** construct
- 

Restrictions

The following restrictions apply to work-sharing constructs:

- Each work-sharing region must be encountered by all threads in a team or by none at all.
- The sequence of work-sharing regions and **barrier** regions encountered must be the same for every thread in a team.

2.5.1 Loop Construct

Summary

The loop construct specifies that the iterations of the associated loop will be executed in parallel. The iterations of the loop 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-loop
```

The *clause* is one of the following:

```
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
ordered
schedule(kind[, chunk_size])
nowait
```

1 The **for** directive places restrictions on the structure of the corresponding *for-loop*.
2 Specifically, the corresponding *for-loop* must have the following canonical form:

3 **for** (*init-expr*; *var relational-op b*; *incr-expr*) *statement*

4 *init-expr* One of the following:
5 $var = lb$
6 $integer\text{-}type\ var = lb$

7 *incr-expr* One of the following:
8 $++var$
9 $var++$
10 $--var$
11 $var--$
12 $var += incr$
13 $var -= incr$
14 $var = var + incr$
15 $var = incr + var$
16 $var = var - incr$

17 *var* A signed integer variable, of type *integer-type*, as defined in the
18 base language. If this variable would otherwise be shared, it is
19 implicitly made private on the loop construct. This variable
20 must not be modified during the execution of the *for-loop* other
21 than in *incr-expr*. Unless the variable is specified
22 **lastprivate** on the loop construct, its value after the loop is
23 undefined.

24 *relational-op* One of the following:
25 $<$
26 $<=$
27 $>$
28 $>=$

29 *lb*, *b*, and *incr* Loop invariant integer expressions. There is no implied
30 synchronization during the evaluation of these expressions. It is
31 unspecified whether, in what order, or how many times any side
32 effects within the *lb*, *b*, or *incr* expressions occur.

33 *statement* Defined according to the base language.

34 Note that the canonical form allows the number of loop iterations to be computed on
35 entry to the loop. This computation is performed with values in the type of *var*, after
36 integral promotions. In particular, if the value of $b - lb + incr$, or any intermediate result
37 required to compute this value, cannot be represented in that type, the behavior is
38 unspecified.



The syntax of the loop construct is as follows:

```

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

The *clause* is one of the following:

```

private(list)
firstprivate(list)
lastprivate(list)
reduction({operator | intrinsic_procedure_name} : list)
ordered
schedule(kind[, chunk_size])
    
```

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

The *do-loop* must be a *do-construct* as defined in Section 8.1.4.1 of the Fortran 95 standard. If an **end do** directive follows a *do-construct* in which several **DO** statements share a **DO** termination statement, then a **do** directive can only be specified for the first (i.e. outermost) of these **DO** statements. See Section A.6 on page 125 for examples.

If the loop iteration variable would otherwise be shared, it is implicitly made private on the loop construct. See Section A.7 on page 127 for examples. Unless the variable is specified **lastprivate** on the loop construct, its value after the loop is undefined.

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 (optional) implicit barrier of the loop region.

Description

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

1 The **schedule** clause specifies how iterations of the loop are divided into contiguous
2 non-empty subsets, called chunks, and how these chunks are assigned among threads of
3 the team. Programs which depend on which thread executes a particular iteration are
4 non-conforming. The *chunk_size* expression is evaluated using the original list items of
5 any variables that are made private for the duration of the loop construct. It is
6 unspecified whether, in what order, or how many times, any side-effects of the
7 evaluation of this expression occur.

8 See Section 2.5.1.1 on page 38 for details of how the schedule for a work-sharing loop
9 is determined.

10 The schedule *kind* can be one of those specified in Table 2-3.

11 **TABLE 2-3** **schedule** clause *kind* values

12 static	When schedule(static, chunk_size) is specified, iterations are divided 13 into chunks of size <i>chunk_size</i> , and the chunks are statically assigned to 14 threads in the team in a round-robin fashion in the order of the thread number. 15 Note that the last chunk to be assigned may have a smaller number of 16 iterations.
	When no <i>chunk_size</i> is specified, the iteration space is divided into chunks 17 which are approximately equal in size, and each thread is assigned at most one 18 chunk. 19
20 dynamic	When schedule(dynamic, chunk_size) is specified, the iterations are 21 assigned to threads in chunks as the threads request them. The thread executes 22 the chunk of iterations, then requests another chunk, until no chunks remain to 23 be assigned.
	Each chunk contains <i>chunk_size</i> iterations, except for the last chunk to be 24 assigned, which may have fewer iterations. 25
	When no <i>chunk_size</i> is specified, it defaults to 1.
27 guided	When schedule(guided, chunk_size) is specified, the iterations are 28 assigned to threads in chunks as the threads request them. The thread executes 29 the chunk of iterations, then requests another chunk, until no chunks remain to 30 be assigned.
	For a <i>chunk_size</i> of 1, the size of each chunk is proportional to the 31 number of unassigned iterations divided by the number of threads, 32 decreasing to 1. For a <i>chunk_size</i> with value <i>k</i> (greater than 1), the 33 size of each chunk is determined in the same way with the restriction 34 that the chunks do not contain fewer than <i>k</i> iterations (except for the last chunk 35 to be assigned, which may have fewer than <i>k</i> iterations). 36
	When no <i>chunk_size</i> is specified, it defaults to 1. 37

1 **runtime** When `schedule(runtime)` is specified, the decision regarding scheduling
2 is deferred until run time, and the schedule and chunk size are taken from the
3 *run-sched-var* control variable.

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

11 A compliant implementation of the **guided** schedule with a *chunk_size* value of k
12 would assign $q = \lceil n/p \rceil$ iterations to the first available thread and set n to the larger of
13 $n-q$ and $p*k$. It would then repeat this process until q is greater than or equal to the
14 number of remaining iterations, at which time the remaining iterations form the final
15 chunk. Another compliant implementation could use the same method, except with $q =$
16 $\lceil n/(2p) \rceil$, and set n to the larger of $n-q$ and $2*p*k$.

17 Restrictions

18 Restrictions to the loop construct are as follows:

- 19 • The values of the loop control expressions of the loop associated with the loop
20 directive must be the same for all the threads in the team.
- 21 • Only a single **schedule** clause can appear on a loop directive.
- 22 • *chunk_size* must be a loop invariant integer expression with a positive value.
- 23 • The value of the *chunk_size* expression must be the same for all threads in the team.
- 24 • When **schedule(runtime)** is specified, *chunk_size* must not be specified.
- 25 • Only a single **ordered** clause can appear on a loop directive.
- 26 • The **ordered** clause must be present on the loop construct if any **ordered** region
27 ever binds to a loop region arising from the loop construct.
- 28 • The loop iteration variable may not appear in a **threadprivate** directive.

29 C/C++

- 30 • The *for-loop* must be a structured block, and in addition, its execution must not be
31 terminated by a **break** statement.
- 32 • The *for-loop* iteration variable *var* must have a signed integer type.
- 33 • Only a single **nowait** clause can appear on a **for** directive.

- If *relational-op* is < or <= then *incr-expr* must cause *var* to increase on each iteration of the loop. Conversely, if *relational-op* is > or >= then *incr-expr* must cause *var* to decrease on each iteration of the loop.

C/C++

Fortran

- The *do-loop* must be a structured block, and in addition, its execution must not be terminated by an **EXIT** statement.
- 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

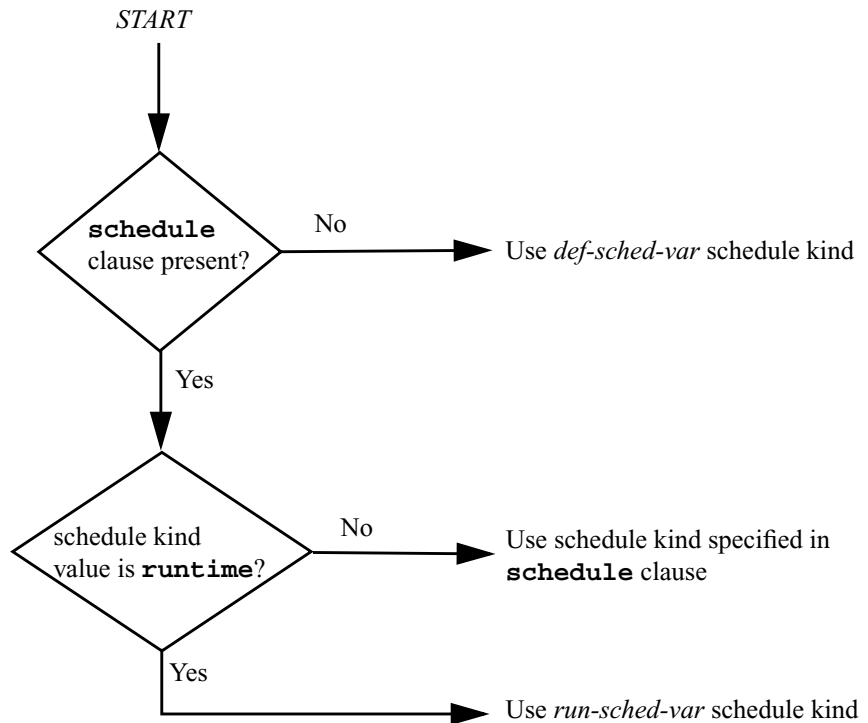
- **private**, **firstprivate**, **lastprivate**, and **reduction** clauses, see Section 2.8.3 on page 70.
- **OMP_SCHEDULE** environment variable, see Section 4.1 on page 114.
- **ordered** construct, see Section 2.7.6 on page 61.

2.5.1.1 Determining the Schedule of a Work-sharing Loop

When execution encounters a loop directive, the **schedule** clause (if any) on the directive, and the *run-sched-var* and *def-sched-var* internal control variables are used to determine how loop iterations are assigned to threads. See Section 2.3 on page 24 for details of how the values of the internal control variables are determined. If no **schedule** clause is used on the work-sharing loop directive, then the schedule is taken from the current value of *def-sched-var*. If the **schedule** clause is used and specifies the **runtime** schedule kind, then the schedule is taken from the *run-sched-var* control variable. Otherwise, the schedule is taken from the value of the **schedule** clause. Figure 2-2 describes how the schedule for a work-sharing loop is determined.

Cross References

- Internal control variables, see Section 2.3 on page 24.



9 **FIGURE 2-2** Determining the schedule for a work-sharing loop.

10 **2.5.2 sections Construct**

11 **Summary**

12 The **sections** construct is a noniterative work-sharing construct that contains a set of
 13 structured blocks that are to be divided among, and executed by, the threads in a team.
 14 Each structured block is executed once by one of the threads in the team.

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 ]
  ...
}
```

The *clause* is one of the following:

```
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: 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]
```


1 The *clause* is one of the following:

```
2     private(list)  
3     firstprivate(list)  
4     lastprivate(list)  
5     reduction( {operator | intrinsic_procedure_name } :list)
```

6  Fortran 

7 Binding

8 The binding thread set for a **sections** region is the current team. A **sections**
9 region binds to the innermost enclosing **parallel** region. Only the threads of the team
10 executing the binding **parallel** region participate in the execution of the structured
11 blocks and (optional) implicit barrier of the **sections** region.

12 Description

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

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

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

19 Restrictions

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

- 21 • The **section** directives must appear within the **sections** construct and may not
22 be encountered elsewhere in the **sections** region.
- 23 • The code enclosed in a **sections** construct must be a structured block.

24  C/C++ 

- 25 • Only a single **nowait** clause can appear on a **sections** directive.

26  C/C++ 

Cross References

- `private`, `firstprivate`, `lastprivate`, and `reduction` clauses, see Section 2.8.3 on page 70.

2.5.3 `single` Construct

Summary

The `single` construct specifies that the associated structured block is executed by only one thread in the team (not necessarily the master thread). The other threads in the team do not execute the block, and wait at an implicit barrier at the end of `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
```

The *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] ...]
```

1 The *clause* is one of the following:

```
2     private(list)  
3     firstprivate(list)
```

4 and *end_clause* is one of the following:

```
5     copyprivate(list)  
6     nowait
```

7  Fortran

8 **Binding**

9 The binding thread set for a **single** region is the current team. A **single** region
10 binds to the innermost enclosing **parallel** region. Only the threads of the team
11 executing the binding **parallel** region participate in the execution of the structured
12 block and (optional) implicit barrier of the **single** region.

13 **Description**

14 The method of choosing a thread to execute the structured block is implementation
15 defined. There is an implicit barrier after the **single** construct unless a **nowait** clause
16 is specified.

17 For an example of the **single** construct, see Section A.10 on page 130.

18 **Restrictions**

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

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

22 **Cross References**

- 23 • **private** and **firstprivate** clauses, see Section 2.8.3 on page 70.
- 24 • **copyprivate** clause, see Section 2.8.4.2 on page 85.

2.5.4 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.

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 (optional) implicit barrier of the **workshare** region.

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 have been completed 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.
- If a **workshare** directive is applied to an array assignment statement, the assignment of each element is a unit of work.
- If a **workshare** directive is applied to a scalar assignment statement, the assignment operation is a single unit of work.
- If a **workshare** directive is applied to a **WHERE** statement or construct, the evaluation of the mask expression and the masked assignments are workshared.
- If a **workshare** directive is applied to 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 workshared.
- For **atomic** constructs, the update of each scalar variable is a single unit of work.
- For **critical** constructs, each construct is a single unit of work.
- For **parallel** constructs, each construct is a single unit of work with respect to the **workshare** construct. The statements contained in **parallel** constructs are executed by new teams of threads formed for each **parallel** construct.
- If none of the rules above apply to a portion of a statement in the structured block, then that portion is a single 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.

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

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

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

8 For examples of the **workshare** construct, see Section A.11 on page 132.

9 **Restrictions**

10 The following restrictions apply to the **workshare** directive:

- 11 • The construct must not contain any user defined function calls unless the function is
12 **ELEMENTAL**.

13  Fortran 

14 **2.6 Combined Parallel Work-sharing** 15 **Constructs**

16 Combined parallel work-sharing constructs are shortcuts for specifying a work-sharing
17 construct nested immediately inside a **parallel** construct. The semantics of these
18 directives are identical to that of explicitly specifying a **parallel** construct containing
19 one work-sharing construct and no other statements.

20 The combined parallel work-sharing constructs allow certain clauses which are
21 permitted on both **parallel** constructs and on work-sharing constructs. If a program
22 would have different behavior depending on whether the clause were applied to the
23 **parallel** construct or to the work-sharing construct, then the program's behavior is
24 unspecified.

25 The following sections describe the combined parallel work-sharing constructs:

- 26 • The parallel loop construct.
- 27 • The **parallel sections** construct.

28  Fortran 

- 29 • The **parallel workshare** construct.

30  Fortran 

2.6.1 Parallel loop construct

Summary

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

The *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:

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

The *clause* can be any of the clauses accepted by the **parallel** or **do** directives, with identical meanings and restrictions. However, **nowait** may not be specified on an **end parallel do** directive.

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

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 2.4 on page 26.
- loop construct, see Section 2.5.1 on page 33.
- Data attribute clauses, see Section 2.8.3 on page 70.

2.6.2 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 ]
...
}
```


1 The *clause* can be any of the clauses accepted by the **parallel** or **sections**
2 directives, except the **nowait** clause, with identical meanings and restrictions.

3  C/C++ 

4  Fortran 

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

```
6 !$omp parallel sections [clause[[, clause] ...]  
7   [!$omp section]  
8     structured-block  
9   [!$omp section  
10     structured-block ]  
11   ...  
12 !$omp end parallel sections
```

13 The *clause* can be any of the clauses accepted by the **parallel** or **sections**
14 directives, with identical meanings and restrictions. However, **nowait** cannot be
15 specified on an **end parallel sections** directive.

16 The last section ends at the **end parallel sections** directive.

17  Fortran 

18 Description

19  C/C++ 

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

22  C/C++ 

23  Fortran 

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

27  Fortran 

28 For an example of the parallel sections construct, see Section A.9 on page 129.

29 Restrictions

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

Cross References:

- **parallel** construct, see Section 2.4 on page 26.
- **sections** construct, see Section 2.5.2 on page 39.
- Data attribute clauses, see Section 2.8.3 on page 70.

2.6.3 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
```

The *clause* can be any of the clauses accepted by the **parallel** directive, with identical meanings and restrictions. However, **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 2.4 on page 26.
- **workshare** construct, see Section 2.5.4 on page 44.

- Data attribute clauses, see Section 2.8.3 on page 70.

Fortran

2.7 Master and Synchronization Constructs

The following sections describe :

- the **master** construct.
- the **critical** construct.
- the **barrier** construct.
- the **atomic** construct.
- the **flush** construct.
- the **ordered** construct.

2.7.1 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.

For an example of the **master** construct, see Section A.12 on page 136.

2.7.2 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. Region execution is restricted to a single thread at a time among all the threads in the program, 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 other thread 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, not just in the current team.

C/C++

Identifiers used to identify a **critical** construct have external linkage and are in a name space which 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

For an example of the **critical** construct, see Section A.13 on page 138.

Restrictions

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

2.7.3 barrier Construct

Summary

The **barrier** construct specifies an explicit barrier at the point at which the construct appears.

Syntax

C/C++

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

```
#pragma omp barrier new-line
```

Note that because the **barrier** construct does not have a C language statement as part of its syntax, there are some restrictions on its placement within a program. The **barrier** directive may only be placed in the program at a position where ignoring or deleting the directive would result in a program with correct syntax. See Appendix C for the formal grammar. The examples in Section A.20 on page 153 illustrate these restrictions.

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. See Section A.15 on page 140 for examples.

Description

All of the threads of the team executing the binding **parallel** region must execute the **barrier** region before any are allowed to continue execution beyond the barrier.

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.
- The sequence of work-sharing regions and **barrier** regions encountered must be the same for every thread in a team.

2.7.4 **atomic Construct**

Summary

The **atomic** construct ensures that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

Syntax

C/C++

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

```
#pragma omp atomic new-line
                expression-stmt
```

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

```
x binop= expr
x++
```

1 ++x

2 x--

3 --x

4 In the preceding expressions:

- 5 • *x* is an lvalue expression with scalar type.
- 6 • *expr* is an expression with scalar type, and it does not reference the object designated
- 7 by *x*.
- 8 • *binop* is not an overloaded operator and is one of +, *, -, /, &, ^, |, <<, or >>.

9 C/C++

10 Fortran

11 The syntax of the **atomic** construct is as follows:

```
12 !$omp atomic
13     statement
```

14 where *statement* has one of the following forms:

15 *x* = *x operator expr*

16 *x* = *expr operator x*

17 *x* = *intrinsic_procedure_name* (*x*, *expr_list*)

18 *x* = *intrinsic_procedure_name* (*expr_list*, *x*)

19 In the preceding statements:

- 20 • *x* is a scalar variable of intrinsic type.
- 21 • *expr* is a scalar expression that does not reference *x*.
- 22 • *expr_list* is a comma-separated, non-empty list of scalar expressions that do not
- 23 reference *x*. When *intrinsic_procedure_name* refers to **IAND**, **IOR**, or **IEOR**, exactly
- 24 one expression must appear in *expr_list*.
- 25 • *intrinsic_procedure_name* is one of **MAX**, **MIN**, **IAND**, **IOR**, or **IEOR**.
- 26 • *operator* is one of +, *, -, /, **.AND.**, **.OR.**, **.EQV.**, or **.NEQV.**
- 27 • The operators in *expr* must have precedence equal to or greater than the precedence
- 28 of *operator*, *x operator expr* must be mathematically equivalent to *x operator (expr)*,
- 29 and *expr operator x* must be mathematically equivalent to *(expr) operator x*.
- 30 • *intrinsic_procedure_name* must refer to the intrinsic procedure name and not to other
- 31 program entities.

- *operator* must refer to the intrinsic operator and not to a user-defined operator.
- The assignment must be intrinsic assignment.

Fortran

Binding

The binding thread set for an **atomic** region is all threads. **atomic** regions enforce exclusive access with respect to other **atomic** regions that update the same storage location x among all the threads in the program without regard to the team(s) to which the threads belong.

Description

Only the load and store of the object designated by x are atomic; the evaluation of *expr* is not atomic. To avoid race conditions, all updates of the location which could potentially occur in parallel must be protected with an **atomic** directive. **atomic** regions do not enforce exclusive access with respect to any **critical** or **ordered** regions which access the same storage location x .

A compliant implementation may enforce exclusive access between **atomic** regions which update different storage locations. The circumstances under which this occurs are implementation defined.

For an example of the **atomic** construct, see Section A.16 on page 142.

Restrictions

C/C++

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

- All atomic references to the storage location x throughout the program are required to have a compatible type. See Section A.17 on page 144 for examples.

C/C++

Fortran

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

- All atomic references to the storage location of variable x throughout the program are required to have the same type and type parameters. See Section A.17 on page 144 for examples.

Fortran

Cross References

- `critical` construct, see Section 2.7.2 on page 52.

2.7.5 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 10 for more details.

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 only affects the view of memory from the thread which executes the region. Other threads must themselves execute a flush operation in order to observe the effects of the encountering thread's flush operation.

Description

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. 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.

C/C++

If a pointer is present in the list, the pointer itself is flushed, not the object 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 has the **ALLOCATABLE** attribute and the list item is allocated, the allocated array is flushed; otherwise the allocation status is flushed.

Fortran

For examples of the **flush** construct, see Section A.18 on page 147 and Section A.19 on page 150.

Note – 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 critical 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**.

Incorrect example:

a = b = 0

thread 1

```
b = 1  
flush(b)  
flush(a)  
if (a == 0) then  
    critical section  
end if
```

thread 2

```
a = 1  
flush(a)  
flush(b)  
if (b == 0) then  
    critical section  
end if
```

1 The problem with this example is that operations on variables **a** and **b** are not ordered
2 with respect to each other. For instance, nothing prevents the compiler from moving the
3 flush of **b** on thread 1 or the flush of **a** on thread 2 to a position completely after the
4 critical section (assuming that the critical section on thread 1 does not reference **b** and the
5 critical section on thread 2 does not reference **a**). If either re-ordering happens, the critical
6 section can be active on both threads simultaneously.

7 The following correct pseudocode example correctly ensures that the critical section is
8 executed by not more than one of the two threads at any one time. Notice that execution of
9 the critical section by neither thread is considered correct in this example.

10 *Correct example:*

11 **a = b = 0**

12 *thread 1*

13 **b = 1**
14 *flush(a,b)*
15 **if (a == 0) then**
16 *critical section*
17 **end if**

thread 2

a = 1
flush(a,b)
if (b == 0) then
critical section
end if

18 The compiler is prohibited from moving the flush at all for either thread, ensuring that the
19 respective assignment is complete and the data is flushed before the **if** statement is
20 executed.



21 **C/C++**

22 Note that because the **flush** construct does not have a C language statement as part of its
23 syntax, there are some restrictions on its placement within a program. The **flush**
24 directive may only be placed in the program at a position where ignoring or deleting the
25 directive would result in a program with correct syntax. See Appendix C for the formal
26 grammar. See Section A.20 on page 153 for an example that illustrates these placement
27 restrictions.

28 **C/C++**

29 A **flush** region without a list is implied at the following locations:

- 30 • During a **barrier** region.
- 31 • At entry to and exit from **parallel**, **critical** and **ordered** regions.
- 32 • At exit from work-sharing regions, unless a **nowait** is present.
- 33 • At entry to and exit from combined parallel work-sharing regions.

- 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.

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

- At entry to and exit from **atomic** regions, where the list contains only the object updated in the **atomic** construct.

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

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

2.7.6 ordered Construct

Summary

The **ordered** construct specifies a structured block in a loop region which 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 new-line
    structured-block
```

C/C++

Fortran

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

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

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 completely independently of each other.

Description

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 each of the previous iterations that contains an **ordered** region has completed the **ordered** region.

For examples of the **ordered** construct, see Section A.21 on page 154.

Restrictions

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

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

Cross References

- loop construct, see Section 2.5.1 on page 33.
- parallel loop construct, see Section 2.6.1 on page 47.

2.8 Data Environment

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

- Section 2.8.1 on page 63 describes how the sharing attributes of variables referenced in **parallel** regions are determined.
- The **threadprivate** directive, which is provided to create threadprivate memory, is described in Section 2.8.2 on page 66.
- Clauses that may be specified on directives to control the sharing attributes of variables referenced in **parallel** or work-sharing constructs are described in Section 2.8.3 on page 70.
- Clauses that may be specified on directives to copy data values from private or threadprivate objects on one thread to the corresponding objects on other threads in the team are described in Section 2.8.4 on page 83.

2.8.1 Sharing Attribute Rules

This section describes how the sharing attributes of variables referenced in **parallel** regions are determined. The following two cases are described separately:

- Section 2.8.1.1 on page 63 describes the sharing attribute rules for variables referenced in a construct.
- Section 2.8.1.2 on page 65 describes the sharing attribute rules for variables referenced in a region, but outside any construct.

