

# OpenMP Application Program Interface

**DRAFT Version 3.1.2011-0203a**  
***THIS IS A DRAFT AND NOT FOR PUBLICATION***

Copyright © 1997-2011 OpenMP Architecture Review Board.  
Permission to copy without fee all or part of this material is granted,  
provided the OpenMP Architecture Review Board copyright notice and  
the title of this document appear. Notice is given that copying is by  
permission of OpenMP Architecture Review Board.

*THIS DRAFT IS 2011.0203a which includes a correction of ticket 14, and new ticket 94, and an updated Appendix F.*

*PREVIOUS DRAFTS*

*2011.0130 integrated tickets 91, 92, 93*

*Draft 1113:*

*Tickets integrated: 57, 58, 83, 84, 85, 86, 89:*

*Tickets integrated in draft (0706): 52, 53, 70 (Corrected), 81*

*Tickets integrated in draft (0621):*

*47(corrected), 54 (corrected), 73 (corrected), 77 (corrected), 79*

*Tickets integrated in draft (0609):*

*17,21,32,35,45,47,48,63,74,75,76,77,78,80*

<b>1. Introduction</b>	<b>1</b>
1.1 Scope	1
1.2 Glossary	2
1.2.1 Threading Concepts	2
1.2.2 OpenMP language terminology	2
1.2.3 Tasking Terminology	8
1.2.4 Data Terminology	9
1.2.5 Implementation Terminology	11
1.3 Execution Model	12
1.4 Memory Model	13
1.4.1 Structure of the OpenMP Memory Model	13
1.4.2 The Flush Operation	15
1.4.3 OpenMP Memory Consistency	16
1.5 OpenMP Compliance	17
1.6 Normative References	17
1.7 Organization of this document	18
<b>2. Directives</b>	<b>21</b>
2.1 Directive Format	22
2.1.1 Fixed Source Form Directives	23
2.1.2 Free Source Form Directives	24
2.2 Conditional Compilation	26
2.2.1 Fixed Source Form Conditional Compilation Sentinels	26
2.2.2 Free Source Form Conditional Compilation Sentinel	27
2.3 Internal Control Variables	28
2.3.1 ICV Descriptions	28
2.3.2 Modifying and Retrieving ICV Values	29
2.3.3 How the Per-Data Environment ICVs Work	30
2.3.4 ICV Override Relationships	31
2.4 <b>parallel</b> Construct	32

2.4.1	Determining the Number of Threads for a <code>parallel</code> Region	36
2.5	Worksharing Constructs	37
2.5.1	Loop Construct	38
2.5.1.1	Determining the Schedule of a Worksharing Loop	46
2.5.2	<code>sections</code> Construct	47
2.5.3	<code>single</code> Construct	49
2.5.4	<code>workshare</code> Construct	51
2.6	Combined Parallel Worksharing Constructs	54
2.6.1	Parallel Loop construct	54
2.6.2	<code>parallel sections</code> Construct	56
2.6.3	<code>parallel workshare</code> Construct	58
2.7	Tasking Constructs	59
2.7.1	<code>task</code> Construct	59
2.7.2	<code>taskyield</code> Construct	62
2.7.3	Task Scheduling	63
2.8	Master and Synchronization Constructs	65
2.8.1	<code>master</code> Construct	65
2.8.2	<code>critical</code> Construct	67
2.8.3	<code>barrier</code> Construct	68
2.8.4	<code>taskwait</code> Construct	70
2.8.5	<code>atomic</code> Construct	71
2.8.6	<code>flush</code> Construct	76
2.8.7	<code>ordered</code> Construct	81
2.9	Data Environment	82
2.9.1	Data-sharing Attribute Rules	83
2.9.1.1	Data-sharing Attribute Rules for Variables Referenced in a Construct	83
2.9.1.2	Data-sharing Attribute Rules for Variables Referenced in a Region but not in a Construct	85
2.9.2	<code>threadprivate</code> Directive	86
2.9.3	Data-Sharing Attribute Clauses	90