2.8.1.1 Sharing Attribute Rules for Variables Referenced in a Construct

The sharing attributes of variables which are referenced in a construct may be one of the following: *predetermined*, *explicitly determined*, or *implicitly determined*.

Note that specifying a variable on a **firstprivate**, **lastprivate**, or **reduction** clause of an enclosed construct causes an implicit reference to the variable in the enclosing construct. Such implicit references are also subject to the following rules.

The following variables have predetermined sharing attributes:

- ▼ C/C++ ▼
- Variables appearing in **threadprivate** directives are threadprivate.

- Variables with automatic storage duration which are declared in a scope inside the construct are private.
- Variables with heap allocated storage are shared.
- Static data members are shared.
- The loop iteration variable in the *for-loop* of a **for** or **parallel for** construct is private in that construct.
- Variables with const-qualified type having no mutable member are shared.

C/C++

Fortran

- Variables and common blocks appearing in **threadprivate** directives are **threadprivate**.
- The loop iteration variable in the *do-loop* of a **do** or **parallel do** construct is private in that construct.
- Variables used as loop iteration variables in sequential loops in a **parallel** construct are private in the **parallel** construct.
- **implied-do** and **forall** indices are private.
- Cray pointees inherit the sharing attribute of the storage with which their Cray pointers are associated.

Fortran

Variables with predetermined sharing attributes may not be listed in data-sharing attribute clauses, with the following exceptions:

C/C++

- The loop iteration variable in the *for-loop* of a **for** or **parallel for** construct may be listed in a **private** or **lastprivate** clause.

C/C++

Fortran

- The loop iteration variable in the *do-loop* of a **do** or **parallel do** construct may be listed in a **private** or **lastprivate** clause.
- Variables used as loop iteration variables in sequential loops in a **parallel** construct may be listed in **private**, **firstprivate**, **lastprivate**, **shared**, or **reduction** clauses.

Fortran

Additional restrictions on the variables which may appear in individual clauses are described with each clause in Section 2.8.3 on page 70.

Variables referenced in the construct are said to have an explicitly determined sharing attribute if they are listed in a data-sharing attribute clause on the construct.

1 Variables referenced in the construct whose sharing attribute is not predetermined or
2 explicitly determined will have their sharing attribute implicitly determined. In a
3 **parallel** construct, the sharing attributes of these variables is determined by the
4 default clause, if present (see Section 2.8.3.1 on page 71). If no default clause is present,
5 variables with implicitly determined sharing attributes are shared. For other constructs,
6 variables with implicitly determined sharing attributes inherit their sharing attributes
7 from the enclosing context.

8 2.8.1.2 Sharing Attribute Rules for Variables Referenced in a Region, 9 but not in a Construct

10 The sharing attributes of variables which are referenced in a region, but not in a
11 construct, are determined as follows:

12 C/C++

- 13 • Static variables declared in called routines in the region are shared.
- 14 • Variables with const-qualified type having no mutable member, and that are declared
15 in called routines, are shared.
- 16 • File-scope or namespace-scope variables referenced in called routines in the region
17 are shared unless they appear in a **threadprivate** directive.
- 18 • Variables with heap-allocated storage are shared.
- 19 • Static data members are shared.
- 20 • Formal arguments of called routines in the region that are passed by reference inherit
21 the data-sharing attributes of the associated actual argument.
- 22 • Other variables declared in called routines in the region are private.

23 C/C++

24 Fortran

- 25 • Local variables declared in called routines in the region and that have the **save**
26 attribute, or that are data initialized, are shared unless they appear in a
27 **threadprivate** directive.
- 28 • Variables belonging to common blocks, or declared in modules, and referenced in
29 called routines in the region are shared unless they appear in a **threadprivate**
30 directive.
- 31 • Dummy arguments of called routines in the region that are passed by reference inherit
32 the data-sharing attributes of the associated actual argument.
- 33 • **implied-do** and **forall** indices are private.
- 34 • Cray pointees inherit the sharing attribute of the storage with which their Cray
35 pointers are associated.

- Other local variables declared in called routines in the region are private.

Fortran

2.8.2 threadprivate Directive

Summary

The **threadprivate** directive specifies that named global-lifetime objects are replicated, with each thread having its own copy.

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** object 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.

A thread may not reference another thread's copy of a **threadprivate** object.

1 During the sequential part, and in non-nested inactive **parallel** regions, references
2 will be to the initial thread's copy of the object. In **parallel** regions, references by
3 the master thread will be to the copy of the object in the thread which encountered the
4 **parallel** region.

5 The values of data in the initial thread's copy of a threadprivate object are guaranteed to
6 persist between any two consecutive references to the object in the program.

7 The values of data in the threadprivate objects of threads other than the initial thread are
8 guaranteed to persist between two consecutive active **parallel** regions only if *all* the
9 following conditions hold:

- 10 • Neither **parallel** region is nested inside another **parallel** region.
- 11 • The number of threads used to execute both **parallel** regions is the same.
- 12 • The value of the *dyn-var* internal control variable is *false* at entry to the first
13 **parallel** region and remains *false* until entry to the second **parallel** region.
- 14 • The value of the *nthreads-var* internal control variable is the same at entry to both
15 **parallel** regions and has not been modified between these points.

16 If these conditions all hold, and if a threadprivate object is referenced in both regions,
17 then threads with the same thread number in their respective regions will reference the
18 same copy of that variable.

19 C/C++

20 If the above conditions hold, the storage duration, lifetime, and value of a thread's copy
21 of a threadprivate variable that does not appear in any **copyin** clause on the second
22 region will be retained. Otherwise, the storage duration, lifetime, and value of a thread's
23 copy of the variable in the second region is undefined.

24 If an object is referenced in an explicit initializer of a threadprivate variable, and the
25 value of the object is modified prior to the first reference to a copy of the variable, then
26 the behavior is unspecified.

27 C/C++

28 Fortran

29 A variable is said to be affected by a **copyin** clause if the variable appears in the
30 **copyin** clause or it is in a common block that appears in the **copyin** clause.

31 If the above conditions hold, the definition, association, or allocation status of a thread's
32 copy of a threadprivate variable or a variable in a threadprivate common block, that is
33 not affected by any **copyin** clause that appears on the second region, will be retained.
34 Otherwise, the definition and association status of a thread's copy of the variable in the
35 second region is undefined, and the allocation status of an allocatable array will be
36 implementation defined.

1 If a common block, or a variable that is declared in the scope of a module, appears in a
2 **threadprivate** directive, it implicitly has the **SAVE** attribute.

3 If a **threadprivate** variable or a variable in a **threadprivate** common block is not affected
4 by any **copyin** clause that appears on the first **parallel** region in which it is
5 referenced, the variable or any subobject of the variable is initially defined or undefined
6 according to the following rules:

- 7 • If it has the **ALLOCATABLE** attribute, each copy created will have an initial
8 allocation status of not currently allocated.
- 9 • If it has the **POINTER** attribute:
 - 10 . if it has an initial association status of disassociated, either through explicit
11 initialization or default initialization, each copy created will have an association
12 status of disassociated;
 - 13 . otherwise, each copy created will have an association status of undefined.
- 14 • If it does not have either the **POINTER** or the **ALLOCATABLE** attribute:
 - 15 . if it is initially defined, either through explicit initialization or default
16 initialization, each copy created is so defined;
 - 17 . otherwise, each copy created is undefined.

Fortran

18 For examples of the **threadprivate** directive, see Section A.22 on page 158.

Restrictions

20 The restrictions to the **threadprivate** directive are as follows:

- 21 • A **threadprivate** object must not appear in any clause except the **copyin**,
22 **copyprivate**, **schedule**, **num_threads**, and **if** clauses.

C/C++

- 24 • A **threadprivate** directive for file-scope variables must appear outside any
25 definition or declaration, and must lexically precede all references to any of the
26 variables in its list.
- 27 • A **threadprivate** directive for namespace-scope variables must appear outside
28 any definition or declaration other than the namespace definition itself, and must
29 lexically precede all references to any of the variables in its list.
- 30 • Each variable in the list of a **threadprivate** directive at file or namespace scope
31 must refer to a variable declaration at file or namespace scope that lexically precedes
32 the directive.
- 33 • A **threadprivate** directive for static block-scope variables must appear in the
34 scope of the variable and not in a nested scope. The directive must lexically precede
35 all references to any of the variables in its list.
- 36

- 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.
- A threadprivate variable must not have an incomplete type or a reference type.
- A threadprivate variable with non-POD class type must have an accessible, unambiguous copy constructor if it is declared with an explicit initializer.

C/C++

Fortran

- 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.
- 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 be declared in an **EQUIVALENCE** statement.
- A variable that appears in a **threadprivate** directive and is not declared in the scope of a module must have the **SAVE** attribute.

Fortran

Cross References:

- Dynamic adjustment of threads, see Section 2.4.1 on page 29.
- **copyin** clause, see Section 2.8.4.1 on page 84.
- Internal control variables, see Section 2.3 on page 24.

2.8.3 Data-Sharing Attribute Clauses

Several constructs accept clauses that allow a user to control the sharing attributes of variables for the duration of the construct. Data-sharing attribute clauses apply only to variables whose names are visible in the construct on which the clause appears, except that formal arguments that are passed by reference inherit the data-sharing attributes of the associated actual argument.

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 18). 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/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 undefined.

C/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. This means that 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 (see Section A.23 on page 163 for examples). When individual members of a common block are privatized, the storage of the specified variables is no longer associated with the storage of the common block itself (see Section A.28 on page 171 for examples).

Fortran

2.8.3.1 default clause

Summary

The default clause allows the user to control the sharing attributes of variables which are referenced in a **parallel** construct, and whose sharing attributes are implicitly determined (see Section 2.8.1.1 on page 63).

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 | shared | none)
```

Fortran

Description

The **default(shared)** clause causes all variables referenced in the construct which have implicitly determined sharing attributes to be shared.

Fortran

The **default(private)** clause causes all variables referenced in the construct which have implicitly determined sharing attributes to be private.

Fortran

The **default(none)** clause requires that each variable which is referenced in the construct, and that does not have a predetermined sharing attribute, must have its sharing attribute explicitly determined by being listed in a data-sharing attribute clause. See Section A.24 on page 165 for examples.

Restrictions

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

- Only a single default clause may be specified on a **parallel** directive.

2.8.3.2 shared clause

Summary

The **shared** clause declares one or more list items to be shared among all the threads in a team.

Syntax

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

```
shared(list)
```

Description

All threads within a team access the same storage area for each shared object.

Fortran

The association status of a shared pointer becomes undefined upon entry to and on exit from the **parallel** 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 **parallel** 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. This situation will 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.

- 1 · An array section with a vector subscript.
- 2 · An assumed-shape array.
- 3 · A pointer array.
- 4 c. The associated dummy argument for this actual argument is an explicit-shape
- 5 array or an assumed-size array.

6 This effectively results in references to, and definitions of, the storage during the
7 procedure reference. Any references to (or definitions of) the shared storage that is
8 associated with the dummy argument by any other thread must be synchronized with the
9 procedure reference to avoid possible race conditions.

10 It is implementation defined whether this situation might occur under other conditions.
11 See Section A.25 on page 167 for an example of this behavior.

12  **Fortran**

13 **2.8.3.3 private clause**

14 **Summary**

15 The **private** clause declares one or more list items to be private to a thread.

16 **Syntax**

17 The syntax of the **private** clause is as follows:

```
18 private(list)
```

19 **Description**

20 Each thread in the team that references a list item that appears in a **private** clause in
21 any statement in the construct receives a new list item whose language-specific attributes
22 are derived from the original list item. Inside the construct, all references to the original
23 list item are replaced by references to the new list item. If a thread does not reference a
24 list item that appears in a **private** clause, it is unspecified whether that thread receives
25 a new list item.

26 The value of the original list item is not defined upon entry to the region. The original
27 list item must not be referenced within the region. The value of the original list item is
28 not defined upon exit from the region.

1 List items that are privatized in a **parallel** region may be privatized again in an
2 enclosed **parallel** or work-sharing construct. As a result, list items that appear in a
3 **private** clause on a **parallel** or work-sharing construct may be either shared or
4 private in the enclosing context. See Section A.27 on page 170 for an example.

5 C/C++

6 A new list item of the same type, with automatic storage duration, is allocated for the
7 construct. The size and alignment of the new list item are determined by the type of the
8 variable. This allocation occurs once for each thread in the team that references the list
9 item in any statement in the construct.

10 The new list item is initialized, or has an undefined initial value, as if it had been locally
11 declared without an initializer. The order in which any default constructors for different
12 private objects are called is unspecified.

13 C/C++

14 Fortran

15 A new list item of the same type is declared once for each thread in the team that
16 references the list item in any statement in the construct. The initial value of the new list
17 item is undefined. Within a **parallel** region, the initial status of a **private** pointer
18 is undefined.

19 A list item that appears in a **private** clause may be storage-associated with other
20 variables when the **private** clause is encountered. Storage association may exist
21 because of constructs such as **EQUIVALENCE**, **COMMON**, etc. If A is a variable
22 appearing in a **private** clause and B is a variable which is storage-associated with A ,
23 then:

- 24 • The contents, allocation, and association status of B are undefined on entry to the
25 **parallel** region.
- 26 • Any definition of A , or of its allocation or association status, causes the contents,
27 allocation, and association status of B to become undefined.
- 28 • Any definition of B , or of its allocation or association status, causes the contents,
29 allocation, and association status of A to become undefined.

30 For examples, see Section A.28 on page 171.

31 Fortran

32 For examples of the **private** clause, see Section A.26 on page 168.

33 Restrictions

34 The restrictions to the **private** clause are as follows:

- A list item that appears in the **reduction** clause of a **parallel** construct must not appear in a **private** clause on a work-sharing construct if any of the work-sharing regions arising from the work-sharing construct ever bind to any of the **parallel** regions arising from the **parallel** construct.

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.
- A variable that appears in a **private** clause must not have an incomplete type or a reference type.

C/C++

Fortran

- A variable that appears in a **private** clause must either be definable, or an allocatable array.
- An allocatable array that appears in a **private** clause must have an allocation status of “not currently allocated” on entry to and on exit from the construct.
- Assumed-size arrays may not appear in a **private** 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.

Fortran

2.8.3.4 **firstprivate** clause

Summary

The **firstprivate** clause declares one or more list items to be private to a thread, 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 2.8.3.3 on page 73. 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 thread in the team that references the list item in any statement in the construct. The initialization is done prior to the thread's execution of the construct.

For a **firstprivate** clause on a **parallel** construct, the initial value of the new list item is the value of the original list item that exists immediately prior to the **parallel** construct for the thread that encounters the construct. For a **firstprivate** clause on a work-sharing construct, the initial value of the new list item for a thread that executes the work-sharing construct is the value of the original list item that exists immediately prior to the point in time that the thread encounters the work-sharing construct.

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 as if by assignment. For a (possibly multi-dimensional) array of objects 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. For class types, a copy constructor is invoked to perform the initialization. The order in which copy constructors for different objects are called is unspecified.

C/C++

Fortran

The initialization of the new list items occurs as if by assignment.

Fortran

Restrictions

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

- A list item that is private within a **parallel** region, or that appears in the **reduction** clause of a **parallel** construct, must not appear in a **firstprivate** clause on a work-sharing construct if any of the work-sharing regions arising from the work-sharing construct ever bind to any of the **parallel** regions arising from the **parallel** construct.

C/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.
- A variable that appears in a **firstprivate** clause must not have a **const**-qualified type unless it is of class type with a **mutable** member.
- A variable that appears in a **firstprivate** clause must not have an incomplete type or a reference type.

C/C++

Fortran

- A variable that appears in a **firstprivate** clause must be definable.
- Fortran pointers, Cray pointers, assumed-size arrays and allocatable arrays may not appear in a **firstprivate** clause.
- 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

2.8.3.5 **lastprivate** clause

Summary

The **lastprivate** clause declares one or more list items to be private to a thread, 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.

1 A list item that appears in a **lastprivate** clause is subject to the **private** clause
2 semantics described in Section 2.8.3.3 on page 73. In addition, when a **lastprivate**
3 clause appears on the directive that identifies a work-sharing construct, the value of each
4 new list item from the sequentially last iteration of the associated loop, or the lexically
5 last **section** construct, is assigned to the original list item.

6 C/C++

7 For a (possibly multi-dimensional) array of objects of non-array type, each element is
8 assigned to the corresponding element of the original array.

9 C/C++

10 List items that are not assigned a value by the sequentially last iteration of the loop, or
11 by the lexically last **section** construct, have unspecified values after the construct.
12 Unassigned subobjects also have an unspecified value after the construct.

13 The original list item becomes defined at the end of the construct if there is an implicit
14 barrier at that point. Any concurrent uses or definitions of the original list item must be
15 synchronized with the definition that occurs at the end of the construct to avoid race
16 conditions.

17 If the **lastprivate** clause is used on a construct to which **nowait** is also applied,
18 the original list item remains undefined until a barrier synchronization has been
19 performed to ensure that the thread that executed the sequentially last iteration, or the
20 lexically last **section** construct, has stored that list item.

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

23 For an example of the **lastprivate** clause, see Section A.30 on page 175.

24 Restrictions

25 The restrictions to the **lastprivate** clause are as follows:

- 26 • A list item that is private within a **parallel** region, or that appears in the
27 **reduction** clause of a **parallel** construct, must not appear in a **lastprivate**
28 clause on a work-sharing construct if any of the corresponding work-sharing regions
29 ever binds to any of the corresponding **parallel** regions.

30 C/C++

- 31 • A variable of class type (or array thereof) that appears in a **lastprivate** clause
32 requires an accessible, unambiguous default constructor for the class type, unless the
33 list item is also specified in a **firstprivate** clause.
- 34 • A variable of class type (or array thereof) that appears in a **lastprivate** clause
35 requires an accessible, unambiguous copy assignment operator for the class type. The
36 order in which copy assignment operators for different objects are called is
37 unspecified.

- 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++

Fortran

- A variable that appears in a **lastprivate** clause must be definable.
- Fortran pointers, Cray pointers, assumed-size arrays and allocatable arrays may not appear in a **lastprivate** clause.
- 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

2.8.3.6 reduction clause

Summary

The **reduction** clause specifies an operator and one or more list items. For each list item, a private copy is created on each thread, and is initialized appropriately for the operator. After the end of the region, the original list item is updated with the values of the private copies using the specified operator.

Syntax

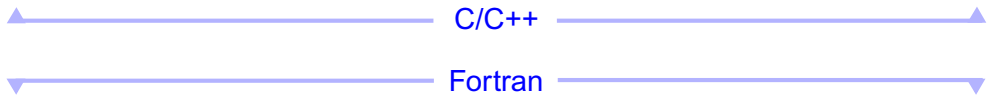
C/C++

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

```
reduction(operator:list)
```

The following table lists the *operators* that are valid and their initialization values. The actual initialization value depends on the data type of the reduction variable.

Operator	Initialization value
+	0
*	1
-	0
&	~0
	0
^	0
&&	1
	0



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

```
reduction( { operator | intrinsic_procedure_name } : list )
```

The following table lists the *operators* and *intrinsic_procedure_names* that are valid and their initialization values. The actual initialization value depends on the data type of the reduction variable.

Operator/ Intrinsic	Initialization value
+	0
*	1
-	0
.and.	.true.
.or.	.false.
.eqv.	.true.
.neqv.	.false.
max	Most negative representable number in the reduction variable type

min	Largest representable number in the reduction variable type
iand	All bits on
ior	0
ieor	0

Fortran

Description

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

A private copy of each list item is created, one for each thread, as if the **private** clause had been used. The private copy is then initialized to the initialization value for the operator, 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 operator specified. (The partial results of a subtraction reduction are added to form the final value.)

The value of the original list item becomes undefined when the first thread reaches the construct that specifies the clause and remains so until the reduction computation is complete. Normally, the 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, the value of the original list item remains undefined until a barrier synchronization has been performed to ensure that all threads have completed the reduction. Any concurrent uses or definitions of the original list item must be synchronized with the definition that occurs at the end of the construct, or at the subsequent barrier, to avoid race conditions.

The order in which the values are combined is unspecified. Therefore, comparing sequential and parallel runs, or 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.

Note – List items specified in a **reduction** clause are typically used in the enclosed region in certain forms.

C/C++

A reduction is typically specified for statements of the form:

```
x = x op expr
x binop= expr
x = expr op x      (except for subtraction)
x++
++x
x--
--x
```

where *expr* has scalar type and does not reference *x*, *op* is not an overloaded operator, but one of +, *, -, &, ^, |, &&, or | |, and *binop* is not an overloaded operator, but one of +, *, -, &, ^, or |.

C/C++

Fortran

A reduction using an operator is typically specified for statements of the form:

```
x = x op expr
x = expr op x      (except for subtraction)
```

where *op* is +, *, -, .and., .or., .eqv., or .neqv., the expression does not involve *x*, and the reduction *op* is the last operation performed on the right hand side.

A reduction using an intrinsic is typically specified for statements of the form:

```
x = intr(x, expr_list)
x = intr(expr_list, x)
```

where *intr* is **max**, **min**, **iand**, **ior**, or **ieor** and *expr_list* is a comma separated list of expressions not involving *x*.

Fortran

For examples, see Section A.31 on page 176.

Restrictions

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

- A list item that appears in a **reduction** clause of a work-sharing construct must be shared in the **parallel** regions to which any of the work-sharing regions arising from the work-sharing construct bind.
- A list item that appears in a **reduction** clause of a **parallel** construct must not be privatized on any enclosed work-sharing construct if any of the work-sharing regions arising from the work-sharing construct bind to any of the **parallel** regions arising from the **parallel** construct.
- Any number of **reduction** clauses can be specified on the directive, but a list item can appear only once in the **reduction** clause(s) for that directive.

C/C++

- The type of a list item that appears in a **reduction** clause must be valid for the reduction operator.
- Aggregate types (including arrays), pointer types and reference types may not appear in a **reduction** clause.
- A variable that appears in a **reduction** clause must not be **const**-qualified.
- The operator specified in a **reduction** clause cannot be overloaded with respect to the variables that appear in that clause.

C/C++

Fortran

- The type of a list item that appears in a **reduction** clause must be valid for the reduction operator or intrinsic.
- A variable that appears in a **reduction** clause must be definable.
- A list item that appears in a **reduction** clause must be a named variable of intrinsic type.
- Fortran pointers, Cray pointers, assumed-size arrays and allocatable arrays may not appear in a **reduction** clause.
- Operators specified must be intrinsic operators and any *intrinsic_procedure_name* must refer to one of the allowed intrinsic procedures. Assignment to the reduction variables must be via intrinsic assignment. See Section A.31 on page 176 for examples.

Fortran

2.8.4 Data Copying Clauses

This section describes the **copyin** clause (valid on the **parallel** directive and combined parallel work-sharing directives) and the **copyprivate** clause (valid on the **single** directive).

1 These clauses support the copying of data values from private or threadprivate objects
2 on one thread, to the corresponding objects on other threads in the team.

3 The clauses accept a comma-separated list of list items (see Section 2.1 on page 18). All
4 list items appearing in a clause must be visible, according to the scoping rules of the
5 base language. Clauses may be repeated as needed, but a list item that specifies a given
6 variable may not appear in more than one clause on the same directive.

7 2.8.4.1 `copyin` clause

8 Summary



9 The `copyin` clause provides a mechanism to copy the value of the master thread's
10 threadprivate variable to the threadprivate variable of each other member of the team
11 executing the `parallel` region.

12 Syntax



13 The syntax of the `copyin` clause is as follows:

```
14 copyin(list)
```

15 Description

16  **C/C++** 
17 The copy is done after the team is formed and prior to the start of execution of the
18 `parallel` region. For variables of non-array type, the copy occurs as if by assignment.
19 For a (possibly multi-dimensional) array of objects of non-array type, each element is
20 copied as if by assignment from an element of the master thread's array to the
21 corresponding element of the other thread's array. For class types, the copy assignment
22 operator is invoked. The order in which copy assignment operators for different objects
23 are called is unspecified.

24  **C/C++** 

25  **Fortran** 
26 The copy is done, as if by assignment, after the team is formed and prior to the start of
27 execution of the `parallel` region.

28 On entry to any `parallel` region, each thread's copy of a variable that is affected by
29 a `copyin` clause for the `parallel` region will acquire the allocation, association, and
30 definition status of the master thread's copy, according to the following rules:

- 31 • If it has the `POINTER` attribute:

- if the master thread's copy is associated with a target that each copy can become associated with, each copy will become associated with the same target;
- if the master thread's copy is disassociated, each copy will become disassociated;
- otherwise, each copy will have an undefined association status.
- If it does not have the **POINTER** attribute, each copy becomes defined with the value of the master thread's copy as if by intrinsic assignment.

Fortran

For an example of the **copyin** clause, see Section A.32 on page 180.

Restrictions

The restrictions on 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.
- Allocatable arrays may not appear in a **copyin** clause.

Fortran

2.8.4.2 **copyprivate** clause

Summary

The **copyprivate** clause provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members of the team.

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 2.5.3 on page 42), and before any of the threads in the team have left the barrier at the end of the construct.

C/C++

In all other threads in the team, each specified list item becomes defined with the value of the corresponding list item in the thread that executed the structured block. For variables of non-array type, the definition occurs as if by copy assignment. For a (possibly multi-dimensional) array of objects of non-array type, each element is copied as if by copy assignment from an element of the array in the thread that executed the structured block to the corresponding element of the array in the other threads. For class types, a copy assignment operator is invoked. The order in which copy assignment operators for different objects are called is unspecified.

C/C++

Fortran

If a list item is not a pointer, then in all other threads in the team, the list item becomes defined (as if by assignment) with the value of the corresponding list item in the thread that executed the structured block. If the list item is a pointer, then in all other threads in the team, the list item becomes pointer associated (as if by pointer assignment) with the corresponding list item in the thread that executed the structured block.

Fortran

For examples of the `copyprivate` clause, see Section A.33 on page 181.

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/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/C++

Fortran

- A common block that appears in a **copyprivate** clause must be **threadprivate**.
- Allocatable arrays and assumed-size arrays may not appear in a **copyprivate** clause.

Fortran

2.9 Nesting of Regions

This section describes a set of restrictions on the nesting of regions. The restrictions on nesting are as follows:

- A work-sharing region may not be closely nested inside a work-sharing, **critical**, **ordered**, or **master** region.
- A **barrier** region may not be closely nested inside a work-sharing, **critical**, **ordered**, or **master** region.
- A **master** region may not be closely nested inside a work-sharing region.
- An **ordered** region may not be closely nested inside a **critical** 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.

For examples illustrating these rules, see Section A.14 on page 139, Section A.34 on page 185 and Section A.35 on page 187.

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 90).
- Execution environment routines that can be used to control and query the parallel execution environment (Section 3.2 on page 91).
- Lock routines that can be used to synchronize access to data (Section 3.3 on page 102).
- Portable timer routines (Section 3.4 on page 108).

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* and *nestable lock* 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`.

See Section D.1 on page 223 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 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 21).

See Section D.2 on page 225 and Section D.3 on page 227 for examples of these files.

1 It is implementation defined whether any of the OpenMP runtime library routines that
2 take an argument are extended with a generic interface so arguments of different **KIND**
3 type can be accommodated. See Appendix D.4 for an example of such an extension.
4

Fortran

5 3.2 Execution Environment Routines

6 The routines described in this section affect and monitor threads, processors, and the
7 parallel environment.

- 8 • the `omp_set_num_threads` routine.
- 9 • the `omp_get_num_threads` routine.
- 10 • the `omp_get_max_threads` routine.
- 11 • the `omp_get_thread_num` routine.
- 12 • the `omp_get_num_procs` routine.
- 13 • the `omp_in_parallel` routine.
- 14 • the `omp_set_dynamic` routine.
- 15 • the `omp_get_dynamic` routine.
- 16 • the `omp_set_nested` routine.
- 17 • the `omp_get_nested` routine.

18 3.2.1 `omp_set_num_threads`

19 Summary

20 The `omp_set_num_threads` routine affects the number of threads to be used for
21 subsequent `parallel` regions that do not specify a `num_threads` clause, by setting
22 the value of the `nthreads-var` internal control variable.

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.

Binding

When called from the sequential part of the program, the binding thread set for an **omp_set_num_threads** 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_num_threads** region is implementation defined.

Effect

The effect of this routine is to set the value of the *nthreads-var* internal control variable to the value specified in the argument.

See Section 2.4.1 on page 29 for the rules governing the number of threads used to execute a **parallel** region.

If the number of threads requested exceeds the number the implementation can support, or is not a positive integer, the behavior of this routine is implementation defined.

For an example of the **omp_set_num_threads** routine, see Section A.36 on page 193.

Calling Context Rules

This routine has the described effect only when called from the sequential part of the program. If it is called from any **parallel** region, the behavior of this routine is implementation defined.

Cross References

- Internal control variables, see Section 2.3 on page 24.

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 thread set for an `omp_get_num_threads` region is the current team. The binding region for an `omp_get_num_threads` region is the innermost enclosing `parallel` region. The return value of this routine depends on the characteristics of the team executing the binding `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. For examples, see Section A.37 on page 195.

See Section 2.4.1 on page 29 for the rules governing the number of threads used to execute a `parallel` region.

Cross References

- `parallel` construct, see Section 2.4 on page 26.

3.2.3 `omp_get_max_threads`

Summary

The `omp_get_max_threads` routine returns the value of the `nthreads-var` internal control variable, which is used to determine the number of threads that would form the new team, if an active `parallel` region without a `num_threads` clause were to be encountered at that point in the program.

Format

C/C++

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

C/C++

Fortran

```
integer function omp_get_max_threads()
```

Fortran

Binding

When called from the sequential part of the program, the binding thread set for an `omp_get_max_threads` 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_threads` region is implementation defined.

Effect

The following expresses a lower bound on the value of `omp_get_max_threads`: the number of threads that would be used to form a team if an active `parallel` region without a `num_threads` clause were to be encountered at that point in the program is less than or equal to the value returned by `omp_get_max_threads`.

See Section 2.4.1 on page 29 for the rules governing the number of threads used to execute a `parallel` region.

Note – The return value of `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

- Internal control variables, see Section 2.3 on page 24.
- `parallel` construct, see Section 2.4 on page 26.
- `num_threads` clause, see Section 2.4 on page 26.

3.2.4 `omp_get_thread_num`

Summary

The `omp_get_thread_num` routine returns the thread number, within the team, of the thread executing the `parallel` region from which `omp_get_thread_num` is called.

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. The return value of this routine depends on the characteristics of the team executing the binding `parallel` region.

Effect

The `omp_get_thread_num` routine returns the thread number of the current 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.

Cross References

- `omp_get_num_threads` routine, see Section 3.2.2 on page 93.

3.2.5 `omp_get_num_procs`

Summary

The `omp_get_num_procs` routine returns the number of processors available to the program.

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. 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 program at the time the routine is called.

3.2.6 `omp_in_parallel`

Summary

The `omp_in_parallel` routine returns *true* if the call to the routine is enclosed by an active `parallel` region; otherwise, it returns *false*.

Format

C/C++

```
int omp_in_parallel(void);
```

C/C++

Fortran

```
logical function omp_in_parallel()
```

Fortran

Binding

The binding thread set for an `omp_in_parallel` region is all threads. The effect of executing this routine is not related to any specific `parallel` region but instead depends on the state of all enclosing `parallel` regions.

Effect

`omp_in_parallel` returns the logical OR of the `if` clauses of all enclosing `parallel` regions. If a `parallel` region does not have an `if` clause, this is equivalent to `if(true)`.

If the routine is called from the sequential part of the program, then `omp_in_parallel` returns *false*.

Cross References

- `if` clause, see Section 2.4.1 on page 29.

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 `parallel` regions by setting the value of the *dyn-var* internal control variable.

Format

C/C++

```
void omp_set_dynamic(int dynamic_threads);
```

C/C++

Fortran

```
subroutine omp_set_dynamic (dynamic_threads)  
  logical dynamic_threads
```

Fortran

Binding

When called from the sequential part of the program, the binding thread set for an **omp_set_dynamic** 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_dynamic** region is implementation defined.

Effect

For implementations that provide the ability to dynamically adjust the number of threads, if the argument to **omp_set_dynamic** evaluates to *true*, dynamic adjustment of the number of threads is enabled; otherwise, dynamic adjustment is disabled.

For implementations that do not provide the ability to dynamically adjust the number of threads, this routine has no effect: the value of *dyn-var* remains *false*.

For an example of the **omp_set_dynamic** routine, see Section A.36 on page 193.

See Section 2.4.1 on page 29 for the rules governing the number of threads used to execute a **parallel** region.

Calling Context Rules

The **omp_set_dynamic** routine has the described effect only when called from the sequential part of the program. If called from within any explicit **parallel** region, the behavior of this routine is implementation defined.

Cross References:

- Internal control variables, see Section 2.3 on page 24.
- **omp_get_num_threads** routine, see Section 3.2.2 on page 93.

3.2.8 `omp_get_dynamic`

Summary

The `omp_get_dynamic` routine returns the value of the *dyn-var* internal control variable, 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

When called from the sequential part of the program, the binding thread set for an `omp_get_dynamic` 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_dynamic` region is implementation defined.

Effect

This routine returns *true* if dynamic adjustment of the number of threads is enabled; it returns *false*, otherwise.

If the implementation does not provide the ability to dynamically adjust the number of threads, then this routine always returns *false*.

See Section 2.4.1 on page 29 for the rules governing the number of threads used to execute a `parallel` region.

Cross References

- Internal control variables, see Section 2.3 on page 24.

3.2.9 `omp_set_nested`

Summary

The `omp_set_nested` routine enables or disables nested parallelism, by setting the *nest-var* internal control variable.

Format

C/C++

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

C/C++

Fortran

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

Fortran

Binding

When called from the sequential part of the program, the binding thread set for an `omp_set_nested` 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_nested` region is implementation defined.

Effect

For implementations that support nested parallelism, if the argument to `omp_set_nested` evaluates to *true*, nested parallelism is enabled; otherwise, nested parallelism is disabled.

For implementations that do not support nested parallelism, this routine has no effect: the value of *nest-var* remains *false*.

See Section 2.4.1 on page 29 for the rules governing the number of threads used to execute a `parallel` region.

Calling Context Rules

The `omp_set_nested` routine has the described effect only when called from the sequential part of the program. If called from within any explicit `parallel` region, the behavior of this routine is implementation defined.

Cross References

- Internal control variables, see Section 2.3 on page 24.

3.2.10 `omp_get_nested`

Summary

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

Format

C/C++

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

C/C++

Fortran

```
logical function omp_get_nested()
```

Fortran

Binding

When called from the sequential part of the program, the binding thread set for an `omp_get_nested` 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_nested` region is implementation defined.

Effect

This routine returns *true* if nested parallelism is enabled; it returns *false*, otherwise.

If an implementation does not support nested parallelism, this routine always returns *false*.

See Section 2.4.1 on page 29 for the rules governing the number of threads used to execute a `parallel` region.

Cross References

- Internal control variables, see Section 2.3 on page 24.

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. An OpenMP lock variable must be accessed only through the routines described in this section.

An OpenMP lock may be in one of the following states: *uninitialized*, *unlocked*, or *locked*. If a lock is in the unlocked state, a thread may *set* the lock, which changes its state to locked. The thread which sets the lock is then said to *own* the lock. A thread which owns a lock may *unset* that lock, returning it to the unlocked state. A thread may not set or unset a lock which is owned by another thread.

Two types of locks are supported: *simple locks* and *nestable locks*. A nestable lock may be set multiple times by the same thread before being unset; a simple lock may not be set if it is already owned by the thread trying to set it. Simple lock variables are associated with simple locks and may only be passed to simple lock routines. Nestable lock variables are associated with nestable locks and may 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. Therefore, 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 threads.

See Section A.39 on page 198 and Section A.40 on page 200, for examples of using the simple and the nestable lock routines, respectively.

Binding

The binding thread set for all lock routine regions is all threads. For each OpenMP lock, the lock routine effects relate to all threads which execute the routines, without regard to which team(s) the threads belong.

Simple Lock Routines

C/C++

The type `omp_lock_t` is an object type capable of representing a simple lock. For the following routines, a 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, *svar* 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 uninitialized 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 an object type capable of representing a nestable lock. For the following routines, a 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, *nvar* 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 uninitialized 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.

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 lock accessed by either routine must be in the uninitialized state.

Effect

The effect of these routines is to initialize the lock to the unlocked state (that is, no thread owns the lock). In addition, the nesting count for a nestable lock is set to zero.

For an example of the `omp_init_lock` routine, see Section A.38 on page 197.

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 lock accessed by either routine must be in the unlocked state.

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 thread blocks 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 lock accessed by either routine must not be in the uninitialized state. A simple lock accessed by `omp_set_lock` which is in the locked state must not be owned by the thread executing the routine.

Effect

Each of these routines blocks the thread 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 thread executing the routine.

A nestable lock is available if it is unlocked or if it is already owned by the thread executing the routine. The thread 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 lock accessed by either routine must be in the locked state and owned by the thread executing the routine.

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 threads are waiting for this lock, the effect is that one thread 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 block execution of the thread 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 lock accessed by either routine must not be in the uninitialized state. A simple lock accessed by `omp_test_lock` which is in the locked state must not be owned by the thread executing the routine.

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 block execution of the thread 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

The routines described in this section support a portable wall clock timer.

- the `omp_get_wtime` routine.
- the `omp_get_wtick` routine.

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 times returned are “per-thread times”, so they are not required to be globally consistent across all the threads participating in an application.

1 **Note** – It is anticipated that the routine will be used to measure elapsed times as shown
2 in the following example:

3 C/C++

```
4 double start;  
5 double end;  
6 start = omp_get_wtime();  
7 ... work to be timed ...  
8 end = omp_get_wtime();  
9 printf("Work took %f seconds\n", end - start);
```

10 C/C++

11 Fortran

```
12 DOUBLE PRECISION START, END  
13 START = omp_get_wtime()  
14 ... work to be timed ...  
15 END = omp_get_wtime()  
16 PRINT *, "Work took", END - START, "seconds"
```

17 Fortran

18 3.4.2 omp_get_wtick

19 Summary

20 The `omp_get_wtick` routine returns the precision of the timer used by
21 `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`.

Environment Variables

This chapter describes the OpenMP environment variables that specify the settings of the internal control variables that affect the execution of OpenMP programs (see Section 2.3 on page 24). The names of the environment variables must be uppercase. 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 the internal control variables 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* internal control variable for the runtime schedule type and chunk size.
- **OMP_NUM_THREADS** sets the *nthreads-var* internal control variable for the number of threads to use for **parallel** regions.
- **OMP_DYNAMIC** sets the *dyn-var* internal control variable for the dynamic adjustment of threads to use for **parallel** regions.
- **OMP_NESTED** sets the *nest-var* internal control variable to enable or disable nested parallelism.

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* internal control variable.

The value of this environment variable takes the form:

type[,*chunk*]

where

- *type* is one of **static**, **dynamic** or **guided**
- *chunk* is an optional positive integer which specifies the chunk size

If *chunk* is present, there may be white space on either side of the “,”. See Section 2.5.1 on page 33 for a detailed description of the schedule types.

If **OMP_SCHEDULE** is not set, the initial value of the *run-sched-var* internal control variable is implementation defined.

Example:

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

Cross References:

- Internal control variables, see Section 2.3 on page 24.
- Loop construct, see Section 2.5.1 on page 33.
- Parallel loop construct, see Section 2.6.1 on page 47.

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` internal control variable. See Section 2.3 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.

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_NUM_THREADS` is greater than the number of threads an implementation can support, or if the value is not a positive integer.

If the `OMP_NUM_THREADS` environment variable is not set, the initial value of the `nthreads-var` internal control variable is implementation defined.

The `nthreads-var` internal control variable can be modified using the `omp_set_num_threads` library routine. The number of threads in the current team can be queried using the `omp_get_num_threads` library routine. The maximum number of threads in future teams can be queried using the `omp_get_max_threads` library routine.

Example:

```
setenv OMP_NUM_THREADS 16
```

Cross References:

- Internal control variables, see Section 2.3 on page 24.
- `num_threads` clause, Section 2.4 on page 26.
- `omp_set_num_threads` routine, see Section 3.2.1 on page 91.
- `omp_get_num_threads` routine, see Section 3.2.2 on page 93.
- `omp_get_max_threads` routine, see Section 3.2.3 on page 94.
- `omp_get_dynamic` routine, see Section 3.2.8 on page 99.

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* internal control variable. 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.

If the `OMP_DYNAMIC` environment variable is not set, the initial value of the *dyn-var* internal control variable is implementation defined.

The *dyn-var* internal control variable can be modified by calling the `omp_set_dynamic` library routine. The current value of *dyn-var* can be queried using the `omp_get_dynamic` library routine.

Example:

```
setenv OMP_DYNAMIC true
```

Cross References:

- Internal control variables, see Section 2.3 on page 24.
- `omp_get_num_threads` routine, see Section 3.2.2 on page 93.
- `omp_set_dynamic` routine, see Section 3.2.7 on page 97.
- `omp_get_dynamic` routine, see Section 3.2.8 on page 99.

4.4 OMP_NESTED

The `OMP_NESTED` environment variable controls nested parallelism by setting the initial value of the *nest-var* internal control variable. 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.

If the `OMP_NESTED` environment variable is not set, the initial value of the *nest-var* internal control variable is `false`.

1 The *nest-var* internal control variable can be modified by calling the
2 **omp_set_nested** library routine. The current value of *nest-var* can be queried using
3 the **omp_get_nested** library routine.

4 Example:

```
5 setenv OMP_NESTED false
```

6 **Cross References:**

- 7 • Internal control variables, see Section 2.3 on page 24.
- 8 • **omp_set_nested** routine, see Section 3.2.9 on page 100.

Examples

The following are examples of the constructs and routines defined in this document.

C/C++

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

C/C++

A.1 A Simple Parallel Loop

The following example demonstrates how to parallelize a simple loop using the parallel loop construct (Section 2.6.1 on page 47). The loop iteration variable is private by default, so it is not necessary to specify it explicitly in a **private** clause.

C/C++

Example A.1.1c

```
void a1(int n, float *a, float *b)
{
    int i;

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

C/C++

Fortran

Example A.1.1f

```
SUBROUTINE A1(N, A, B)
```

```

1      INTEGER I, N
2      REAL B(N), A(N)

3      !$OMP PARALLEL DO !I is private by default
4          DO I=2,N
5              B(I) = (A(I) + A(I-1)) / 2.0
6          ENDDO
7      !$OMP END PARALLEL DO

8      END SUBROUTINE A1

```

▶ Fortran ◀

A.2 The OpenMP Memory Model

In the following example, at Print 1, the value of x could be either 2 or 5, depending on the timing of the threads, and the implementation of the assignment to x . There are two reasons that the value at Print 1 might not be 5. First, Print 1 might be executed before the assignment to x is executed. Second, even if Print 1 is executed after the assignment, the value 5 is not guaranteed to be seen by thread 1 because a flush may not have been executed by thread 0 since the assignment.

The barrier after Print 1 contains implicit flushes on all threads, as well as a thread synchronization, so the programmer is guaranteed that the value 5 will be printed by both Print 2 and Print 3.

▶ C/C++ ◀

Example A.2.1c

```

22 #include <stdio.h>
23 #include <omp.h>

24 int main(){
25     int x;

26     x = 2;
27     #pragma omp parallel num_threads(2) shared(x)
28     {

29         if (omp_get_thread_num() == 0) {
30             x = 5;
31         } else {
32             /* Print 1: the following read of x has a race */
33             printf("1: Thread# %d: x = %d\n", omp_get_thread_num(), x );
34         }

```



```

1      #pragma omp barrier
2
3      if (omp_get_thread_num() == 0) {
4          /* Print 2 */
5          printf("2: Thread# %d: x = %d\n", omp_get_thread_num(),x );
6      } else {
7          /* Print 3 */
8          printf("3: Thread# %d: x = %d\n", omp_get_thread_num(),x );
9      }
10     }
11     return 0;
12 }

```

▲ C/C++ ▲

▼ Fortran ▼

Example A.2.1f

```

15 PROGRAM A2
16     INCLUDE "omp_lib.h"      ! or USE OMP_LIB
17     INTEGER X
18
19     X = 2
20     !$OMP PARALLEL NUM_THREADS(2) SHARED(X)
21
22     IF (OMP_GET_THREAD_NUM() .EQ. 0) THEN
23         X = 5
24     ELSE
25         ! PRINT 1: The following read of x has a race
26         PRINT *, "1: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
27     ENDIF
28
29     !$OMP BARRIER
30
31     IF (OMP_GET_THREAD_NUM() .EQ. 0) THEN
32         ! PRINT 2
33         PRINT *, "2: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
34     ELSE
35         ! PRINT 3
36         PRINT *, "3: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
37     ENDIF
38
39     !$OMP END PARALLEL
40
41     END PROGRAM A2

```

▲ Fortran ▲

A.3 Conditional Compilation

C/C++

The following example illustrates the use of conditional compilation using the OpenMP macro `_OPENMP` (Section 2.2 on page 21). With OpenMP compilation, the `_OPENMP` macro becomes defined.

Example A.3.1c

```
#include <stdio.h>

int main()
{
    # ifdef _OPENMP
        printf("Compiled by an OpenMP-compliant implementation.\n");
    # endif

    return 0;
}
```

C/C++

Fortran

The following example illustrates the use of the conditional compilation sentinel (see Section 2.2 on page 21). With OpenMP compilation, the conditional compilation sentinel `!$` is recognized and treated as two spaces. In fixed form source, statements guarded by the sentinel must start after column 6.

Example A.3.1f

```
PROGRAM A3

C234567890
!$ PRINT *, "Compiled by an OpenMP-compliant implementation."

END PROGRAM A3
```

Fortran

A.4 The parallel Construct

The `parallel` construct (Section 2.4 on page 26) can be used in coarse-grain parallel programs. In the following example, each thread in the `parallel` region decides what part of the global array `x` to work on, based on the thread number:

C/C++

Example A.4.1c

```
#include <omp.h>

void subdomain(float *x, int  istart, int  ipoints)
{
    int i;

    for (i = 0; i < ipoints; i++)
        x[istart+i] = 123.456;
}

void sub(float *x, int  npoints)
{
    int iam, nt, ipoints, istart;

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

int main()
{
    float array[10000];

    sub(array, 10000);

    return 0;
}
```

C/C++

Example A.4.1f

```

1      SUBROUTINE SUBDOMAIN(X, ISTART, IPOINITS)
2          INTEGER ISTART, IPOINITS
3          REAL X(*)
4
5          INTEGER I
6
7          DO 100 I=1,IPOINITS
8              X(ISTART+I) = 123.456
9      100    CONTINUE
10
11     END SUBROUTINE SUBDOMAIN
12
13     SUBROUTINE SUB(X, NPOINITS)
14         INCLUDE "omp_lib.h"      ! or USE OMP_LIB
15
16         REAL X(*)
17         INTEGER NPOINITS
18
19         INTEGER IAM, NT, IPOINITS, ISTART
20
21     !$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,NPOINITS)
22
23         IAM = OMP_GET_THREAD_NUM()
24         NT = OMP_GET_NUM_THREADS()
25         IPOINITS = NPOINITS/NT
26         ISTART = IAM * IPOINITS
27         IF (IAM .EQ. NT-1) THEN
28             IPOINITS = NPOINITS - ISTART
29         ENDIF
30         CALL SUBDOMAIN(X,ISTART,IPOINITS)
31
32     !$OMP END PARALLEL
33
34     END SUBROUTINE SUB
35
36     PROGRAM A4
37
38         REAL ARRAY(10000)
39
40         CALL SUB(ARRAY, 10000)
41
42     END PROGRAM A4

```

A.5 The `num_threads` Clause

The following example demonstrates the `num_threads` clause (Section 2.4 on page 26). The parallel region is executed with a maximum of 10 threads.

Example A.5.1c C/C++

```
#include <omp.h>
int main()
{
    omp_set_dynamic(1);

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

Example A.5.1f Fortran

```
PROGRAM A5
    INCLUDE "omp_lib.h"          ! or USE OMP_LIB
    CALL OMP_SET_DYNAMIC(.TRUE.)

    !$OMP    PARALLEL NUM_THREADS(10)
             ! do work here
    !$OMP    END PARALLEL
END PROGRAM A5
```

Fortran

Fortran

A.6 Fortran Restrictions on the `do` Construct

If an `end do` directive follows a *do-construct* in which several `DO` statements share a `DO` termination statement, then a `do` directive can only be specified for the first (i.e. outermost) of these `DO` statements. For more information, see Section 2.5.1 on page 33. The following example contains correct usages of loop constructs:

Example A.6.1f

```

3      SUBROUTINE WORK(I, J)
4      INTEGER I,J
5      END SUBROUTINE WORK

6      SUBROUTINE A6_GOOD()
7      INTEGER I, J
8      REAL A(1000)

9      DO 100 I = 1,10
10     !$OMP DO
11         DO 100 J = 1,10
12             CALL WORK(I,J)
13     100    CONTINUE      ! !$OMP ENDDO implied here

14     !$OMP DO
15         DO 200 J = 1,10
16     200    A(I) = I + 1
17     !$OMP ENDDO

18     !$OMP DO
19         DO 300 I = 1,10
20             DO 300 J = 1,10
21                 CALL WORK(I,J)
22     300    CONTINUE
23     !$OMP ENDDO
24     END SUBROUTINE A6_GOOD

```

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

Example A.6.2f

```

28     SUBROUTINE WORK(I, J)
29     INTEGER I,J
30     END SUBROUTINE WORK

31     SUBROUTINE A6_WRONG
32     INTEGER I, J

33     DO 100 I = 1,10
34     !$OMP DO
35         DO 100 J = 1,10
36             CALL WORK(I,J)
37     100    CONTINUE

```

```
1      !$OMP   ENDDO
2          END SUBROUTINE A6_WRONG
```

3  Fortran 

4  Fortran 

5 A.7 Fortran Private Loop Iteration Variables

6 In general loop iteration variables will be private, when used in the *do-loop* of a **do** and
7 **parallel do** construct or in sequential loops in a **parallel** construct (see
8 Section 2.5.1 on page 33 and Section 2.8.1 on page 63). In the following example of a
9 sequential loop in a **parallel** construct the loop iteration variable *I* will be private.

10 *Example A.7.1f*

```
11 SUBROUTINE A7_1(A,N)
12 INCLUDE "omp_lib.h"      ! or USE OMP_LIB
13
14 REAL A(*)
15 INTEGER I, MYOFFSET, N
16
17 !$OMP PARALLEL PRIVATE(MYOFFSET)
18     MYOFFSET = OMP_GET_THREAD_NUM()*N
19     DO I = 1, N
20         A(MYOFFSET+I) = FLOAT(I)
21     ENDDO
22 !$OMP END PARALLEL
23
24 END SUBROUTINE A7_1
```

22 In exceptional cases, loop iteration variables can be made shared, as in the following
23 example:

24 *Example A.7.2f*

```
25 SUBROUTINE A7_2(A,B,N,I1,I2)
26 REAL A(*), B(*)
27 INTEGER I1, I2, N
28
29 !$OMP PARALLEL SHARED(A,B,I1,I2)
30 !$OMP SECTIONS
31 !$OMP SECTION
32     DO I1 = I1, N
```

```

1         IF (A(I1).NE.0.0) EXIT
2         ENDDO
3     !$OMP SECTION
4         DO I2 = I2, N
5             IF (B(I2).NE.0.0) EXIT
6             ENDDO
7     !$OMP END SECTIONS
8     !$OMP SINGLE
9         IF (I1.LE.N) PRINT *, 'ITEMS IN A UP TO ', I1, ' ARE ALL ZERO.'
10        IF (I2.LE.N) PRINT *, 'ITEMS IN B UP TO ', I2, ' ARE ALL ZERO.'
11    !$OMP END SINGLE
12    !$OMP END PARALLEL

13    END SUBROUTINE A7_2

```

Note however that the use of shared loop iteration variables can easily lead to race conditions.

Fortran

A.8 The `nowait` clause

If there are multiple independent loops within a `parallel` region, you can use the `nowait` clause (see Section 2.5.1 on page 33) to avoid the implied barrier at the end of the loop construct, as follows:

C/C++

Example A.8.1c

```

23    #include <math.h>

24    void a8(int n, int m, float *a, float *b, float *y, float *z)
25    {
26        int i;
27        #pragma omp parallel
28        {
29            #pragma omp for nowait
30            for (i=1; i<n; i++)
31                b[i] = (a[i] + a[i-1]) / 2.0;

32            #pragma omp for nowait
33            for (i=0; i<m; i++)
34                y[i] = sqrt(z[i]);
35        }
36    }

```

C/C++

Example A.8.1f

```

SUBROUTINE A8(N, M, A, B, Y, Z)

    INTEGER N, M
    REAL A(*), B(*), Y(*), Z(*)

    INTEGER I

!$OMP PARALLEL

!$OMP DO
    DO I=2,N
        B(I) = (A(I) + A(I-1)) / 2.0
    ENDDO
!$OMP END DO NOWAIT

!$OMP DO
    DO I=1,M
        Y(I) = SQRT(Z(I))
    ENDDO
!$OMP END DO NOWAIT

!$OMP END PARALLEL

    END SUBROUTINE A8

```

A.9 The parallel sections Construct

In the following example (for Section 2.5.2 on page 39) routines *xaxis*, *yaxis*, and *zaxis* can be executed concurrently. The first **section** directive is optional. Note that all **section** directives need to appear in the **parallel sections** construct.

Example A.9.1c

```

void XAXIS();
void YAXIS();
void ZAXIS();

void a9()
{

```

```

1      #pragma omp parallel sections
2      {
3          #pragma omp section
4              XAXIS();

5          #pragma omp section
6              YAXIS();

7          #pragma omp section
8              ZAXIS();
9      }
10     }

```

C/C++

Fortran

Example A.9.1f

```

14         SUBROUTINE A9()

15         !$OMP PARALLEL SECTIONS
16         !$OMP SECTION
17             CALL XAXIS()

18         !$OMP SECTION
19             CALL YAXIS()

20         !$OMP SECTION
21             CALL ZAXIS()

22         !$OMP END PARALLEL SECTIONS

23         END SUBROUTINE A9

```

Fortran

A.10 The `single` Construct

The following example demonstrates the `single` construct (Section 2.5.3 on page 42). In the example, only one thread prints each of the progress messages. All other threads will skip the `single` region and stop at the barrier at the end of the `single` construct until all threads in the team have reached the barrier. If other threads can proceed without waiting for the thread executing the `single` region, a `nowait` clause can be specified, as is done in the third `single` construct in this example. The user must not make any assumptions as to which thread will execute a `single` region.

Example A.10.1c

```

1  #include <stdio.h>
2
3  void work1() {}
4  void work2() {}
5
6  void a10()
7  {
8      #pragma omp parallel
9      {
10         #pragma omp single
11         printf("Beginning work1.\n");
12
13         work1();
14
15         #pragma omp single
16         printf("Finishing work1.\n");
17
18         #pragma omp single nowait
19         printf("Finished work1 and beginning work2.\n");
20
21         work2();
22     }
23 }

```

Example A.10.1f

```

23     SUBROUTINE WORK1()
24     END SUBROUTINE WORK1
25
26     SUBROUTINE WORK2()
27     END SUBROUTINE WORK2
28
29     PROGRAM A10
30     !$OMP PARALLEL
31
32     !$OMP SINGLE
33     print *, "Beginning work1."
34     !$OMP END SINGLE
35
36     CALL WORK1()
37
38     !$OMP SINGLE
39     print *, "Finishing work1."
40     !$OMP END SINGLE

```

```

1      !$OMP SINGLE
2          print *, "Finished work1 and beginning work2."
3      !$OMP END SINGLE NOWAIT
4
5          CALL WORK2()
6
7      !$OMP END PARALLEL
8
9      END PROGRAM A10

```



A.11 The workshare Construct

The following are examples of the **workshare** construct (see Section 2.5.4 on page 44).

In the following example, **workshare** spreads work across the threads executing the **parallel** region, and there is a barrier after the last statement. Implementations must enforce Fortran execution rules inside of the **workshare** block.

Example A.11.1f

```

15      SUBROUTINE A11_1(AA, BB, CC, DD, EE, FF, N)
16          INTEGER N
17          REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N), EE(N,N), FF(N,N)
18
19      !$OMP    PARALLEL
20      !$OMP    WORKSHARE
21          AA = BB
22          CC = DD
23          EE = FF
24      !$OMP    END WORKSHARE
25      !$OMP    END PARALLEL
26
27      END SUBROUTINE A11_1

```

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

Example A.11.2f

```

1          SUBROUTINE A11_2(AA, BB, CC, DD, EE, FF, N)
2          INTEGER N
3          REAL AA(N,N), BB(N,N), CC(N,N)
4          REAL DD(N,N), EE(N,N), FF(N,N)
5
6
7          !$OMP   PARALLEL
8          !$OMP   WORKSHARE
9              AA = BB
10             CC = DD
11          !$OMP   END WORKSHARE NOWAIT
12          !$OMP   WORKSHARE
13             EE = FF
14          !$OMP   END WORKSHARE
15          !$OMP   END PARALLEL
16          END SUBROUTINE A11_2

```

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

Example A.11.3f

```

19          SUBROUTINE A11_3(AA, BB, CC, DD, N)
20          INTEGER N
21          REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
22          REAL R
23
24             R=0
25          !$OMP   PARALLEL
26          !$OMP   WORKSHARE
27             AA = BB
28          !$OMP   ATOMIC
29             R = R + SUM(AA)
30             CC = DD
31          !$OMP   END WORKSHARE
32          !$OMP   END PARALLEL
33
34          END SUBROUTINE A11_3

```

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

Each task gets worked on in order by the threads:

AA = BB then

CC = DD then

EE .ne. 0 then

FF = 1 / EE then

GG = HH

Example A.11.4f

```

SUBROUTINE A11_4(AA, BB, CC, DD, EE, FF, GG, HH, N)
  INTEGER N
  REAL AA(N,N), BB(N,N), CC(N,N)
  REAL DD(N,N), EE(N,N), FF(N,N)
  REAL GG(N,N), HH(N,N)

  !$OMP PARALLEL
  !$OMP WORKSHARE
      AA = BB
      CC = DD
      WHERE (EE .ne. 0) FF = 1 / EE
      GG = HH
  !$OMP END WORKSHARE
  !$OMP END PARALLEL

  END SUBROUTINE A11_4

```

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

Example A.11.5f

```

SUBROUTINE A11_5(AA, BB, CC, DD, N)
  INTEGER N
  REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)

  INTEGER SHR

  !$OMP PARALLEL SHARED(SHR)
  !$OMP WORKSHARE
      AA = BB
      SHR = 1
      CC = DD * SHR

```

```

1
2      !$OMP      END WORKSHARE
3      !$OMP      END PARALLEL
4
5      END SUBROUTINE A11_5

```

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

Example A.11.6f

```

9
10     SUBROUTINE A11_6_WRONG(AA, BB, CC, DD, N)
11     INTEGER N
12     REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
13
14     INTEGER PRI
15
16     !$OMP  PARALLEL PRIVATE(PRI)
17     !$OMP  WORKSHARE
18         AA = BB
19         PRI = 1
20         CC = DD * PRI
21     !$OMP  END WORKSHARE
22     !$OMP  END PARALLEL
23
24     END SUBROUTINE A11_6_WRONG

```

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

Example A.11.7f

```

26
27     SUBROUTINE A11_7(AA, BB, CC, N)
28     INTEGER N
29     REAL AA(N), BB(N), CC(N)
30
31     !$OMP  PARALLEL
32     !$OMP  WORKSHARE
33         AA(1:50) = BB(11:60)
34         CC(11:20) = AA(1:10)

```

```
1      !$OMP      END WORKSHARE
2      !$OMP      END PARALLEL
3
4      END SUBROUTINE A11_7
```

Fortran

A.12 The master Construct

The following example demonstrates the master construct (Section 2.7.1 on page 51). In the example, the master keeps track of how many iterations have been executed and prints out a progress report. The other threads skip the master region without waiting.

C/C++

Example A.12.1c

```
11 #include <stdio.h>
12
13 extern float average(float,float,float);
14
15 void a12( float* x, float* xold, int n, float tol )
16 {
17     int c, i, toobig;
18     float error, y;
19     c = 0;
20     #pragma omp parallel
21     {
22         do{
23             #pragma omp for private(i)
24             for( i = 1; i < n-1; ++i ){
25                 xold[i] = x[i];
26             }
27             #pragma omp single
28             {
29                 toobig = 0;
30             }
31             #pragma omp for private(i,y,error) reduction(+:toobig)
32             for( i = 1; i < n-1; ++i ){
33                 y = x[i];
34                 x[i] = average( xold[i-1], x[i], xold[i+1] );
35                 error = y - x[i];
36                 if( error > tol || error < -tol ) ++toobig;
37             }
38             #pragma omp master
39             {
40                 ++c;
41                 printf( "iteration %d, toobig=%d\n", c, toobig );
42             }
43         }
44     }
45 }
```



```

1      }
2      }while( toobig > 0 );
3      }
4  }

```

▲ C/C++ ▲

▼ Fortran ▼

Example A.12.1f

```

8      SUBROUTINE A12( X, XOLD, N, TOL )
9      REAL X(*), XOLD(*), TOL
10     INTEGER N
11     INTEGER C, I, TOOBIG
12     REAL ERROR, Y, AVERAGE
13     EXTERNAL AVERAGE
14     C = 0
15     TOOBIG = 1
16     !$OMP PARALLEL
17         DO WHILE( TOOBIG > 0 )
18     !$OMP DO PRIVATE(I)
19         DO I = 2, N-1
20             XOLD(I) = X(I)
21         ENDDO
22     !$OMP SINGLE
23         TOOBIG = 0
24     !$OMP END SINGLE
25     !$OMP DO PRIVATE(I,Y,ERROR), REDUCTION(+:TOOBIG)
26         DO I = 2, N-1
27             Y = X(I)
28             X(I) = AVERAGE( XOLD(I-1), X(I), XOLD(I+1) )
29             ERROR = Y-X(I)
30             IF( ERROR > TOL .OR. ERROR < -TOL ) TOOBIG = TOOBIG+1
31         ENDDO
32     !$OMP MASTER
33         C = C + 1
34         PRINT *, 'Iteration ', C, ' TOOBIG=', TOOBIG
35     !$OMP END MASTER
36     ENDDO
37     !$OMP END PARALLEL
38     END SUBROUTINE A12

```

▲ Fortran ▲

A.13 The critical Construct

The following example includes several **critical** constructs (Section 2.7.2 on page 52). The example illustrates a queuing model in which a task is dequeued and worked on. To guard against multiple threads dequeuing the same task, the dequeuing operation must be in a **critical** region. Because the two queues in this example are independent, they are protected by **critical** constructs with different names, *xaxis* and *yaxis*.

C/C++

Example A.13.1c

```
int dequeue(float *a);
void work(int i, float *a);

void a13(float *x, float *y)
{
    int ix_next, iy_next;

    #pragma omp parallel shared(x, y) private(ix_next, iy_next)
    {
        #pragma omp critical (xaxis)
            ix_next = dequeue(x);
        work(ix_next, x);

        #pragma omp critical (yaxis)
            iy_next = dequeue(y);
        work(iy_next, y);
    }
}
```

C/C++

Fortran

Example A.13.1f

```
      SUBROUTINE A13(X, Y)

          REAL X(*), Y(*)
          INTEGER IX_NEXT, IY_NEXT

          !$OMP PARALLEL SHARED(X, Y) PRIVATE(IX_NEXT, IY_NEXT)

          !$OMP CRITICAL(XAXIS)
              CALL DEQUEUE(IX_NEXT, X)
          !$OMP END CRITICAL(XAXIS)
              CALL WORK(IX_NEXT, X)
```

```

1      !$OMP CRITICAL(YAXIS)
2          CALL DEQUEUE(IY_NEXT,Y)
3      !$OMP END CRITICAL(YAXIS)
4          CALL WORK(IY_NEXT, Y)

5      !$OMP END PARALLEL

6          END SUBROUTINE A13

```

Fortran

A.14 Work-Sharing Constructs Inside a critical Construct

The following example demonstrates using a work-sharing construct inside a **critical** construct (see Section 2.7.2 on page 52). This example is conforming because the **single** region and the **critical** region are not closely nested (see Section 2.9 on page 87).

C/C++

Example A.14.1c

```

16 void a14()
17 {
18     int i = 1;
19     #pragma omp parallel sections
20     {
21         #pragma omp section
22         {
23             #pragma omp critical (name)
24             {
25                 #pragma omp parallel
26                 {
27                     #pragma omp single
28                     {
29                         i++;
30                     }
31                 }
32             }
33         }
34     }
35 }

```

C/C++

Example A.14.1f

```

SUBROUTINE A14()

    INTEGER I
    I = 1

    !$OMP PARALLEL SECTIONS
    !$OMP SECTION
    !$OMP CRITICAL (NAME)
    !$OMP PARALLEL
    !$OMP SINGLE
        I = I + 1
    !$OMP END SINGLE
    !$OMP END PARALLEL
    !$OMP END CRITICAL (NAME)
    !$OMP END PARALLEL SECTIONS
END SUBROUTINE A14

```

A.15 Binding of barrier Regions

The binding rules call for a **barrier** region to bind to the closest enclosing **parallel** region (see Section 2.7.3 on page 54).

In the following example, the call from the main program to *sub2* is conforming because the **barrier** region (in *sub3*) binds to the **parallel** region in *sub2*. The call from the main program to *sub1* is conforming because the **barrier** region binds to the **parallel** region in subroutine *sub2*.

The call from the main program to *sub3* is conforming because the **barrier** region binds to the implicit inactive **parallel** region enclosing the sequential part. Also note that the **barrier** region in *sub3* when called from *sub2* only synchronizes the team of threads in the enclosing **parallel** region and not all the threads created in *sub1*.

Example A.15.1c

```

void work(int n) {}

void sub3(int n)
{
    work(n);
    #pragma omp barrier
}

```

```

1      work(n);
2    }

3    void sub2(int k)
4    {
5      #pragma omp parallel shared(k)
6        sub3(k);
7    }

8    void sub1(int n)
9    {
10     int i;
11     #pragma omp parallel private(i) shared(n)
12     {
13       #pragma omp for
14       for (i=0; i<n; i++)
15         sub2(i);
16     }
17   }

18   int main()
19   {
20     sub1(2);
21     sub2(2);
22     sub3(2);
23     return 0;
24   }

```

▶ C/C++ ◀

◀ Fortran ▶

Example A.15.1f

```

28     SUBROUTINE WORK(N)
29       INTEGER N
30     END SUBROUTINE WORK

31     SUBROUTINE SUB3(N)
32       INTEGER N
33       CALL WORK(N)
34   !$OMP BARRIER
35       CALL WORK(N)
36     END SUBROUTINE SUB3

37     SUBROUTINE SUB2(K)
38       INTEGER K
39   !$OMP PARALLEL SHARED(K)
40       CALL SUB3(K)
41   !$OMP END PARALLEL
42     END SUBROUTINE SUB2

```

```

1      SUBROUTINE SUB1(N)
2      INTEGER N
3      INTEGER I
4      !$OMP PARALLEL PRIVATE(I) SHARED(N)
5      !$OMP DO
6      DO I = 1, N
7      CALL SUB2(I)
8      END DO
9      !$OMP END PARALLEL
10     END SUBROUTINE SUB1

11     PROGRAM A15
12     CALL SUB1(2)
13     CALL SUB2(2)
14     CALL SUB3(2)
15     END PROGRAM A15

```

Fortran

A.16 The atomic Construct

The following example avoids race conditions (simultaneous updates of an element of x by multiple threads) by using the **atomic** construct (Section 2.7.4 on page 55).

The advantage of using the **atomic** construct in this example is that it allows updates of two different elements of x to occur in parallel. If a **critical** construct (see Section 2.7.2 on page 52) were used instead, then all updates to elements of x would be executed serially (though not in any guaranteed order).

Note that the **atomic** directive applies only to the statement immediately following it. As a result, elements of y are not updated atomically in this example.

Example A.16.1c

```

29     float work1(int i)
30     {
31     return 1.0 * i;
32     }

33     float work2(int i)
34     {
35     return 2.0 * i;
36     }

```

C/C++

```

1 void a16(float *x, float *y, int *index, int n)
2 {
3     int i;
4
5     #pragma omp parallel for shared(x, y, index, n)
6     for (i=0; i<n; i++) {
7         #pragma omp atomic
8         x[index[i]] += work1(i);
9         y[i] += work2(i);
10    }
11
12    int main()
13    {
14        float x[1000];
15        float y[10000];
16        int index[10000];
17        int i;
18
19        for (i = 0; i < 10000; i++) {
20            index[i] = i % 1000;
21            y[i]=0.0;
22        }
23
24        for (i = 0; i < 1000; i++)
25            x[i] = 0.0;
26
27        a16(x, y, index, 10000);
28        return 0;
29    }

```

C/C++

Fortran

Example A.16.1f

```

29 REAL FUNCTION WORK1(I)
30     INTEGER I
31     WORK1 = 1.0 * I
32     RETURN
33 END FUNCTION WORK1
34
35 REAL FUNCTION WORK2(I)
36     INTEGER I
37     WORK2 = 2.0 * I
38     RETURN
39 END FUNCTION WORK2
40
41 SUBROUTINE SUBA16(X, Y, INDEX, N)
42     REAL X(*), Y(*)

```

```

1      INTEGER INDEX(*), N
2
3      INTEGER I
4
5      !$OMP PARALLEL DO SHARED(X, Y, INDEX, N)
6          DO I=1,N
7      !$OMP ATOMIC
8          X(INDEX(I)) = X(INDEX(I)) + WORK1(I)
9          Y(I) = Y(I) + WORK2(I)
10         ENDDO
11
12     END SUBROUTINE SUBA16
13
14     PROGRAM A16
15         REAL X(1000), Y(10000)
16         INTEGER INDEX(10000)
17         INTEGER I
18
19         DO I=1,10000
20             INDEX(I) = MOD(I, 1000) + 1
21             Y(I) = 0.0
22         ENDDO
23
24         DO I = 1,1000
25             X(I) = 0.0
26         ENDDO
27
28         CALL SUBA16(X, Y, INDEX, 10000)
29
30     END PROGRAM A16

```

Fortran

A.17 Restrictions on the `atomic` Construct

The following examples illustrate the restrictions on the atomic construct. For more information, see Section 2.7.4 on page 55.

C/C++

All atomic references to the storage location of each variable that appears on the left-hand side of an `atomic` assignment statement throughout the program are required to have a compatible type.

C/C++

1 Fortran

2 All atomic references to the storage location of each variable that appears on the left-
3 hand side of an **atomic** assignment statement throughout the program are required to
4 have the same type and type parameters.

5 Fortran

6 The following are some non-conforming examples:

7 C/C++

8 Example A.17.1c

```
9 void a17_1_wrong ()  
10 {  
11     union {int n; float x;} u;  
  
12     #pragma omp parallel  
13     {  
14         #pragma omp atomic  
15             u.n++;  
  
16         #pragma omp atomic  
17             u.x += 1.0;  
  
18         /* Incorrect because the atomic constructs reference the same location  
19            through incompatible types */  
20     }  
21 }
```

22 C/C++

23 Fortran

24 Example A.17.1f

```
25     SUBROUTINE A17_1_WRONG()  
26         INTEGER:: I  
27         REAL:: R  
28         EQUIVALENCE(I,R)  
  
29         !$OMP PARALLEL  
30         !$OMP ATOMIC  
31             I = I + 1  
32         !$OMP ATOMIC  
33             R = R + 1.0  
34         ! incorrect because I and R reference the same location  
35         ! but have different types  
36         !$OMP END PARALLEL
```



```

1      !$OMP  PARALLEL
2
3      !$OMP    ATOMIC
4          I = I + 1
5          CALL SUB()
6      !$OMP  END PARALLEL
7      END SUBROUTINE A17_2_WRONG

```

Although the following example might work on some implementations, this is also non-conforming:

Example A.17.3f

```

10     SUBROUTINE A17_3_WRONG
11     INTEGER:: I
12     REAL:: R
13     EQUIVALENCE(I,R)
14
15     !$OMP  PARALLEL
16     !$OMP    ATOMIC
17         I = I + 1
18     ! incorrect because I and R reference the same location
19     ! but have different types
20     !$OMP  END PARALLEL
21
22     !$OMP  PARALLEL
23     !$OMP    ATOMIC
24         R = R + 1.0
25     ! incorrect because I and R reference the same location
26     ! but have different types
27     !$OMP  END PARALLEL
28
29     END SUBROUTINE A17_3_WRONG

```

Fortran

A.18 The flush Construct with a List

The following example uses the **flush** construct (see Section 2.7.5 on page 58) for point-to-point synchronization of specific objects between pairs of threads:

C/C++

Example A.18.1c

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

C/C++ (cont.)

```

1  #define NUMBER_OF_THREADS 256
2
3  int   synch[NUMBER_OF_THREADS];
4  float work[NUMBER_OF_THREADS];
5  float result[NUMBER_OF_THREADS];
6
6  float fn1(int i)
7  {
8      return i*2.0;
9  }
10
10 float fn2(float a, float b)
11 {
12     return a + b;
13 }
14
14 int main()
15 {
16     int iam, neighbor;
17
17 #pragma omp parallel private(iam,neighbor) shared(work,synch)
18 {
19     iam = omp_get_thread_num();
20     synch[iam] = 0;
21
21     #pragma omp barrier
22     /*Do computation into my portion of work array */
23     work[iam] = fn1(iam);
24
24     /* Announce that I am done with my work. The first flush
25      * ensures that my work is made visible before synch.
26      * The second flush ensures that synch is made visible.
27      */
28
28     #pragma omp flush(work,synch)
29     synch[iam] = 1;
30     #pragma omp flush(synch)
31
31     /* Wait for neighbor. The first flush ensures that synch is read
32      * from memory, rather than from the temporary view of memory.
33      * The second flush ensures that work is read from memory, and
34      * is done so after the while loop exits.
35      */
36
36     neighbor = (iam>0 ? iam : omp_get_num_threads()) - 1;
37     while (synch[neighbor] == 0) {
38         #pragma omp flush(synch)

```

```

1         }
2         #pragma omp flush(work,synch)
3         /* Read neighbor's values of work array */
4         result[iam] = fn2(work[neighbor], work[iam]);
5     }
6
6     /* output result here */
7
7     return 0;
8 }

```

C/C++

Fortran

Example A.18.1f

```

12     REAL FUNCTION FN1(I)
13         INTEGER I
14         FN1 = I * 2.0
15         RETURN
16     END FUNCTION FN1
17
17     REAL FUNCTION FN2(A, B)
18         REAL A, B
19         FN2 = A + B
20         RETURN
21     END FUNCTION FN2
22
22     PROGRAM A18
23
23         INCLUDE "omp_lib.h"      ! or USE OMP_LIB
24         INTEGER ISYNC(256)
25         REAL    WORK(256)
26         REAL    RESULT(256)
27
27         INTEGER IAM, NEIGHBOR
28
28     !$OMP PARALLEL PRIVATE(IAM, NEIGHBOR) SHARED(WORK, ISYNC)
29         IAM = OMP_GET_THREAD_NUM() + 1
30         ISYNC(IAM) = 0
31
31     !$OMP BARRIER
32
32     C        Do computation into my portion of work array
33
33         WORK(IAM) = FN1(IAM)
34
34     C        Announce that I am done with my work.

```

```

1      C      The first flush ensures that my work is made visible before
2      C      synch. The second flush ensures that synch is made visible.

3      !$OMP   FLUSH(WORK,ISYNC)
4              ISYNC(IAM) = 1
5      !$OMP   FLUSH(ISYNC)

6      C      Wait until neighbor is done. The first flush ensures that
7      C      synch is read from memory, rather than from the temporary
8      C      view of memory. The second flush ensures that work is read
9      C      from memory, and is done so after the while loop exits.

10     IF (IAM .EQ. 1) THEN
11         NEIGHBOR = OMP_GET_NUM_THREADS()
12     ELSE
13         NEIGHBOR = IAM - 1
14     ENDIF

15     DO WHILE (ISYNC(NEIGHBOR) .EQ. 0)
16 !$OMP   FLUSH(ISYNC)
17     END DO

18 !$OMP   FLUSH(WORK, ISYNC)
19     RESULT(IAM) = FN2(WORK(NEIGHBOR), WORK(IAM))
20 !$OMP   END PARALLEL

21     END PROGRAM A18

```

▶ Fortran ◀

A.19 The flush Construct without a List

The following example (for Section 2.7.5 on page 58) distinguishes the shared objects affected by a `flush` construct with no list from the shared objects that are not affected:

▶ C/C++ ◀

Example A.19.1c

```

26     int x, *p = &x;
27
28     void f1(int *q)
29     {
30         *q = 1;
31         #pragma omp flush
32         /* x, p, and *q are flushed */
33         /* because they are shared and accessible */
34         /* q is not flushed because it is not shared. */
35     }

```

```

1      }
2
3      void f2(int *q)
4      {
5          #pragma omp barrier
6          *q = 2;
7          #pragma omp barrier
8
9          /* a barrier implies a flush */
10         /* x, p, and *q are flushed */
11         /* because they are shared and accessible */
12         /* q is not flushed because it is not shared. */
13     }
14
15     int g(int n)
16     {
17         int i = 1, j, sum = 0;
18         *p = 1;
19         #pragma omp parallel reduction(+: sum) num_threads(10)
20         {
21             f1(&j);
22
23             /* i, n and sum were not flushed */
24             /* because they were not accessible in f1 */
25             /* j was flushed because it was accessible */
26             sum += j;
27
28             f2(&j);
29
30             /* i, n, and sum were not flushed */
31             /* because they were not accessible in f2 */
32             /* j was flushed because it was accessible */
33             sum += i + j + *p + n;
34         }
35
36         return sum;
37     }
38
39     int main()
40     {
41         int result = g(7);
42         return result;
43     }

```

▲ C/C++ ▲

▼ Fortran ▼

Example A.19.1f

SUBROUTINE F1(Q)

```

1
2         COMMON /DATA/ X, P
3         INTEGER, TARGET  :: X
4         INTEGER, POINTER :: P
5         INTEGER Q

6         Q = 1
7     !$OMP FLUSH
8         ! X, P and Q are flushed
9         ! because they are shared and accessible
10        END SUBROUTINE F1

11       SUBROUTINE F2(Q)
12         COMMON /DATA/ X, P
13         INTEGER, TARGET  :: X
14         INTEGER, POINTER :: P
15         INTEGER Q

16     !$OMP BARRIER
17         Q = 2
18     !$OMP BARRIER
19         ! a barrier implies a flush
20         ! X, P and Q are flushed
21         ! because they are shared and accessible
22        END SUBROUTINE F2

23       INTEGER FUNCTION G(N)
24         COMMON /DATA/ X, P
25         INTEGER, TARGET  :: X
26         INTEGER, POINTER :: P
27         INTEGER N
28         INTEGER I, J, SUM

29         I = 1
30         SUM = 0
31         P = 1
32     !$OMP PARALLEL REDUCTION(+: SUM) NUM_THREADS(10)
33         CALL F1(J)
34         ! I, N and SUM were not flushed
35         ! because they were not accessible in F1
36         ! J was flushed because it was accessible
37         SUM = SUM + J

38         CALL F2(J)
39         ! I, N, and SUM were not flushed
40         ! because they were not accessible in f2
41         ! J was flushed because it was accessible
42         SUM = SUM + I + J + P + N

```



```

1      !$OMP   END PARALLEL
2
3      G = SUM
      END FUNCTION G
4
5      PROGRAM A19
      COMMON /DATA/ X, P
6      INTEGER, TARGET  :: X
7      INTEGER, POINTER :: P
8      INTEGER RESULT, G
9
10     P => X
11     RESULT = G(7)
12     PRINT *, RESULT
      END PROGRAM A19

```

Fortran

C/C++

A.20 Placement of `flush` and `barrier` Directives

The following example is non-conforming, because the `flush` and `barrier` directives cannot be the immediate substatement of an `if` statement. See Section 2.7.3 on page 54 and Section 2.7.5 on page 58.

Example A.20.1c

```

21 void a20_wrong()
22 {
23     int a = 1;
24
25     #pragma omp parallel
26     {
27         if (a != 0)
28         #pragma omp flush(a)
29         /* incorrect as flush cannot be immediate substatement
30         of if statement */
31
32         if (a != 0)
33         #pragma omp barrier
34         /* incorrect as barrier cannot be immediate substatement
35         of if statement */

```

```
1 }  
2 }
```

3 The following version of the above example is conforming because the **flush** and
4 **barrier** directives are enclosed in a compound statement.

5 Example A.20.2c

```
6 void a20()  
7 {  
8     int a = 1;  
  
9     #pragma omp parallel  
10    {  
11        if (a != 0) {  
12            #pragma omp flush(a)  
13        }  
14        if (a != 0) {  
15            #pragma omp barrier  
16        }  
17    }  
18 }
```

19  C/C++

20 A.21 The ordered Clause and the ordered 21 Construct

22 Ordered constructs (Section 2.7.6 on page 61) are useful for sequentially ordering the
23 output from work that is done in parallel. The following program prints out the indices
24 in sequential order:

25 Example A.21.1c

```
26 #include <stdio.h>  
  
27  
28 void work(int k)  
29 {  
30     #pragma omp ordered  
31     printf(" %d\n", k);  
32 }  
  
33 void a21(int lb, int ub, int stride)  
34 {
```

25  C/C++

```

1      int i;
2
3      #pragma omp parallel for ordered schedule(dynamic)
4      for (i=lb; i<ub; i+=stride)
5          work(i);
6      }
7
8      int main()
9      {
10         a21(0, 100, 5);
11         return 0;
12     }

```

C/C++

Fortran

Example A.21.1f

```

14      SUBROUTINE WORK(K)
15          INTEGER k
16
17      !$OMP ORDERED
18          WRITE(*,*) K
19      !$OMP END ORDERED
20
21      END SUBROUTINE WORK
22
23      SUBROUTINE SUBA21(LB, UB, STRIDE)
24          INTEGER LB, UB, STRIDE
25          INTEGER I
26
27      !$OMP PARALLEL DO ORDERED SCHEDULE(DYNAMIC)
28          DO I=LB,UB,STRIDE
29              CALL WORK(I)
30          END DO
31      !$OMP END PARALLEL DO
32
33      END SUBROUTINE SUBA21
34
35      PROGRAM A21
36          CALL SUBA21(1,100,5)
37      END PROGRAM A21

```

Fortran

It is possible to have multiple **ordered** constructs within a loop region with the **ordered** clause specified. The first example is non-conforming because all iterations execute two **ordered** regions. An iteration of a loop must not execute more than one **ordered** region:

C/C++

Example A.21.2c

```
void work(int i) {}

void a21_wrong(int n)
{
    int i;
    #pragma omp for ordered
    for (i=0; i<n; i++) {
/* incorrect because an iteration may not execute more than one
   ordered region */
        #pragma omp ordered
            work(i);
        #pragma omp ordered
            work(i+1);
    }
}
```

C/C++

Fortran

Example A.21.2f

```
      SUBROUTINE WORK(I)
      INTEGER I
      END SUBROUTINE WORK

      SUBROUTINE A21_WRONG(N)
      INTEGER N

      INTEGER I
!$OMP DO ORDERED
      DO I = 1, N
! incorrect because an iteration may not execute more than one
! ordered region
!$OMP ORDERED
          CALL WORK(I)
!$OMP END ORDERED

!$OMP ORDERED
          CALL WORK(I+1)
!$OMP END ORDERED
      END DO
      END SUBROUTINE A21_WRONG
```

Fortran

The following is a conforming example with more than one **ordered** construct. Each iteration will execute only one **ordered** region:

C/C++

Example A.21.3c

```
void a21_good(int n)
{
    int i;

    #pragma omp for ordered
    for (i=0; i<n; i++) {
        if (i <= 10) {
            #pragma omp ordered
            work(i);
        }

        if (i > 10) {
            #pragma omp ordered
            work(i+1);
        }
    }
}
```

C/C++

Fortran

Example A.21.3f

```
      SUBROUTINE A21_GOOD(N)
      INTEGER N

      !$OMP DO ORDERED
      DO I = 1,N
          IF (I <= 10) THEN
      !$OMP ORDERED
              CALL WORK(I)
      !$OMP END ORDERED
          ENDIF

          IF (I > 10) THEN
      !$OMP ORDERED
              CALL WORK(I+1)
      !$OMP END ORDERED
          ENDIF
      ENDDO
      END SUBROUTINE A21_GOOD
```

Fortran

A.22 The threadprivate Directive

The following examples demonstrate how to use the **threadprivate** directive (Section 2.8.2 on page 66) to give each thread a separate counter.

C/C++

Example A.22.1c

```
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
    counter++;
    return(counter);
}
```

C/C++

Fortran

Example A.22.1f

```
INTEGER FUNCTION INCREMENT_COUNTER()
COMMON/A22_COMMON/COUNTER
!$OMP THREADPRIVATE(/A22_COMMON/)

COUNTER = COUNTER +1
INCREMENT_COUNTER = COUNTER
RETURN
END FUNCTION INCREMENT_COUNTER
```

Fortran

C/C++

The following example uses **threadprivate** on a static variable:

Example A.22.2c

```
int increment_counter_2()
{
    static int counter = 0;
    #pragma omp threadprivate(counter)
    counter++;
    return(counter);
}
```

1 The following example illustrates how modifying a variable that appears in an initializer
2 can cause unspecified behavior, and also how to avoid this problem by using an auxiliary
3 object and a copy-constructor.

4 Example A.22.3c

```
5 class T {  
6     public:  
7         int val;  
8         T (int);  
9         T (const T&);  
10    };  
  
11    T :: T (int v){  
12        val = v;  
13    }  
  
14    T :: T (const T& t) {  
15        val = t.val;  
16    }  
  
17    void g(T a, T b){  
18        a.val += b.val;  
19    }  
  
20    int x = 1;  
21    T a(x);  
22    const T b_aux(x); /* Capture value of x = 1 */  
23    T b(b_aux);  
24    #pragma omp threadprivate(a, b)  
  
25    void f(int n) {  
26        x++;  
27        #pragma omp parallel for  
28        /* In each thread:  
29         * Object a is constructed from x (with value 1 or 2?)  
30         * Object b is copy-constructed from b_aux  
31         */  
  
32        for (int i=0; i<n; i++) {  
33            g(a, b); /* Value of a is unspecified. */  
34        }  
35    }  
36
```

▲ C/C++ ▲

The following examples show non-conforming uses and correct uses of the **threadprivate** directive. For more information, see Section 2.8.2 on page 66 and Section 2.8.4.1 on page 84.

The following example is non-conforming because the common block is not declared local to the subroutine that refers to it:

Example A.22.4f

```

MODULE A22_MODULE
  COMMON /T/ A
END MODULE A22_MODULE

SUBROUTINE A22_4_WRONG()
  USE A22_MODULE
!$OMP  THREADPRIVATE(/T/)
  !non-conforming because /T/ not declared in A22_4_WRONG
END SUBROUTINE A22_4_WRONG

```

The following example is also non-conforming because the common block is not declared local to the subroutine that refers to it:

Example A.22.5f

```

SUBROUTINE A22_5_WRONG()
  COMMON /T/ A
!$OMP  THREADPRIVATE(/T/)

  CONTAINS
    SUBROUTINE A22_5S_WRONG()
!$OMP  PARALLEL COPYIN(/T/)
    !non-conforming because /T/ not declared in A22_5S_WRONG
!$OMP  END PARALLEL
    END SUBROUTINE A22_5S_WRONG
END SUBROUTINE A22_5_WRONG

```

The following example is a correct rewrite of the previous example:

Example A.22.6f

```

SUBROUTINE A22_6_GOOD()
  COMMON /T/ A
!$OMP  THREADPRIVATE(/T/)

```



```

CONTAINS
  SUBROUTINE A22_6S_GOOD()
    COMMON /T/ A
!$OMP    THREADPRIVATE(/T/)

!$OMP    PARALLEL COPYIN(/T/)
!$OMP    END PARALLEL
  END SUBROUTINE A22_6S_GOOD
END SUBROUTINE A22_6_GOOD

```

The following is an example of the use of `threadprivate` for local variables:

Example A.22.7f

```

PROGRAM A22_7_GOOD
  INTEGER, ALLOCATABLE, SAVE :: A(:)
  INTEGER, POINTER, SAVE :: PTR
  INTEGER, SAVE :: I
  INTEGER, TARGET :: TARG
  LOGICAL :: FIRSTIN = .TRUE.
!$OMP  THREADPRIVATE(A, I, PTR)

  ALLOCATE (A(3))
  A = (/1,2,3/)
  PTR => TARG
  I = 5

!$OMP  PARALLEL COPYIN(I, PTR)
!$OMP  CRITICAL
  IF (FIRSTIN) THEN
    TARG = 4           ! Update target of ptr
    I = I + 10
    IF (ALLOCATED(A)) A = A + 10
    FIRSTIN = .FALSE.
  END IF

  IF (ALLOCATED(A)) THEN
    PRINT *, 'a = ', A
  ELSE
    PRINT *, 'A is not allocated'
  END IF

  PRINT *, 'ptr = ', PTR
  PRINT *, 'i = ', I
  PRINT *

```

```

1
2      !$OMP      END CRITICAL
3      !$OMP      END PARALLEL
4      END PROGRAM A22_7_GOOD

```

The above program, if executed by two threads, will print one of the following two sets of output:

```

7      a = 11 12 13
8      ptr = 4
9      i = 15

```

```

10     A is not allocated
11     ptr = 4
12     i = 5

```

or

```

14     A is not allocated
15     ptr = 4
16     i = 15

```

```

17     a = 1 2 3
18     ptr = 4
19     i = 5

```

The following is an example of the use of **threadprivate** for module variables:

Example A.22.8f

```

21
22     MODULE A22_MODULES
23         REAL, POINTER :: WORK(:)
24         SAVE WORK
25     !$OMP   THREADPRIVATE(WORK)
26     END MODULE A22_MODULES
27
28     SUBROUTINE SUB1(N)
29     USE A22_MODULES
30     !$OMP   PARALLEL PRIVATE(THIS_SUM)
31         ALLOCATE(WORK(N))
32         CALL SUB2(THIS_SUM)
33         WRITE(*,*)THIS_SUM
34     !$OMP   END PARALLEL
35     END SUBROUTINE SUB1

```

```

1      SUBROUTINE SUB2(THE_SUM)
2          USE A22_MODULE8
3          WORK(:) = 10
4          THE_SUM=SUM(WORK)
5      END SUBROUTINE SUB2

6      PROGRAM A22_8_GOOD
7          N = 10
8          CALL SUB1(N)
9      END PROGRAM A22_8_GOOD

```

Fortran

Fortran

A.23 Fortran Restrictions on shared and private Clauses with Common Blocks

When a named common block is specified in a **private**, **firstprivate**, or **lastprivate** clause of a construct, none of its members may be declared in another data-sharing attribute clause on that construct. The following examples illustrate this point. For more information, see Section 2.8.3 on page 70.

The following example is conforming:

Example A.23.1f

```

20      SUBROUTINE A23_1_GOOD()
21          COMMON /C/ X,Y
22          REAL X, Y

23      !$OMP PARALLEL PRIVATE (/C/)
24          ! do work here
25      !$OMP END PARALLEL

26      !$OMP PARALLEL SHARED (X,Y)
27          ! do work here
28      !$OMP END PARALLEL
29      END SUBROUTINE A23_1_GOOD

```

The following example is also conforming:

Example A.23.2f

```

SUBROUTINE A23_2_GOOD()
  COMMON /C/ X,Y
  REAL X, Y

  INTEGER I

!$OMP PARALLEL
!$OMP DO PRIVATE(/C/)
  DO I=1,1000
    ! do work here
  ENDDO
!$OMP END DO
!
!$OMP DO PRIVATE(X)
  DO I=1,1000
    ! do work here
  ENDDO
!$OMP END DO
!$OMP END PARALLEL
END SUBROUTINE A23_2_GOOD

```

The following example is conforming:

Example A.23.3f

```

SUBROUTINE A23_3_GOOD()
  COMMON /C/ X,Y

!$OMP PARALLEL PRIVATE (/C/)
  ! do work here
!$OMP END PARALLEL

!$OMP PARALLEL SHARED (/C/)
  ! do work here
!$OMP END PARALLEL
END SUBROUTINE A23_3_GOOD

```

The following example is non-conforming because *x* is a constituent element of *c*:

Example A.23.4f

```

SUBROUTINE A23_4_WRONG()

```

```

1          COMMON /C/ X,Y
2      ! Incorrect because X is a constituent element of C
3      !$OMP  PARALLEL PRIVATE(/C/), SHARED(X)
4          ! do work here
5      !$OMP  END PARALLEL
6          END SUBROUTINE A23_4_WRONG

```

The following example is non-conforming because a common block may not be declared both shared and private:

Example A.23.5f

```

9          SUBROUTINE A23_5_WRONG()
10         COMMON /C/ X,Y
11         ! Incorrect: common block C cannot be declared both
12         ! shared and private
13         !$OMP  PARALLEL PRIVATE (/C/), SHARED(/C/)
14             ! do work here
15         !$OMP  END PARALLEL
16
17         END SUBROUTINE A23_5_WRONG

```

▶ Fortran ◀

A.24 The default (none) Clause

The following example distinguishes the variables that are affected by the **default(none)** clause from those that are not. For more information on the **default** clause, see Section 2.8.3.1 on page 71.

▶ C/C++ ◀

Example A.24.1c

```

25 #include <omp.h>
26 int x, y, z[1000];
27 #pragma omp threadprivate(x)
28
29 void a24(int a) {
30     const int c = 1;
31     int i = 0;
32
33     #pragma omp parallel default(none) private(a) shared(z)
34     {
35         int j = omp_get_num_threads();
36         /* O.K. - j is declared within parallel region */

```

```

1      a = z[j]; /* O.K. - a is listed in private clause */
2              /*      - z is listed in shared clause */
3      x = c;    /* O.K. - x is threadprivate */
4              /*      - c has const-qualified type */
5      z[i] = y; /* Error - cannot reference i or y here */

6      #pragma omp for firstprivate(y)
7      for (i=0; i<10 ; i++) {
8          z[i] = y; /* O.K. - i is the loop iteration variable */
9                  /* - y is listed in firstprivate clause */
10         }

11     z[i] = y; /* Error - cannot reference i or y here */
12 }
13 }
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38

```

Example A.24.1f

```

17      SUBROUTINE A24(A)
18      INCLUDE "omp_lib.h"      ! or USE OMP_LIB

19      INTEGER A

20      INTEGER X, Y, Z(1000)
21      COMMON/BLOCKX/X
22      COMMON/BLOCKY/Y
23      COMMON/BLOCKZ/Z
24      !$OMP THREADPRIVATE(/BLOCKX/)

25      INTEGER I, J
26      i = 1

27      !$OMP PARALLEL DEFAULT(NONE) PRIVATE(A) SHARED(Z) PRIVATE(J)
28          J = OMP_GET_NUM_THREADS();
29              ! O.K. - J is listed in PRIVATE clause
30          A = Z(J) ! O.K. - A is listed in PRIVATE clause
31              !      - Z is listed in SHARED clause
32          X = 1    ! O.K. - X is THREADPRIVATE
33          Z(I) = Y ! Error - cannot reference I or Y here

34      !$OMP DO firstprivate(y)
35          DO I = 1,10
36              Z(I) = Y ! O.K. - I is the loop iteration variable
37                  ! Y is listed in FIRSTPRIVATE clause
38          END DO

```

```

1           Z(I) = Y      ! Error - cannot reference I or Y here
2 !$OMP   END PARALLEL
3           END SUBROUTINE A24

```

Fortran

Fortran

A.25 Race Conditions Caused by Implied Copies of Shared Variables in Fortran

The following example contains a race condition, because the shared variable, which is an array section, is passed as an actual argument to a routine that has an assumed-size array as its dummy argument (see Section 2.8.3.2 on page 72). The subroutine call passing an array section argument may cause the compiler to copy the argument into a temporary location prior to the call and copy from the temporary location into the original variable when the subroutine returns. This copying would cause races in the `parallel` region.

Example A.25.1f

```

15 SUBROUTINE A25
16
17     INCLUDE "omp_lib.h"      ! or USE OMP_LIB
18
19     REAL A(20)
20     INTEGER MYTHREAD
21
22 !$OMP PARALLEL SHARED(A) PRIVATE(MYTHREAD)
23
24     MYTHREAD = OMP_GET_THREAD_NUM()
25     IF (MYTHREAD .EQ. 0) THEN
26         CALL SUB(A(1:10)) ! compiler may introduce writes to A(6:10)
27     ELSE
28         A(6:10) = 12
29     ENDIF
30
31 !$OMP END PARALLEL
32
33 END SUBROUTINE A25
34
35 SUBROUTINE SUB(X)
36     REAL X(*)
37     X(1:5) = 4

```

1 END SUBROUTINE SUB

2 └────────────────── Fortran ───────────────────┘

3 A.26 The private Clause

4 In the following example, the values of *i* and *j* are undefined on exit from the
5 **parallel** region. For more information on the **private** clause, see Section 2.8.3.3
6 on page 73.

7 └────────────────── C/C++ ───────────────────┘

8 Example A.26.1c

```
9   #include <stdio.h>
10  int main()
11  {
12     int i, j;
13
14     i = 1;
15     j = 2;
16
17     #pragma omp parallel private(i) firstprivate(j)
18     {
19         i = 3;
20         j = j + 2;
21     }
22     printf("%d %d\n", i, j); /* i and j are undefined */
23     return 0;
24  }
```

23 └────────────────── C/C++ ───────────────────┘

24 └────────────────── Fortran ───────────────────┘

25 Example A.26.1f

```
26           PROGRAM A26
27            INTEGER I, J
28
29            I = 1
30            J = 2
31
32            !$OMP   PARALLEL PRIVATE(I) FIRSTPRIVATE(J)
33                 I = 3
34                 J = J + 2
35            !$OMP   END PARALLEL
```



```
1          PRINT *, I, J ! I and J are undefined
2      END PROGRAM A26
```

Fortran

The **private** clause of a **parallel** construct is only in effect inside the construct, and not for the rest of the region. Therefore, in the example that follows, any uses of the variable *a* within the loop in the routine *f* refers to a private copy of *a*, while a usage in routine *g* refers to the global *a*.

C/C++

Example A.26.2c

```
10      int a;

11      void g(int k) {
12          a = k;          /* The global "a", not the private "a" in f */
13      }

14      void f(int n) {
15          int a = 0;

16          #pragma omp parallel for private(a)
17          for (int i=1; i<n; i++) {
18              a = i;
19              g(a*2);      /* Private copy of "a" */
20          }
21      }
```

C/C++

Fortran

Example A.26.2f

```
25      MODULE A26_2
26          REAL A

27          CONTAINS

28          SUBROUTINE G(K)
29              REAL K
30              A = K ! This is A in module A26_2, not the private
31                  ! A in F
32          END SUBROUTINE G

33          SUBROUTINE F(N)
34              INTEGER N
35              REAL A
```

```

1          INTEGER I
2      !$OMP   PARALLEL DO PRIVATE(A)
3              DO I = 1,N
4                  A = I
5                  CALL G(A*2)
6              ENDDO
7      !$OMP   END PARALLEL DO
8      END SUBROUTINE F

9
10     END MODULE A26_2

```

Fortran

A.27 Reprivatization

The following example demonstrates the reprivatization of variables (see Section 2.8.3.3 on page 73). Private variables can be marked **private** again in a nested construct. They do not have to be shared in the enclosing **parallel** region.

C/C++

Example A.27.1c

```

17 void a27()
18 {
19     int i, a;

20     #pragma omp parallel private(a)
21     {
22         #pragma omp parallel for private(a)
23         for (i=0; i<10; i++)
24         {
25             /* do work here */
26         }
27     }
28 }

```

C/C++

Fortran

Example A.27.1f

```

32     SUBROUTINE A27()
33         INTEGER I, A

34     !$OMP   PARALLEL PRIVATE(A)
35     !$OMP   PARALLEL DO PRIVATE(A)
36         DO I = 1, 10
37             ! do work here

```

```

1           END DO
2 !$OMP     END PARALLEL DO
3 !$OMP     END PARALLEL
4           END SUBROUTINE A27

```

Fortran

Fortran

A.28 Fortran Restrictions on Storage Association with the `private` Clause

The following non-conforming examples illustrate the implications of the `private` clause rules with regard to storage association (see Section 2.8.3.3 on page 73).

Example A.28.1f

```

12           SUBROUTINE SUB()
13             COMMON /BLOCK/ X
14             PRINT *,X           ! X is undefined
15           END SUBROUTINE SUB

16           PROGRAM A28_1
17             COMMON /BLOCK/ X
18             X = 1.0
19 !$OMP     PARALLEL PRIVATE (X)
20             X = 2.0
21             CALL SUB()
22 !$OMP     END PARALLEL
23           END PROGRAM A28_1

```

Example A.28.2f

```

25           PROGRAM A28_2
26             COMMON /BLOCK2/ X
27             X = 1.0

28 !$OMP     PARALLEL PRIVATE (X)
29             X = 2.0
30             CALL SUB()
31 !$OMP     END PARALLEL

32           CONTAINS

33           SUBROUTINE SUB()

```

```

2          COMMON /BLOCK2/ Y

3          PRINT *,X           ! X is undefined
4          PRINT *,Y           ! Y is undefined
5          END SUBROUTINE SUB

6          END PROGRAM A28_2

```

Example A.28.3f

```

8          PROGRAM A28_3
9          EQUIVALENCE (X,Y)
10         X = 1.0

11         !$OMP PARALLEL PRIVATE(X)
12             PRINT *,Y           ! Y is undefined
13             Y = 10
14             PRINT *,X           ! X is undefined
15         !$OMP END PARALLEL
16         END PROGRAM A28_3

```

Example A.28.4f

```

18         PROGRAM A28_4
19         INTEGER I, J
20         INTEGER A(100), B(100)
21         EQUIVALENCE (A(51), B(1))

22         !$OMP PARALLEL DO DEFAULT(PRIVATE) PRIVATE(I,J) LASTPRIVATE(A)
23             DO I=1,100
24                 DO J=1,100
25                     B(J) = J - 1
26                 ENDDO

27                 DO J=1,100
28                     A(J) = J     ! B becomes undefined at this point
29                 ENDDO

30                 DO J=1,50
31                     B(J) = B(J) + 1 ! B is undefined
32                                     ! A becomes undefined at this point
33                 ENDDO
34             ENDDO
35         !$OMP END PARALLEL DO     ! The LASTPRIVATE write for A has
36                                     ! undefined results

```

```

1         PRINT *, B      ! B is undefined since the LASTPRIVATE
2                       ! write of A was not defined
3     END PROGRAM A28_4

```

Example A.28.5f

```

5     SUBROUTINE SUB1(X)
6         DIMENSION X(10)

7         ! This use of X does not conform to the
8         ! specification. It would be legal Fortran 90,
9         ! but the OpenMP private directive allows the
10        ! compiler to break the sequence association that
11        ! A had with the rest of the common block.

```

```

12        FORALL (I = 1:10) X(I) = I
13    END SUBROUTINE SUB1

```

```

14    PROGRAM A28_5
15        COMMON /BLOCK5/ A

```

```

16        DIMENSION B(10)
17        EQUIVALENCE (A,B(1))

```

```

18        ! the common block has to be at least 10 words
19        A = 0

```

```

20    !$OMP PARALLEL PRIVATE(/BLOCK5/)

```

```

21        ! Without the private clause,
22        ! we would be passing a member of a sequence
23        ! that is at least ten elements long.
24        ! With the private clause, A may no longer be
25        ! sequence-associated.

```

```

26        CALL SUB1(A)
27    !$OMP MASTER
28        PRINT *, A
29    !$OMP END MASTER

```

```

30    !$OMP END PARALLEL
31    END PROGRAM A28_5

```

Fortran

A.29 C/C++ Arrays in a `firstprivate` Clause

The following example illustrates the size and value of list items of array or pointer type in a `firstprivate` clause (Section 2.8.3.4 on page 75). The size of new list items is based on the type of the corresponding original list item, as determined by the base language.

In this example:

- The type of **A** is array of two arrays of two ints.
- The type of **B** is adjusted to pointer to array of **n** ints, because it is a function parameter.
- The type of **C** is adjusted to pointer to int, because it is a function parameter.
- The type of **D** is array of two arrays of two ints.
- The type of **E** is array of **n** arrays of **n** ints.

Note that **B** and **E** involve variable length array types.

The new items of array type are initialized as if each integer element of the original array is assigned to the corresponding element of the new array. Those of pointer type are initialized as if by assignment from the original item to the new item.

Example A.29.1c

```
#include <assert.h>

int A[2][2] = {1, 2, 3, 4};

void f(int n, int B[n][n], int C[])
{
    int D[2][2] = {1, 2, 3, 4};
    int E[n][n];

    assert(n >= 2);
    E[1][1] = 4;

    #pragma omp parallel firstprivate(B, C, D, E)
    {
        assert(sizeof(B) == sizeof(int (*)[n]));
        assert(sizeof(C) == sizeof(int*));
        assert(sizeof(D) == 4 * sizeof(int));
        assert(sizeof(E) == n * n * sizeof(int));
    }
}
```

```

1      /* Private B and C have values of original B and C. */
2      assert(&B[1][1] == &A[1][1]);
3      assert(&C[3] == &A[1][1]);
4      assert(D[1][1] == 4);
5      assert(E[1][1] == 4);
6      }
7      }

8      int main() {
9          f(2, A, A[0]);
10         return 0;
11     }

```

C/C++

A.30 The lastprivate Clause

Correct execution sometimes depends on the value that the last iteration of a loop assigns to a variable. Such programs must list all such variables in a **lastprivate** clause (Section 2.8.3.5 on page 77) so that the values of the variables are the same as when the loop is executed sequentially.

C/C++

Example A.30.1c

```

20 void a30 (int n, float *a, float *b)
21 {
22     int i;

23     #pragma omp parallel
24     {
25         #pragma omp for lastprivate(i)
26         for (i=0; i<n-1; i++)
27             a[i] = b[i] + b[i+1];
28     }

29     a[i]=b[i];      /* i == n-1 here */
30 }

```

C/C++

Fortran

Example A.30.1f

```

34     SUBROUTINE A30(N, A, B)

35         INTEGER N
36         REAL A(*), B(*)

```

```

1           INTEGER I
2
3       !$OMP PARALLEL
4       !$OMP DO LASTPRIVATE(I)
5
6           DO I=1,N-1
7           A(I) = B(I) + B(I+1)
8           ENDDO
9
10      !$OMP END PARALLEL
11
12      A(I) = B(I)      ! I has the value of N here
13
14      END SUBROUTINE A30

```

Fortran

A.31 The reduction Clause

The following example demonstrates the **reduction** clause (Section 2.8.3.6 on page 79):

Example A.31.1c

```

15 void a31_1(float *x, int *y, int n)
16 {
17     int i, b;
18     float a;
19
20     a = 0.0;
21     b = 0;
22
23     #pragma omp parallel for private(i) shared(x, y, n) \
24     reduction(+:a) reduction(^:b)
25     for (i=0; i<n; i++) {
26         a += x[i];
27         b ^= y[i];
28     }
29 }

```

C/C++

Fortran

Example A.31.1f

```
1          SUBROUTINE A31_1(A, B, X, Y, N)
2
3              INTEGER N
4              REAL X(*), Y(*), A, B
5
6              !$OMP PARALLEL DO PRIVATE(I) SHARED(X, N) REDUCTION(+:A)
7              !$OMP& REDUCTION(MIN:B)
8
9                  DO I=1,N
10
11                     A = A + X(I)
12
13                     B = MIN(B, Y(I))
14
15                     ! Note that some reductions can be expressed in
16                     ! other forms. For example, the MIN could be expressed as
17                     ! IF (B > Y(I)) B = Y(I)
18
19                 END DO
20
21             END SUBROUTINE A31_1
```

Fortran

A common implementation of the preceding example is to treat it as if it had been written as follows:

C/C++

Example A.31.2c

```
1 void a31_2(float *x, int *y, int n)
2 {
3     int i, b, b_p;
4     float a, a_p;
5
6     a = 0.0;
7     b = 0;
8
9     #pragma omp parallel shared(a, b, x, y, n) \
10        private(a_p, b_p)
11     {
12         a_p = 0.0;
13         b_p = 0;
14
15         #pragma omp for private(i)
16         for (i=0; i<n; i++) {
```

```

1      a_p += x[i];
2      b_p ^= y[i];
3
4      }
5
6      #pragma omp critical
7      {
8          a += a_p;
9          b ^= b_p;
10     }
11 }

```

C/C++

Fortran

Example A.31.2f

```

14      SUBROUTINE A31_2 (A, B, X, Y, N)
15
16          INTEGER N
17          REAL X(*), Y(*), A, B, A_P, B_P
18
19      !$OMP PARALLEL SHARED(X, Y, N, A, B) PRIVATE(A_P, B_P)
20
21          A_P = 0.0
22          B_P = HUGE(B_P)
23
24      !$OMP DO PRIVATE(I)
25          DO I=1,N
26              A_P = A_P + X(I)
27              B_P = MIN(B_P, Y(I))
28          ENDDO
29      !$OMP END DO
30
31      !$OMP CRITICAL
32          A = A + A_P
33          B = MIN(B, B_P)
34      !$OMP END CRITICAL
35
36      !$OMP END PARALLEL
37
38      END SUBROUTINE A31_2

```

The following program is non-conforming because the reduction is on the *intrinsic procedure name* **MAX** but that name has been redefined to be the variable named **MAX**.

Example A.31.3f

```

PROGRAM A31_3_WRONG
  MAX = HUGE(0)

  M = 0

  !$OMP PARALLEL DO REDUCTION(MAX: M) ! MAX is no longer the
                                     ! intrinsic so this
                                     ! is non-conforming

  DO I = 1, 100
    CALL SUB(M,I)
  END DO

END PROGRAM A31_3_WRONG

SUBROUTINE SUB(M,I)
  M = MAX(M,I)
END SUBROUTINE SUB

```

The following conforming program performs the reduction using the *intrinsic procedure name* **MAX** even though the intrinsic **MAX** has been renamed to **REN**.

Example A.31.4f

```

MODULE M
  INTRINSIC MAX
END MODULE M

PROGRAM A31_4
  USE M, REN => MAX
  N = 0
  !$OMP PARALLEL DO REDUCTION(REN: N) ! still does MAX
  DO I = 1, 100
    N = MAX(N,I)
  END DO
END PROGRAM A31_4

```

The following conforming program performs the reduction using *intrinsic procedure name* **MAX** even though the intrinsic **MAX** has been renamed to **MIN**.

Example A.31.5f

```

MODULE MOD

```

```

1      INTRINSIC MAX, MIN
2      END MODULE MOD

3      PROGRAM A31_5
4          USE MOD, MIN=>MAX, MAX=>MIN
5          REAL :: R
6          R = -HUGE(0.0)

7          !$OMP PARALLEL DO REDUCTION(MIN: R)      ! still does MAX
8              DO I = 1, 1000
9                  R = MIN(R, SIN(REAL(I)))
10             END DO
11             PRINT *, R
12         END PROGRAM A31_5

```

Fortran

A.32 The copyin Clause

The `copyin` clause (see Section 2.8.4.1 on page 84) is used to initialize threadprivate data upon entry to a `parallel` region. The value of the threadprivate variable in the master thread is copied to the threadprivate variable of each other team member.

C/C++

Example A.32.1c

```

20      #include <stdlib.h>

21      float* work;
22      int size;
23      float tol;

24      #pragma omp threadprivate(work,size,tol)

25      void a32( float t, int n )
26      {
27          tol = t;
28          size = n;
29          #pragma omp parallel copyin(tol,size)
30          {
31              build();
32          }
33      }

34      void build()
35      {
36          int i;

```

```

1      work = (float*)malloc( sizeof(float)*size );
2      for( i = 0; i < size; ++i ) work[i] = tol;
3  }

```

C/C++

Fortran

Example A.32.1f

```

7      MODULE M
8          REAL, POINTER, SAVE :: WORK(:)
9          INTEGER :: SIZE
10         REAL :: TOL
11     !$OMP   THREADPRIVATE(WORK,SIZE,TOL)
12     END MODULE M

13     SUBROUTINE A32( T, N )
14         USE M
15         REAL :: T
16         INTEGER :: N
17         TOL = T
18         SIZE = N
19     !$OMP   PARALLEL COPYIN(TOL,SIZE)
20         CALL BUILD
21     !$OMP   END PARALLEL
22     END SUBROUTINE A32

23     SUBROUTINE BUILD
24         USE M
25         ALLOCATE(WORK(SIZE))
26         WORK = TOL
27     END SUBROUTINE BUILD

```

Fortran

A.33 The copyprivate Clause

The **copyprivate** clause (see Section 2.8.4.2 on page 85) can be used to broadcast values acquired by a single thread directly to all instances of the private variables in the other threads. In this example, if the routine is called from the sequential part, its behavior is not affected by the presence of the directives. If it is called from a **parallel** region, then the actual arguments with which *a* and *b* are associated must be private. After the input routine has been executed by one thread, no thread leaves the construct until the private objects designated by *a*, *b*, *x*, and *y* in all threads have become defined with the values read.

C/C++

Example A.33.1c

```
#include <stdio.h>
float x, y;
#pragma omp threadprivate(x, y)

void init(float a, float b ) {
    #pragma omp single copyprivate(a,b,x,y)
    {
        scanf("%f %f %f %f", &a, &b, &x, &y);
    }
}
```

C/C++

Fortran

Example A.33.1f

```
        SUBROUTINE INIT(A,B)
        REAL A, B
        COMMON /XY/ X,Y
!$OMP   THREADPRIVATE (/XY/)

!$OMP   SINGLE
        READ (11) A,B,X,Y
!$OMP   END SINGLE COPYPRIVATE (A,B,/XY/)

        END SUBROUTINE INIT
```

Fortran

In contrast to the previous example, suppose the input must be performed by a particular thread, say the master thread. In this case, the **copyprivate** clause cannot be used to do the broadcast directly, but it can be used to provide access to a temporary shared object.

C/C++

Example A.33.2c

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

float read_next( ) {
    float * tmp;
    float return_val;

    #pragma omp single copyprivate(tmp)
    {
        tmp = (float *) malloc(sizeof(float));
    } /* copies the pointer only */
}
```

```

1      #pragma omp master
2      {
3          scanf("%f", tmp);
4      }
5
6      #pragma omp barrier
7      return_val = *tmp;
8      #pragma omp barrier
9
10     #pragma omp single nowait
11     {
12         free(tmp);
13     }
14
15     return return_val;
16 }

```

C/C++

Fortran

Example A.33.2f

```

17     REAL FUNCTION READ_NEXT()
18         REAL, POINTER :: TMP
19
20     !$OMP SINGLE
21         ALLOCATE (TMP)
22     !$OMP END SINGLE COPYPRIVATE (TMP) ! copies the pointer only
23
24     !$OMP MASTER
25         READ (11) TMP
26     !$OMP END MASTER
27
28     !$OMP BARRIER
29         READ_NEXT = TMP
30     !$OMP BARRIER
31
32     !$OMP SINGLE
33         DEALLOCATE (TMP)
34     !$OMP END SINGLE NOWAIT
35     END FUNCTION READ_NEXT

```

Fortran

Suppose that the number of lock objects required within a **parallel** region cannot easily be determined prior to entering it. The **copyprivate** clause can be used to provide access to shared lock objects that are allocated within that **parallel** region.

Example A.33.3c

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <omp.h>
4
5
6  omp_lock_t *new_lock()
7  {
8      omp_lock_t *lock_ptr;
9
10     #pragma omp single copyprivate(lock_ptr)
11     {
12         lock_ptr = (omp_lock_t *) malloc(sizeof(omp_lock_t));
13         omp_init_lock( lock_ptr );
14     }
15     return lock_ptr;
16 }

```

Example A.33.3f

```

19     FUNCTION NEW_LOCK()
20     USE OMP_LIB      ! or INCLUDE "omp_lib.h"
21     INTEGER(OMP_LOCK_KIND), POINTER :: NEW_LOCK
22
23     !$OMP  SINGLE
24         ALLOCATE(NEW_LOCK)
25         CALL OMP_INIT_LOCK(NEW_LOCK)
26     !$OMP  END SINGLE COPYPRIVATE(NEW_LOCK)
27     END FUNCTION NEW_LOCK

```

Note that the effect of the **copyprivate** clause on a variable with the allocatable attribute is different than on a variable with the pointer attribute.

Example A.33.4f

```

30     SUBROUTINE S(N)
31     INTEGER N
32
33     REAL, DIMENSION(:), ALLOCATABLE :: A
34     REAL, DIMENSION(:), POINTER :: B
35
36     ALLOCATE (A(N))
37     !$OMP  SINGLE
38         ALLOCATE (B(N))

```



```

1         READ (11) A,B
2     !$OMP   END SINGLE COPYPRIVATE(A,B)
3         ! Variable A designates a private object
4         !   which has the same value in each thread
5         ! Variable B designates a shared object

6     !$OMP   BARRIER
7     !$OMP   SINGLE
8         DEALLOCATE (B)
9     !$OMP   END SINGLE NOWAIT
10        END SUBROUTINE S

```

Fortran

A.34 Nested Loop Constructs

The following example of loop construct nesting (see Section 2.9 on page 87) is conforming because the inner and outer loop regions bind to different **parallel** regions:

C/C++

Example A.34.1c

```

18 void work(int i, int j) {}

19 void good_nesting(int n)
20 {
21     int i, j;
22     #pragma omp parallel default(shared)
23     {
24         #pragma omp for
25         for (i=0; i<n; i++) {
26             #pragma omp parallel shared(i, n)
27             {
28                 #pragma omp for
29                 for (j=0; j < n; j++)
30                     work(i, j);
31             }
32         }
33     }
34 }

```

C/C++

Fortran

Example A.34.1f

```

38     SUBROUTINE WORK(I, J)

```

```

1      INTEGER I, J
2      END SUBROUTINE WORK

3      SUBROUTINE GOOD_NESTING(N)
4      INTEGER N

5          INTEGER I
6      !$OMP PARALLEL DEFAULT(SHARED)
7      !$OMP DO
8          DO I = 1, N
9      !$OMP PARALLEL SHARED(I,N)
10     !$OMP DO
11         DO J = 1, N
12             CALL WORK(I,J)
13         END DO
14     !$OMP END PARALLEL
15     END DO
16 !$OMP END PARALLEL
17 END SUBROUTINE GOOD_NESTING

```

Fortran

The following variation of the preceding example is also conforming:

C/C++

Example A.34.2c

```

22 void work(int i, int j) {}

23 void work1(int i, int n)
24 {
25     int j;
26     #pragma omp parallel default(shared)
27     {
28         #pragma omp for
29         for (j=0; j<n; j++)
30             work(i, j);
31     }
32 }

33 void good_nesting2(int n)
34 {
35     int i;
36     #pragma omp parallel default(shared)
37     {
38         #pragma omp for
39         for (i=0; i<n; i++)
40             work1(i, n);

```

```
1 }
2 }
```

C/C++

Fortran

Example A.34.2f

```
6 SUBROUTINE WORK(I, J)
7 INTEGER I, J
8 END SUBROUTINE WORK
```

```
9 SUBROUTINE WORK1(I, N)
10 INTEGER J
11 !$OMP PARALLEL DEFAULT(SHARED)
12 !$OMP DO
13 DO J = 1, N
14 CALL WORK(I,J)
15 END DO
16 !$OMP END PARALLEL
17 END SUBROUTINE WORK1
```

```
18 SUBROUTINE GOOD_NESTING2(N)
19 INTEGER N
20 !$OMP PARALLEL DEFAULT(SHARED)
21 !$OMP DO
22 DO I = 1, N
23 CALL WORK1(I, N)
24 END DO
25 !$OMP END PARALLEL
26 END SUBROUTINE GOOD_NESTING2
```

Fortran

A.35 Restrictions on Nesting of Regions

The examples in this section illustrate the region nesting rules. For more information on region nesting, see Section 2.9 on page 87.

The following example is non-conforming because the inner and outer loop regions are closely nested:

C/C++

Example A.35.1c

```
35 void work(int i, int j) {}
36 void wrong1(int n)
```

```

1  {
2  #pragma omp parallel default(shared)
3  {
4  int i, j;
5  #pragma omp for
6  for (i=0; i<n; i++) {
7      /* incorrect nesting of loop regions */
8      #pragma omp for
9      for (j=0; j<n; j++)
10         work(i, j);
11     }
12 }
13 }

```

C/C++

Fortran

Example A.35.1f

```

17     SUBROUTINE WORK(I, J)
18     INTEGER I, J
19     END SUBROUTINE WORK

```

```

20     SUBROUTINE WRONG1(N)
21     INTEGER N

```

```

22     INTEGER I,J
23 !$OMP PARALLEL DEFAULT(SHARED)
24 !$OMP DO
25     DO I = 1, N
26 !$OMP DO ! incorrect nesting of loop regions
27     DO J = 1, N
28     CALL WORK(I,J)
29     END DO
30     END DO
31 !$OMP END PARALLEL
32     END SUBROUTINE WRONG1

```

Fortran

The following orphaned version of the preceding example is also non-conforming:

Example A.35.2c

```

37 void work1(int i, int n)
38 {
39     int j;
40     /* incorrect nesting of loop regions */
41     #pragma omp for
42     for (j=0; j<n; j++)
43         work(i, j);

```

C/C++

```

1      }
2      void wrong2(int n)
3      {
4          #pragma omp parallel default(shared)
5          {
6              int i;
7              #pragma omp for
8                  for (i=0; i<n; i++)
9                      work1(i, n);
10         }
11     }

```

C/C++

Fortran

Example A.35.2f

```

15          SUBROUTINE WORK1(I,N)
16          INTEGER I, N
17
18              INTEGER J
19          !$OMP DO          ! incorrect nesting of loop regions
20              DO J = 1, N
21                  CALL WORK(I,J)
22              END DO
23          END SUBROUTINE WORK1
24
25          SUBROUTINE WRONG2(N)
26          INTEGER N
27
28              INTEGER I
29          !$OMP PARALLEL DEFAULT(SHARED)
30          !$OMP DO
31              DO I = 1, N
32                  CALL WORK1(I,N)
33              END DO
34          !$OMP END PARALLEL
35          END SUBROUTINE WRONG2

```

Fortran

The following example is non-conforming because the loop and **single** regions are closely nested:

C/C++

Example A.35.3c

```

38      void wrong3(int n)
39      {

```

```

1      #pragma omp parallel default(shared)
2      {
3          int i;
4          #pragma omp for
5              for (i=0; i<n; i++) {
6              /* incorrect nesting of regions */
7                  #pragma omp single
8                      work(i, 0);
9              }
10         }
11     }

```

C/C++

Fortran

Example A.35.3f

```

15      SUBROUTINE WRONG3(N)
16      INTEGER N
17
18      INTEGER I
19      !$OMP PARALLEL DEFAULT(SHARED)
20      !$OMP DO
21          DO I = 1, N
22      !$OMP SINGLE ! incorrect nesting of regions
23          CALL WORK(I, 1)
24      !$OMP END SINGLE
25      END DO
26      !$OMP END PARALLEL
27      END SUBROUTINE WRONG3

```

Fortran

The following example is non-conforming because a **barrier** region cannot be closely nested inside a loop region:

Example A.35.4c

```

32      void wrong4(int n)
33      {
34
35          #pragma omp parallel default(shared)
36          {
37              int i;
38              #pragma omp for
39                  for (i=0; i<n; i++) {
40                  work(i, 0);
41              /* incorrect nesting of barrier region in a loop region */
42                  #pragma omp barrier

```

```

1         work(i, 1);
2     }
3 }
4 }

```

C/C++

Fortran

Example A.35.4f

```

8         SUBROUTINE WRONG4(N)
9         INTEGER N

10        INTEGER I
11        !$OMP PARALLEL DEFAULT(SHARED)
12        !$OMP DO
13            DO I = 1, N
14                CALL WORK(I, 1)
15                ! incorrect nesting of barrier region in a loop region
16        !$OMP BARRIER
17                CALL WORK(I, 2)
18            END DO
19        !$OMP END PARALLEL
20        END SUBROUTINE WRONG4

```

Fortran

The following example is non-conforming because the **barrier** region cannot be closely nested inside the **critical** region. If this were permitted, it would result in deadlock due to the fact that only one thread at a time can enter the **critical** region:

C/C++

Example A.35.5c

```

27 void wrong5(int n)
28 {
29     #pragma omp parallel
30     {
31         #pragma omp critical
32         {
33             work(n, 0);
34             /* incorrect nesting of barrier region in a critical region */
35             #pragma omp barrier
36             work(n, 1);
37         }
38     }
39 }

```

C/C++

Fortran

Example A.35.5f

```

SUBROUTINE WRONG5(N)
  INTEGER N

  !$OMP PARALLEL DEFAULT(SHARED)
  !$OMP CRITICAL
    CALL WORK(N,1)
  ! incorrect nesting of barrier region in a critical region
  !$OMP BARRIER
    CALL WORK(N,2)
  !$OMP END CRITICAL
  !$OMP END PARALLEL
END SUBROUTINE WRONG5

```

Fortran

The following example is non-conforming because the **barrier** region cannot be closely nested inside the **single** region. If this were permitted, it would result in deadlock due to the fact that only one thread executes the **single** region:

C/C++

Example A.35.6c

```

void wrong6(int n)
{
  #pragma omp parallel
  {
    #pragma omp single
    {
      work(n, 0);
    /* incorrect nesting of barrier region in a single region */
    #pragma omp barrier
      work(n, 1);
    }
  }
}

```

C/C++

Fortran

Example A.35.6f

```

SUBROUTINE WRONG6(N)
  INTEGER N

  !$OMP PARALLEL DEFAULT(SHARED)
  !$OMP SINGLE
    CALL WORK(N,1)
  ! incorrect nesting of barrier region in a single region

```



```

1      !$OMP      BARRIER
2              CALL WORK(N,2)
3      !$OMP      END SINGLE
4      !$OMP      END PARALLEL
5      END SUBROUTINE WRONG6

```

Fortran

A.36 The `omp_set_dynamic` and `omp_set_num_threads` Routines

Some programs rely on a fixed, prespecified number of threads to execute correctly. Because the default setting for the dynamic adjustment of the number of threads is implementation defined, such programs can choose to turn off the dynamic threads capability and set the number of threads explicitly to ensure portability. The following example shows how to do this using `omp_set_dynamic` (Section 3.2.7 on page 97), and `omp_set_num_threads` (Section 3.2.1 on page 91).

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

C/C++

Example A.36.1c

```

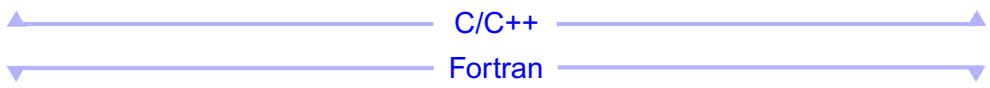
21      #include <omp.h>
22      #include <stdlib.h>
23
24      void do_by_16(float *x, int iam, int ipoints) {}
25
26      void a36(float *x, int npoints)
27      {
28          int iam, ipoints;
29
30          omp_set_dynamic(0);
31          omp_set_num_threads(16);
32
33          #pragma omp parallel shared(x, npoints) private(iam, ipoints)
34          {
35              if (omp_get_num_threads() != 16)
36                  abort();
37          }
38      }

```

```

1     iam = omp_get_thread_num();
2     ipoints = npoints/16;
3     do_by_16(x, iam, ipoints);
4 }
5 }

```



Example A.36.1f

```

9     SUBROUTINE DO_BY_16(X, IAM, IPOINETS)
10        REAL X(*)
11        INTEGER IAM, IPOINETS
12    END SUBROUTINE DO_BY_16

```

```

13    SUBROUTINE SUBA36(X, NPOINTS)

```

```

14        INCLUDE "omp_lib.h"      ! or USE OMP_LIB

```

```

15        INTEGER NPOINTS
16        REAL X(NPOINTS)

```

```

17        INTEGER IAM, IPOINETS

```

```

18        CALL OMP_SET_DYNAMIC(.FALSE.)
19        CALL OMP_SET_NUM_THREADS(16)

```

```

20    !$OMP PARALLEL SHARED(X,NPOINTS) PRIVATE(IAM, IPOINETS)

```

```

21        IF (OMP_GET_NUM_THREADS() .NE. 16) THEN
22            STOP
23        ENDIF

```

```

24        IAM = OMP_GET_THREAD_NUM()
25        IPOINETS = NPOINTS/16
26        CALL DO_BY_16(X,IAM,IPOINETS)

```

```

27    !$OMP END PARALLEL

```

```

28    END SUBROUTINE SUBA36

```



A.37 The `omp_get_num_threads` Routine

In the following example, the `omp_get_num_threads` call (see Section 3.2.2 on page 93) returns 1 in the sequential part of the code, so `np` will always be equal to 1. To determine the number of threads that will be deployed for the `parallel` region, the call should be inside the `parallel` region.

C/C++

Example A.37.1c

```
#include <omp.h>
void work(int i);

void incorrect()
{
    int np, i;

    np = omp_get_num_threads(); /* misplaced */

    #pragma omp parallel for schedule(static)
    for (i=0; i < np; i++)
        work(i);
}
```

C/C++

Fortran

Example A.37.1f

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

SUBROUTINE INCORRECT()
    INCLUDE "omp_lib.h" ! or USE OMP_LIB
    INTEGER I, NP

    NP = OMP_GET_NUM_THREADS() !misplaced: will return 1
!$OMP PARALLEL DO SCHEDULE(STATIC)
    DO I = 0, NP-1
        CALL WORK(I)
    ENDDO
!$OMP END PARALLEL DO
END SUBROUTINE INCORRECT
```

Fortran

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

C/C++

3 Example A.37.2c

```
4 #include <omp.h>
5 void work(int i);
6
7 void correct()
8 {
9     int i;
10
11     #pragma omp parallel private(i)
12     {
13         i = omp_get_thread_num();
14         work(i);
15     }
16 }
```

C/C++

Fortran

17 Example A.37.2f

```
18
19     SUBROUTINE WORK(I)
20         INTEGER I
21
22         I = I + 1
23
24     END SUBROUTINE WORK
25
26     SUBROUTINE CORRECT()
27         INCLUDE "omp_lib.h"      ! or USE OMP_LIB
28         INTEGER I
29
30     !$OMP    PARALLEL PRIVATE(I)
31             I = OMP_GET_THREAD_NUM()
32             CALL WORK(I)
33     !$OMP    END PARALLEL
34
35     END SUBROUTINE CORRECT
```

Fortran

A.38 The `omp_init_lock` Routine

The following example demonstrates how to initialize an array of locks in a `parallel` region by using `omp_init_lock` (Section 3.3.1 on page 104).

C/C++

Example A.38.1c

```
#include <omp.h>

omp_lock_t *new_locks()
{
    int i;
    omp_lock_t *lock = new omp_lock_t[1000];

    #pragma omp parallel for private(i)
        for (i=0; i<1000; i++)
        {
            omp_init_lock(&lock[i]);
        }
    return lock;
}
```

C/C++

Fortran

Example A.38.1f

```
FUNCTION NEW_LOCKS()
    USE OMP_LIB          ! or INCLUDE "omp_lib.h"
    INTEGER(OMP_LOCK_KIND), DIMENSION(1000) :: NEW_LOCKS

    INTEGER I

    !$OMP PARALLEL DO PRIVATE(I)
        DO I=1,1000
            CALL OMP_INIT_LOCK(NEW_LOCKS(I))
        END DO
    !$OMP END PARALLEL DO

    END FUNCTION NEW_LOCKS
```

Fortran

A.39 Simple Lock Routines

In the following example (for Section 3.3 on page 102), the lock routines cause the threads to be idle while waiting for entry to the first critical section, but to do other work while waiting for entry to the second. The `omp_set_lock` function blocks, but the `omp_test_lock` function does not, allowing the work in `skip` to be done.

C/C++

Note that the argument to the lock routines should have type `omp_lock_t`, and that there is no need to flush it.

Example A.39.1c

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

void skip(int i) {}
void work(int i) {}

int main()
{
    omp_lock_t lck;
    int id;

    omp_init_lock(&lck);

    #pragma omp parallel shared(lck) private(id)
    {
        id = omp_get_thread_num();

        omp_set_lock(&lck);
        /* only one thread at a time can execute this printf */
        printf("My thread id is %d.\n", id);
        omp_unset_lock(&lck);

        while (! omp_test_lock(&lck)) {
            skip(id); /* we do not yet have the lock,
                       so we must do something else */
        }

        work(id); /* we now have the lock
                  and can do the work */

        omp_unset_lock(&lck);
    }
}
```

```

1      omp_destroy_lock(&lck);
2
3      return 0;
4  }

```

C/C++

Fortran

Note that there is no need to flush the lock variable.

Example A.39.1f

```

8      SUBROUTINE SKIP(ID)
9      END SUBROUTINE SKIP

10     SUBROUTINE WORK(ID)
11     END SUBROUTINE WORK

12     PROGRAM A39

13         INCLUDE "omp_lib.h"      ! or USE OMP_LIB

14         INTEGER(OMP_LOCK_KIND) LCK
15         INTEGER ID

16         CALL OMP_INIT_LOCK(LCK)

17     !$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
18         ID = OMP_GET_THREAD_NUM()
19         CALL OMP_SET_LOCK(LCK)
20         PRINT *, 'My thread id is ', ID
21         CALL OMP_UNSET_LOCK(LCK)

22         DO WHILE (.NOT. OMP_TEST_LOCK(LCK))
23             CALL SKIP(ID)      ! We do not yet have the lock
24                                 ! so we must do something else
25         END DO

26         CALL WORK(ID)         ! We now have the lock
27                                 ! and can do the work

28         CALL OMP_UNSET_LOCK( LCK )

29     !$OMP END PARALLEL

30         CALL OMP_DESTROY_LOCK( LCK )

```

A.40 Nestable Lock Routines

The following example (for Section 3.3 on page 102) demonstrates how a nestable lock can be used to synchronize updates both to a whole structure and to one of its members.

C/C++

Example A.40.1c

```
#include <omp.h>

typedef struct {
    int a,b;
    omp_nest_lock_t lck; } pair;

int work1();
int work2();
int work3();

void incr_a(pair *p, int a)
{
    /* Called only from incr_pair, no need to lock. */
    p->a += a;
}

void incr_b(pair *p, int b)
{
    /* Called both from incr_pair and elsewhere, */
    /* so need a nestable lock. */

    omp_set_nest_lock(&p->lck);
    p->b += b;
    omp_unset_nest_lock(&p->lck);
}

void incr_pair(pair *p, int a, int b)
{
    omp_set_nest_lock(&p->lck);
    incr_a(p, a);
    incr_b(p, b);
    omp_unset_nest_lock(&p->lck);
}

void a40(pair *p)
{
```



```

1      #pragma omp parallel sections
2      {
3          #pragma omp section
4              incr_pair(p, work1(), work2());
5          #pragma omp section
6              incr_b(p, work3());
7      }
8  }

```

▲ C/C++ ▲

▼ Fortran ▼

Example A.40.1f

```

12      MODULE DATA
13          USE OMP_LIB, ONLY: OMP_NEST_LOCK_KIND
14          TYPE LOCKED_PAIR
15              INTEGER A
16              INTEGER B
17              INTEGER (OMP_NEST_LOCK_KIND) LCK
18          END TYPE
19      END MODULE DATA

20      SUBROUTINE INCR_A(P, A)
21          ! called only from INCR_PAIR, no need to lock
22          USE DATA
23          TYPE(LOCKED_PAIR) :: P
24          INTEGER A
25          P%A = P%A + A
26      END SUBROUTINE INCR_A

27      SUBROUTINE INCR_B(P, B)
28          ! called from both INCR_PAIR and elsewhere,
29          ! so we need a nestable lock
30          USE OMP_LIB          ! or INCLUDE "omp_lib.h"
31          USE DATA
32          TYPE(LOCKED_PAIR) :: P
33          INTEGER B
34          CALL OMP_SET_NEST_LOCK(P%LCK)
35          P%B = P%B + B
36          CALL OMP_UNSET_NEST_LOCK(P%LCK)
37      END SUBROUTINE INCR_B

38      SUBROUTINE INCR_PAIR(P, A, B)
39          USE OMP_LIB          ! or INCLUDE "omp_lib.h"
40          USE DATA
41          TYPE(LOCKED_PAIR) :: P
42          INTEGER A
43          INTEGER B

```

```
1          CALL OMP_SET_NEST_LOCK(P%LCK)
2          CALL INCR_A(P, A)
3          CALL INCR_B(P, B)
4          CALL OMP_UNSET_NEST_LOCK(P%LCK)
5      END SUBROUTINE INCR_PAIR

6      SUBROUTINE A40(P)
7          USE OMP_LIB          ! or INCLUDE "omp_lib.h"
8          USE DATA
9          TYPE(LOCKED_PAIR) :: P
10         INTEGER WORK1, WORK2, WORK3
11         EXTERNAL WORK1, WORK2, WORK3

12     !$OMP PARALLEL SECTIONS
13     !$OMP SECTION
14         CALL INCR_PAIR(P, WORK1(), WORK2())
15     !$OMP SECTION
16         CALL INCR_B(P, WORK3())
17     !$OMP END PARALLEL SECTIONS

18     END SUBROUTINE A40
```

▲————— Fortran —————▲

2 Stubs for Runtime Library

3 Routines

4 This section provides stubs for the runtime library routines defined in the OpenMP API.
5 The stubs are provided to enable portability to platforms that do not support the
6 OpenMP API. On these platforms, OpenMP programs must be linked with a library
7 containing these stub routines. The stub routines assume that the directives in the
8 OpenMP program are ignored. As such, they emulate serial semantics.

9 Note that the lock variable that appears in the lock routines must be accessed exclusively
10 through these routines. It should not be initialized or otherwise modified in the user
11 program.

12 Fortran

13 For the stub routines written in Fortran, the lock variable is declared as a **POINTER** to
14 guarantee that it is capable of holding an address. Alternatively, for Fortran 90
15 implementations, it could be declared as an **INTEGER(OMP_LOCK_KIND)** or
16 **INTEGER(OMP_NEST_LOCK_KIND)**, as appropriate.

17 Fortran

18 In an actual implementation the lock variable might be used to hold the address of an
19 allocated object, but here it is used to hold an integer value. Users should not make
20 assumptions about mechanisms used by OpenMP implementations to implement locks
21 based on the scheme used by the stub procedures.

B.1 C/C++ Stub routines

```
1
2 #include <stdio.h>
3 #include <stdlib.h>
4 #include "omp.h"
5
6 #ifdef __cplusplus
7 extern "C" {
8 #endif
9
10 void omp_set_num_threads(int num_threads)
11 {
12 }
13
14 int omp_get_num_threads(void)
15 {
16     return 1;
17 }
18
19 int omp_get_max_threads(void)
20 {
21     return 1;
22 }
23
24 int omp_get_thread_num(void)
25 {
26     return 0;
27 }
28
29 int omp_get_num_procs(void)
30 {
31     return 1;
32 }
33
34 void omp_set_dynamic(int dynamic_threads)
35 {
36 }
```

1
2
3
4

5
6
7
8

9
10
11

12
13
14
15

16

17
18
19
20

21
22
23
24

```
int omp_get_dynamic(void)
{
    return 0;
}

int omp_in_parallel(void)
{
    return 0;
}

void omp_set_nested(int nested)
{
}

int omp_get_nested(void)
{
    return 0;
}

enum {UNLOCKED = -1, INIT, LOCKED};

void omp_init_lock(omp_lock_t *lock)
{
    *lock = UNLOCKED;
}

void omp_destroy_lock(omp_lock_t *lock)
{
    *lock = INIT;
}
```

```

1 void omp_set_lock(omp_lock_t *lock)
2 {
3     if (*lock == UNLOCKED) {
4         *lock = LOCKED;
5     } else if (*lock == LOCKED) {
6         fprintf(stderr, "error: deadlock in using lock variable\n");
7         exit(1);
8     } else {
9         fprintf(stderr, "error: lock not initialized\n");
10        exit(1);
11    }
12 }

13 void omp_unset_lock(omp_lock_t *lock)
14 {
15     if (*lock == LOCKED) {
16         *lock = UNLOCKED;
17     } else if (*lock == UNLOCKED) {
18         fprintf(stderr, "error: lock not set\n");
19         exit(1);
20     } else {
21         fprintf(stderr, "error: lock not initialized\n");
22         exit(1);
23     }
24 }

25 int omp_test_lock(omp_lock_t *lock)
26 {
27     if (*lock == UNLOCKED) {
28         *lock = LOCKED;
29         return 1;
30     } else if (*lock == LOCKED) {
31         return 0;
32     } else {
33         fprintf(stderr, "error: lock not initialized\n");
34         exit(1);
35     }
36 }

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29

```
#ifndef OMP_NEST_LOCK_T

typedef struct { /* This really belongs in omp.h */
    int owner;
    int count;
} omp_nest_lock_t;

#endif

enum {NOOWNER=-1, MASTER = 0};

void omp_init_nest_lock(omp_nest_lock_t *nlock)
{
    nlock->owner = NOOWNER;
    nlock->count = 0;
}

void omp_destroy_nest_lock(omp_nest_lock_t *nlock)
{
    nlock->owner = NOOWNER;
    nlock->count = UNLOCKED;
}

void omp_set_nest_lock(omp_nest_lock_t *nlock)
{
    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);
    }
}
}
```

```

1 void omp_unset_nest_lock(omp_nest_lock_t *nlock)
2 {
3     if (nlock->owner == NOOWNER && nlock->count >= 1) {
4         nlock->count--;
5         if (nlock->count == 0) {
6             nlock->owner = NOOWNER;
7         }
8     } else if (nlock->owner == NOOWNER && nlock->count == 0) {
9         fprintf(stderr, "error: lock not set\n");
10        exit(1);
11    } else {
12        fprintf(stderr, "error: lock corrupted or not initialized\n");
13        exit(1);
14    }
15 }

16 int omp_test_nest_lock(omp_nest_lock_t *nlock)
17 {
18     omp_set_nest_lock(nlock);
19     return nlock->count;
20 }

21 double omp_get_wtime(void)
22 {
23     /* This function does not provide a working
24        wallclock timer. Replace it with a version
25        customized for the target machine.
26     */
27     return 0.0;
28 }

29 double omp_get_wtick(void)
30 {
31     /* This function does not provide a working
32        clock tick function. Replace it with
33        a version customized for the target machine.
34     */
35     return 365. * 86400.;
36 }

37 #ifdef __cplusplus
38 }
39 #endif

```

B.2 Fortran Stub Routines

```
1
2 SUBROUTINE OMP_SET_NUM_THREADS(NUM_THREADS)
3   INTEGER NUM_THREADS
4   RETURN
5 END SUBROUTINE
6
7 INTEGER FUNCTION OMP_GET_NUM_THREADS()
8   OMP_GET_NUM_THREADS = 1
9   RETURN
10  END FUNCTION
11
12 INTEGER FUNCTION OMP_GET_MAX_THREADS()
13   OMP_GET_MAX_THREADS = 1
14   RETURN
15  END FUNCTION
16
17 INTEGER FUNCTION OMP_GET_THREAD_NUM()
18   OMP_GET_THREAD_NUM = 0
19   RETURN
20  END FUNCTION
21
22 INTEGER FUNCTION OMP_GET_NUM_PROCS()
23   OMP_GET_NUM_PROCS = 1
24   RETURN
25  END FUNCTION
26
27 SUBROUTINE OMP_SET_DYNAMIC(DYNAMIC_THREADS)
28   LOGICAL DYNAMIC_THREADS
29   RETURN
30 END SUBROUTINE
```

1
2
3
4

5
6
7
8

9
10
11
12

13
14
15
16

17
18
19
20
21
22

23
24
25

26
27
28

29
30
31

```
LOGICAL FUNCTION OMP_GET_DYNAMIC()  
    OMP_GET_DYNAMIC = .FALSE.  
    RETURN  
END FUNCTION  
  
LOGICAL FUNCTION OMP_IN_PARALLEL()  
    OMP_IN_PARALLEL = .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_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  
    POINTER (LOCK,IL)  
    INTEGER IL  
  
    LOCK = -1  
    RETURN  
END SUBROUTINE  
  
SUBROUTINE OMP_DESTROY_LOCK(LOCK)  
    POINTER (LOCK,IL)  
    INTEGER IL  
  
    LOCK = 0  
    RETURN  
END SUBROUTINE
```

```

1  SUBROUTINE OMP_SET_LOCK(LOCK)
2  POINTER (LOCK,IL)
3  INTEGER IL

4  IF (LOCK .EQ. -1) THEN
5  LOCK = 1
6  ELSEIF (LOCK .EQ. 1) THEN
7  PRINT *, 'ERROR: DEADLOCK IN USING LOCK VARIABLE'
8  STOP
9  ELSE
10 PRINT *, 'ERROR: LOCK NOT INITIALIZED'
11 STOP
12 ENDIF

13 RETURN
14 END SUBROUTINE

15 SUBROUTINE OMP_UNSET_LOCK(LOCK)
16 POINTER (LOCK,IL)
17 INTEGER IL

18 IF (LOCK .EQ. 1) THEN
19 LOCK = -1
20 ELSEIF (LOCK .EQ. -1) THEN
21 PRINT *, 'ERROR: LOCK NOT SET'
22 STOP
23 ELSE
24 PRINT *, 'ERROR: LOCK NOT INITIALIZED'
25 STOP
26 ENDIF

27 RETURN
28 END SUBROUTINE

```

```

1 LOGICAL FUNCTION OMP_TEST_LOCK(LOCK)
2   POINTER (LOCK,IL)
3   INTEGER IL
4
5   IF (LOCK .EQ. -1) THEN
6     LOCK = 1
7     OMP_TEST_LOCK = .TRUE.
8   ELSEIF (LOCK .EQ. 1) THEN
9     OMP_TEST_LOCK = .FALSE.
10  ELSE
11    PRINT *, 'ERROR: LOCK NOT INITIALIZED'
12    STOP
13  ENDIF
14
15  RETURN
16 END FUNCTION
17
18 SUBROUTINE OMP_INIT_NEST_LOCK(NLOCK)
19   ! NLOCK is  0 if the nestable lock is not initialized
20   !           -1 if the nestable lock is initialized but not set
21   !           1 if the nestable lock is set
22   ! no use count is maintained
23   POINTER (NLOCK,NIL)
24   INTEGER NIL
25
26   NLOCK = -1
27
28   RETURN
29 END SUBROUTINE
30
31 SUBROUTINE OMP_DESTROY_NEST_LOCK(NLOCK)
32   POINTER (NLOCK,NIL)
33   INTEGER NIL
34
35   NLOCK = 0
36
37   RETURN
38 END SUBROUTINE

```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28

```
SUBROUTINE OMP_SET_NEST_LOCK(NLOCK)
  POINTER (NLOCK,NIL)
  INTEGER NIL

  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)
  POINTER (NLOCK,IL)
  INTEGER IL

  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
```

```

1  INTEGER FUNCTION OMP_TEST_NEST_LOCK(NLOCK)
2      POINTER (NLOCK,NIL)
3      INTEGER NIL
4
5      IF (NLOCK .EQ. -1) THEN
6          NLOCK = 1
7          OMP_TEST_NEST_LOCK = 1
8      ELSEIF (NLOCK .EQ. 1) THEN
9          OMP_TEST_NEST_LOCK = 0
10     ELSE
11         PRINT *, 'ERROR: NESTED LOCK NOT INITIALIZED'
12         STOP
13     ENDIF
14
15     RETURN
16 END SUBROUTINE
17
18 DOUBLE PRECISION FUNCTION OMP_GET_WTIME()
19     ! This function does not provide a working
20     ! wall clock timer. Replace it with a version
21     ! customized for the target machine.
22
23     OMP_WTIME = 0.0D0
24
25     RETURN
26 END FUNCTION
27
28 DOUBLE PRECISION FUNCTION OMP_GET_WTICK()
29     ! This function does not provide a working
30     ! clock tick function. Replace it with
31     ! a version customized for the target machine.
32     DOUBLE PRECISION ONE_YEAR
33     PARAMETER (ONE_YEAR=365.D0*86400.D0)
34
35     OMP_WTICK = ONE_YEAR
36
37     RETURN
38 END FUNCTION

```

2

OpenMP C and C++ Grammar

3

C.1 Notation

4 The grammar rules consist of the name for a non-terminal, followed by a colon,
5 followed by replacement alternatives on separate lines.6 The syntactic expression $term_{opt}$ indicates that the term is optional within the
7 replacement.8 The syntactic expression $term_{optseq}$ is equivalent to $term-seq_{opt}$ with the following
9 additional rules:10 *term-seq* :11 *term*12 *term-seq term*13 *term-seq , term*

1 C.2 Rules

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

5 ***/* in C++ (ISO/IEC 14882:1998) */***

6 *statement-seq:*

7 *statement*

8 *openmp-directive*

9 *statement-seq statement*

10 *statement-seq openmp-directive*

11 ***/* in C90 (ISO/IEC 9899:1990) */***

12 *statement-list:*

13 *statement*

14 *openmp-directive*

15 *statement-list statement*

16 *statement-list openmp-directive*

17 ***/* in C99 (ISO/IEC 9899:1999) */***

18 *block-item:*

19 *declaration*

20 *statement*

21 *openmp-directive*


```

1      statement:
2          /* standard statements */
3          openmp-construct
4      openmp-construct:
5          parallel-construct
6          for-construct
7          sections-construct
8          single-construct
9          parallel-for-construct
10         parallel-sections-construct
11         master-construct
12         critical-construct
13         atomic-construct
14         ordered-construct
15     openmp-directive:
16         barrier-directive
17         flush-directive
18     structured-block:
19         statement
20     parallel-construct:
21         parallel-directive structured-block
22     parallel-directive:
23         # pragma omp parallel parallel-clauseoptseq new-line
24     parallel-clause:
25         unique-parallel-clause
26         data-clause

```

1 *unique-parallel-clause:*

2 **if** (*expression*)

3 **num_threads** (*expression*)

4 *for-construct:*

5 *for-directive iteration-statement*

6 *for-directive:*

7 **# pragma omp for** *for-clause_{optseq} new-line*

8 *for-clause:*

9 *unique-for-clause*

10 *data-clause*

11 **nowait**

12 *unique-for-clause:*

13 **ordered**

14 **schedule** (*schedule-kind*)

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

16 *schedule-kind:*

17 **static**

18 **dynamic**

19 **guided**

20 **runtime**

21 *sections-construct:*

22 *sections-directive section-scope*

23 *sections-directive:*

24 **# pragma omp sections** *sections-clause_{optseq} new-line*

25 *sections-clause:*

26 *data-clause*

27 **nowait**

28 *section-scope:*

1 { *section-sequence* }

2 *section-sequence*:

3 *section-directive*_{opt} *structured-block*

4 *section-sequence* *section-directive* *structured-block*

5 *section-directive*:

6 **# pragma omp section** *new-line*

7 *single-construct*:

8 *single-directive* *structured-block*

9 *single-directive*:

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

11 *single-clause*:

12 *data-clause*

13 **nowait**

14 *parallel-for-construct*:

15 *parallel-for-directive* *iteration-statement*

16 *parallel-for-directive*:

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

18 *parallel-for-clause*:

19 *unique-parallel-clause*

20 *unique-for-clause*

21 *data-clause*

22 *parallel-sections-construct*:

23 *parallel-sections-directive* *section-scope*

24 *parallel-sections-directive*:

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

26 *parallel-sections-clause*:

27 *unique-parallel-clause*

28 *data-clause*

1 *master-construct:*

2 *master-directive structured-block*

3 *master-directive:*

4 **# pragma omp master** *new-line*

5 *critical-construct:*

6 *critical-directive structured-block*

7 *critical-directive:*

8 **# pragma omp critical** *region-phrase_{opt} new-line*

9 *region-phrase:*

10 (*identifier*)

11 *barrier-directive:*

12 **# pragma omp barrier** *new-line*

13 *atomic-construct:*

14 *atomic-directive expression-statement*

15 *atomic-directive:*

16 **# pragma omp atomic** *new-line*

17 *flush-directive:*

18 **# pragma omp flush** *flush-vars_{opt} new-line*

19 *flush-vars:*

20 (*variable-list*)

21 *ordered-construct:*

22 *ordered-directive structured-block*

23 *ordered-directive:*

24 **# pragma omp ordered** *new-line*

25 *declaration:*

26 **/* standard declarations */**

27 *threadprivate-directive*

28 *threadprivate-directive:*

```

1      # pragma omp threadprivate ( variable-list ) new-line
2  data-clause:
3      private ( variable-list )
4      copyprivate ( variable-list )
5      firstprivate ( variable-list )
6      lastprivate ( variable-list )
7      shared ( variable-list )
8      default ( shared )
9      default ( none )
10     reduction ( reduction-operator : variable-list )
11     copyin ( variable-list )
12  reduction-operator:
13     One of: + * - & ^ | && ||
14  /* in C */
15  variable-list:
16     identifier
17     variable-list , identifier
18  /* in C++ */
19  variable-list:
20     id-expression
21     variable-list , id-expression

```


Interface Declarations

This appendix gives examples of the C/C++ header file, the Fortran **include** file and Fortran 90 **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.

D.1 Example of the `omp.h` Header File

```
#ifndef _OMP_H_DEF
#define _OMP_H_DEF

/*
 * define the lock data types
 */
#ifndef _OMP_LOCK_T_DEF
#   define _OMP_LOCK_T_DEF
    typedef struct __omp_lock *omp_lock_t;
#endif

#ifndef _OMP_NEST_LOCK_T_DEF
#   define _OMP_NEST_LOCK_T_DEF
    typedef struct __omp_nest_lock *omp_nest_lock_t;
#endif
```

```

1  /*
2   * exported OpenMP functions
3   */
4  #ifdef __cplusplus
5  extern "C" {
6  #endif

7  #if defined(__stdc__) || defined(__STDC__) ||
8  defined(__cplusplus)
9      extern void    omp_set_num_threads(int num_threads);
10     extern int     omp_get_num_threads(void);
11     extern int     omp_get_max_threads(void);
12     extern int     omp_get_thread_num(void);
13     extern int     omp_get_num_procs(void);
14     extern int     omp_in_parallel(void);
15     extern void    omp_set_dynamic(int dynamic_threads);
16     extern int     omp_get_dynamic(void);
17     extern void    omp_set_nested(int nested);
18     extern int     omp_get_nested(void);

19     extern void    omp_init_lock(omp_lock_t *lock);
20     extern void    omp_destroy_lock(omp_lock_t *lock);
21     extern void    omp_set_lock(omp_lock_t *lock);
22     extern void    omp_unset_lock(omp_lock_t *lock);
23     extern int     omp_test_lock(omp_lock_t *lock);

24     extern void    omp_init_nest_lock(omp_nest_lock_t *lock);
25     extern void    omp_destroy_nest_lock(omp_nest_lock_t *lock);
26     extern void    omp_set_nest_lock(omp_nest_lock_t *lock);
27     extern void    omp_unset_nest_lock(omp_nest_lock_t *lock);
28     extern int     omp_test_nest_lock(omp_nest_lock_t *lock);

29     extern double  omp_get_wtime(void);
30     extern double  omp_get_wtick(void);

31 #else
32     extern void    omp_set_num_threads();
33     extern int     omp_get_num_threads();
34     extern int     omp_get_max_threads();
35     extern int     omp_get_thread_num();
36     extern int     omp_get_num_procs();
37     extern int     omp_in_parallel();
38     extern void    omp_set_dynamic();

```



```

1      extern int      omp_get_dynamic();
2      extern void     omp_set_nested();
3      extern int      omp_get_nested();

4      extern void     omp_init_lock();
5      extern void     omp_destroy_lock();
6      extern void     omp_set_lock();
7      extern void     omp_unset_lock();
8      extern int      omp_test_lock();

9      extern void     omp_init_nest_lock();
10     extern void     omp_destroy_nest_lock();
11     extern void     omp_set_nest_lock();
12     extern void     omp_unset_nest_lock();
13     extern int      omp_test_nest_lock();

14     extern double   omp_get_wtime();
15     extern double   omp_get_wtick();
16 #endif
17 #ifdef __cplusplus
18 }
19 #endif

20 #endif

```

D.2 Example of an Interface Declaration include File

```

23  C      the "C" of this comment starts in column 1
24         integer      omp_lock_kind
25         parameter ( omp_lock_kind = 8 )

26         integer      omp_nest_lock_kind
27         parameter ( omp_nest_lock_kind = 8 )

```

```

1      C          default integer type assumed below
2      C          default logical type assumed below
3      C          OpenMP Fortran API v2.5

4      integer    openmp_version
5      parameter ( openmp_version = 200505 )

6      external omp_destroy_lock

7      external omp_destroy_nest_lock

8      external omp_get_dynamic
9      logical    omp_get_dynamic

10     external omp_get_max_threads
11     integer    omp_get_max_threads

12     external omp_get_nested
13     logical    omp_get_nested

14     external omp_get_num_procs
15     integer    omp_get_num_procs

16     external omp_get_num_threads

17     integer    omp_get_num_threads

18     external omp_get_thread_num
19     integer    omp_get_thread_num

20     external omp_get_wtick
21     double precision omp_get_wtick

22     external omp_get_wtime
23     double precision omp_get_wtime

24     external omp_init_lock

25     external omp_init_nest_lock

26     external omp_in_parallel
27     logical    omp_in_parallel

28     external omp_set_dynamic

29     external omp_set_lock

```

```
1      external omp_set_nest_lock
2
3      external omp_set_nested
4
5      external omp_set_num_threads
6
7      external omp_test_lock
8      logical omp_test_lock
9
10     external omp_test_nest_lock
11     integer omp_test_nest_lock
12
13     external omp_unset_lock
14
15     external omp_unset_nest_lock
```

10 D.3 Example of a Fortran 90 Interface Declaration

11 module

```
12     ! the "!" of this comment starts in column 1
13
14     module omp_lib_kinds
15
16         integer, parameter :: omp_integer_kind = 4
17         integer, parameter :: omp_logical_kind = 4
18         integer, parameter :: omp_lock_kind = 8
19         integer, parameter :: omp_nest_lock_kind = 8
20
21     end module omp_lib_kinds
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34

```
module omp_lib

    use omp_lib_kinds

!           OpenMP Fortran API v2.5
integer, parameter :: openmp_version = 200505

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

interface
    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
end interface

interface
    function omp_get_dynamic ( )
    use omp_lib_kinds
    logical ( kind=omp_logical_kind ) :: omp_get_dynamic
    end function omp_get_dynamic
end interface

interface
    function omp_get_max_threads ( )
    use omp_lib_kinds
    integer ( kind=omp_integer_kind ) :: omp_get_max_threads
    end function omp_get_max_threads
end interface

interface
    function omp_get_nested ( )
    use omp_lib_kinds
    logical ( kind=omp_logical_kind ) :: omp_get_nested
    end function omp_get_nested
end interface
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40

```
interface
  function omp_get_num_procs ()
  use omp_lib_kinds
  integer ( kind=omp_integer_kind ) :: omp_get_num_procs
end function omp_get_num_procs
end interface

interface
  function omp_get_num_threads ()
  use omp_lib_kinds
  integer ( kind=omp_integer_kind ) :: omp_get_num_threads
end function omp_get_num_threads
end interface

interface
  function omp_get_thread_num ()
  use omp_lib_kinds
  integer ( kind=omp_integer_kind ) :: omp_get_thread_num
end function omp_get_thread_num
end interface

interface
  function omp_get_wtick ()
  double precision :: omp_get_wtick
end function omp_get_wtick
end interface

interface
  function omp_get_wtime ()
  double precision :: omp_get_wtime
end function omp_get_wtime
end interface

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

interface
  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
end interface
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32

```
interface
  function omp_in_parallel ( )
  use omp_lib_kinds
  logical ( kind=omp_logical_kind ) :: omp_in_parallel
end function omp_in_parallel
end interface

interface
  subroutine omp_set_dynamic ( enable_expr )
  use omp_lib_kinds
  logical ( kind=omp_logical_kind ), intent(in) :: &
&  enable_expr
  end subroutine omp_set_dynamic
end interface

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

interface
  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
end interface

interface
  subroutine omp_set_nested ( enable_expr )
  use omp_lib_kinds
  logical ( kind=omp_logical_kind ), intent(in) :: &
&  enable_expr
  end subroutine omp_set_nested
end interface
```

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34

```
interface
  subroutine omp_set_num_threads ( number_of_threads_expr )
    use omp_lib_kinds
    integer ( kind=omp_integer_kind ), intent(in) :: &
&      number_of_threads_expr
    end subroutine omp_set_num_threads
  end interface

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

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

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

interface
  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
  end interface

end module omp_lib
```

D.4 Example of a Generic Interface for a Library Routine

Any of the OMP runtime library routines that take an argument may be extended with a generic interface so arguments of different **KIND** type can be accommodated.

Assume an implementation supports both default **INTEGER** as **KIND = OMP_INTEGER_KIND** and another **INTEGER KIND, KIND = SHORT_INT**. Then **OMP_SET_NUM_THREADS** could be specified in the **omp_lib** module as the following:

```
! the "!" of this comment starts in column 1
interface omp_set_num_threads

    subroutine omp_set_num_threads_1 ( number_of_threads_expr )
        use omp_lib_kinds
        integer ( kind=omp_integer_kind ), intent(in) :: &
& number_of_threads_expr
    end subroutine omp_set_num_threads_1

    subroutine omp_set_num_threads_2 ( number_of_threads_expr )
        use omp_lib_kinds
        integer ( kind=short_int ), intent(in) :: &
& number_of_threads_expr
    end subroutine omp_set_num_threads_2

end interface omp_set_num_threads
```


Implementation Defined Behaviors in OpenMP

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.

- **Memory model:** it is implementation defined as to whether, and in what sizes, memory accesses by multiple threads to the same variable without synchronization are atomic with respect to each other (see Section 1.4.1 on page 10).
- **Internal control variables:** the number of copies of the internal control variables, and their effects, during the execution of any explicit parallel region are implementation defined. The initial values of *nthreads-var*, *dyn-var*, *run-sched-var*, and *def-sched-var* are implementation defined (see Section 2.3 on page 24).
- **Nested parallelism:** the number of levels of parallelism supported is implementation defined (see Section 1.2.4 on page 8 and Section 2.4.1 on page 29).
- **Dynamic adjustment of threads:** it is implementation defined whether the ability to dynamically adjust the number of threads is provided. Implementations are allowed to deliver fewer threads (but at least one) than indicated in Figure 2-1 in exceptional situations, such as when there is a lack of resources, even if dynamic adjustment is disabled. In these situations, the behavior of the program is implementation defined (see Section 2.4.1 on page 29).
- **sections construct:** the method of scheduling the structured blocks among threads in the team is implementation defined (see Section 2.5.2 on page 39).
- **single construct:** the method of choosing a thread to execute the structured block is implementation defined (see Section 2.5.3 on page 42).
- **atomic construct:** a compliant implementation may enforce exclusive access between **atomic** regions which update different storage locations. The circumstances under which this occurs are implementation defined (see Section 2.7.4 on page 55).

- 1 • **omp_set_num_threads routine**: when called from within any explicit
2 **parallel** region, the binding thread set (and binding region, if required) for the
3 **omp_set_num_threads** region is implementation defined. If the number of
4 threads requested exceeds the number the implementation can support, or is not a
5 positive integer, the behavior of this routine is implementation defined. If this routine
6 is called from within any explicit **parallel** region, the behavior of this routine is
7 implementation defined (see Section 3.2.1 on page 91).
- 8 • **omp_get_max_threads routine**: when called from within any explicit
9 **parallel** region, the binding thread set (and binding region, if required) for the
10 **omp_get_max_threads** region is implementation defined (see Section 3.2.3 on
11 page 94).
- 12 • **omp_set_dynamic routine**: when called from within any explicit **parallel**
13 region, the binding thread set (and binding region, if required) for the
14 **omp_set_dynamic** region is implementation defined. If called from within any
15 explicit **parallel** region, the behavior of this routine is implementation defined
16 (see Section 3.2.7 on page 97).
- 17 • **omp_get_dynamic routine**: when called from within any explicit **parallel**
18 region, the binding thread set (and binding region, if required) for the
19 **omp_get_dynamic** region is implementation defined (see Section 3.2.8 on page
20 99).
- 21 • **omp_set_nested routine**: when called from within any explicit **parallel**
22 region, the binding thread set (and binding region, if required) for the
23 **omp_set_nested** region is implementation defined. If called from within any
24 explicit **parallel** region, the behavior of this routine is implementation defined
25 (see Section 3.2.9 on page 100).
- 26 • **omp_get_nested routine**: when called from within any explicit **parallel**
27 region, the binding thread set (and binding region, if required) for the
28 **omp_get_nested** region is implementation defined (see Section 3.2.10 on page
29 101).
- 30 • **OMP_NUM_THREADS environment variable**: if the requested value of
31 **OMP_NUM_THREADS** is greater than the number of threads an implementation can
32 support, or if the value is not a positive integer, the behavior of the program is
33 implementation defined (see Section 4.2 on page 115).

▼ Fortran ▼

- 35 • **threadprivate directive**: if the conditions for values of data in the threadprivate
36 objects of threads (other than the initial thread) to persist between two consecutive
37 active **parallel** regions do not all hold, the allocation status of an allocatable array
38 in the second region is implementation defined (see Section 2.8.2 on page 66).

- **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 2.8.3.2 on page 72).
- **Runtime library definitions:** it is implementation defined whether the `include` file `omp_lib.h` or the `module` file `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 90).

Fortran

Changes from Version 2.0 to Version 2.5

This appendix summarizes the major changes between the OpenMP API Version 2.0 specifications and the OpenMP API Version 2.5 specification. There are no additional language features in Version 2.5. However, large parts of the text have been rewritten and restructured in order to accommodate all the base languages in a single document, and a number of inconsistencies have been resolved. Only the major changes are listed here.

Terminology

Many terms have been redefined, and the Glossary (Section 1.2 on page 2) has been significantly extended. In particular, readers should note the following changes:

- The Glossary contains new definitions of *construct* and *region*. The terms *lexical extent* and *dynamic extent* are no longer used.
- The term *parallel region* is no longer used. Instead, the terms **parallel construct** and **parallel region** are used, as appropriate.
- The term *serial region* is no longer used: this has been replaced with *sequential part*.
- The Glossary defines binding in terms of *binding thread set* and *binding region*.
- The term *serialize* is no longer used. The terms *active* and *inactive* **parallel region** are used instead.
- The definition of *variable* differs from the previous definitions.
- The Glossary defines what is meant by *supporting nested parallelism*.

Memory model

Version 2.5 contains a description of the OpenMP memory model (see Section 1.4 on page 10). This describes the underlying consistency model assumed by OpenMP, and defines the *flush* operation. It also describes the circumstances under which it is

1 permitted for one thread to access private variables belonging to another thread. The
2 memory model states that a race condition on a shared variable results in unspecified
3 behavior. The relationship between the flush operation and the **volatile** keyword in
4 the C and C++ languages is explained.

5 Fortran

6 **PURE and ELEMENTAL Procedures**

7 OpenMP directives and runtime library routine calls may not appear in **PURE** or
8 **ELEMENTAL** procedures.

9 Fortran

10 **Internal Control Variables**

11 Version 2.5 introduces the notion of internal control variables (see Section 2.3 on page
12 24), that store the information for determining the number of threads to use for a
13 **parallel** region and how to schedule a work-sharing loop. The behavior of certain
14 execution environment routines (see Section 3.2 on page 91) and environment variables
15 (see Chapter 4) is described in terms of the internal control variables.

16 **Determining the Number of Threads for a parallel Region**

17 The rules which determine the number of threads to use for a **parallel** region have
18 been clarified. See Section 2.4.1 on page 29.

19 **Loop Construct**

20 The definition of the **guided** schedule kind has been relaxed: the size of each chunk is
21 proportional to the number of unassigned iterations divided by the number of threads.
22 See Section 2.5.1 on page 33.

23 **sections Construct**

24 The method of scheduling the structured blocks among threads in the team is
25 implementation defined in Version 2.5. See Section 2.5.2 on page 39.

26 **single Construct**

27 The method of choosing a thread to execute the structured block is implementation
28 defined in Version 2.5. See Section 2.5.3 on page 42.

critical Construct

The term *critical section* is no longer used. Instead, the terms **critical** construct and **critical** region are used, as appropriate.

flush Construct

In Version 2.5 it is stated that the flush operation does not imply any ordering between itself and operations on variables not in the flush-set, nor does it imply any ordering between two or more **flush** constructs if the intersection of their flush-sets is empty (see Section 2.7.5 on page 58). Such implied orderings were assumed in Version 2.0, and as a result, the examples in Section A.18 on page 147 in Version 2.5 differ from the equivalent examples in Version 2.0.

In Version 2.0 no flush operation was implied by calls to OpenMP lock routines. In Version 2.5 a flush without a list is implied whenever a lock is set or unset. See Section 2.7.5 on page 58 and Section 3.3 on page 102.

ordered Construct

The description of the **ordered** construct has been modified to account for the case where not every iteration of the loop encounters an **ordered** region.

Sharing Attribute Rules

Version 2.5 clarifies the rules which determine the sharing attributes of variable. See Section 2.8.1 on page 63.

threadprivate Directive

Version 2.5 clarifies the circumstances under which the values of data in the threadprivate objects of threads other than the initial thread are guaranteed to persist between two consecutive active **parallel** regions. See Section 2.8.2 on page 66.

Fortran

private Clause

Version 2.5 confirms that variables that appear in expressions for statement function definitions may not appear in a **private** clause. Section 2.8.3.3 on page 73.

Fortran

Private Arrays

The behavior of arrays which appear in **private**, **firstprivate** and **lastprivate** clauses has been clarified. See Section 2.8.3.3 on page 73, Section 2.8.3.4 on page 75 and Section 2.8.3.5 on page 77.

reduction Clause

Fortran pointers and Cray pointers are not permitted in a **reduction** clause. This restriction was omitted in Version 2.0.

Data Copying Clauses

In Version 2.5, the **copyin** and **copyprivate** clauses are no longer considered data-sharing attribute clauses, but are described as data copying clauses.

Nesting of Regions

The rules governing the nesting of regions are described using the concept of *closely nested* regions. See Section 2.9 on page 87.

Execution Environment Routines

In Version 2.0, the behavior of the **omp_set_num_threads**, **omp_set_dynamic** and **omp_set_nested** routines was undefined if called from an explicit **parallel** region. In Version 2.5, their behavior in this case is implementation defined. See Section 3.2.1 on page 91, Section 3.2.7 on page 97 and Section 3.2.9 on page 100.

Examples

The examples in Appendix A have been extended, corrected and reordered in Version 2.5. Where appropriate, equivalent examples have been provided for C/C++ and Fortran.

Except for examples illustrating non-conforming programs, all the examples consist of compilable program units.

Interface Declarations

An example of the `omp.h` header file has been included in Version 2.5.

Using the `schedule` Clause

This material, which appeared in Appendix D in OpenMP C/C++ Version 2.0 and in Appendix C in OpenMP Fortran Version 2.0, has been removed.