2.9.3.1	default clause	91
2.9.3.2	shared clause	93
2.9.3.3	private clause	94
2.9.3.4	firstprivate clause	97
2.9.3.5	lastprivate clause	99
2.9.3.6	reduction clause	101
2.9.4	Data Copying Clauses	105
2.9.4.1	copyin clause	106
2.9.4.2	copyprivate clause	107
2.10	Nesting of Regions	109
<b>3.</b>	<b>Runtime Library Routines</b>	<b>111</b>
3.1	Runtime Library Definitions	112
3.2	Execution Environment Routines	113
3.2.1	omp_set_num_threads	114
3.2.2	omp_get_num_threads	115
3.2.3	omp_get_max_threads	116
3.2.4	omp_get_thread_num	117
3.2.5	omp_get_num_procs	119
3.2.6	omp_in_parallel	120
3.2.7	omp_set_dynamic	121
3.2.8	omp_get_dynamic	122
3.2.9	omp_set_nested	123
3.2.10	omp_get_nested	124
3.2.11	omp_set_schedule	125
3.2.12	omp_get_schedule	127
3.2.13	omp_get_thread_limit	129
3.2.14	omp_set_max_active_levels	130
3.2.15	omp_get_max_active_levels	131
3.2.16	omp_get_level	133

3.2.17	<code>omp_get_ancestor_thread_num</code>	134
3.2.18	<code>omp_get_team_size</code>	135
3.2.19	<code>omp_get_active_level</code>	137
3.2.20	<code>omp_in_final</code>	138
3.3	Lock Routines	139
3.3.1	<code>omp_init_lock</code> and <code>omp_init_nest_lock</code>	141
3.3.2	<code>omp_destroy_lock</code> and <code>omp_destroy_nest_lock</code>	142
3.3.3	<code>omp_set_lock</code> and <code>omp_set_nest_lock</code>	143
3.3.4	<code>omp_unset_lock</code> and <code>omp_unset_nest_lock</code>	144
3.3.5	<code>omp_test_lock</code> and <code>omp_test_nest_lock</code>	145
3.4	Timing Routines	146
3.4.1	<code>omp_get_wtime</code>	147
3.4.2	<code>omp_get_wtick</code>	148
<b>4.</b>	<b>Environment Variables</b>	<b>151</b>
4.1	<code>OMP_SCHEDULE</code>	152
4.2	<code>OMP_NUM_THREADS</code>	153
4.3	<code>OMP_DYNAMIC</code>	154
4.4	<code>OMP_PROC_BIND</code>	154
4.5	<code>OMP_NESTED</code>	155
4.6	<code>OMP_STACKSIZE</code>	155
4.7	<code>OMP_WAIT_POLICY</code>	156
4.8	<code>OMP_MAX_ACTIVE_LEVELS</code>	157
4.9	<code>OMP_THREAD_LIMIT</code>	157
<b>A.</b>	<b>Examples</b>	<b>159</b>
A.1	A Simple Parallel Loop	159
A.2	The OpenMP Memory Model	160
A.3	Conditional Compilation	167
A.4	Internal Control Variables (ICVs)	168
A.5	The <code>parallel</code> Construct	170

A.6	Interaction Between the <code>num_threads</code> Clause and <code>omp_set_dynamic</code>	173
A.7	Fortran Restrictions on the <code>do</code> Construct	175
A.8	Fortran Private Loop Iteration Variables	177
A.9	The <code>nowait</code> clause	178
A.10	The <code>collapse</code> clause	181
A.11	The <code>parallel sections</code> Construct	185
A.12	The <code>single</code> Construct	186
A.13	Tasking Constructs	187
A.14	The <code>workshare</code> Construct	201
A.15	The <code>master</code> Construct	205
A.16	The <code>critical</code> Construct	207
A.17	worksharing Constructs Inside a <code>critical</code> Construct	209
A.18	Binding of <code>barrier</code> Regions	210
A.19	The <code>atomic</code> Construct	212
A.20	Restrictions on the <code>atomic</code> Construct	215
A.21	The <code>flush</code> Construct without a List	218
A.22	Placement of <code>flush</code> , <code>barrier</code> , and <code>taskwait</code> Directives	222
A.23	The <code>ordered</code> Clause and the <code>ordered</code> Construct	225
A.24	The <code>threadprivate</code> Directive	230
A.25	Parallel Random Access Iterator Loop	236
A.26	Fortran Restrictions on <code>shared</code> and <code>private</code> Clauses with Common Blocks	237
A.27	The <code>default(none)</code> Clause	239
A.28	Race Conditions Caused by Implied Copies of Shared Variables in Fortran	241
A.29	The <code>private</code> Clause	242
A.30	Fortran Restrictions on Storage Association with the <code>private</code> Clause	247
A.31	C/C++ Arrays in a <code>firstprivate</code> Clause	250
A.32	The <code>lastprivate</code> Clause	251

A.33	The <code>reduction</code> Clause	252
A.34	The <code>copyin</code> Clause	258
A.35	The <code>copyprivate</code> Clause	260
A.36	Nested Loop Constructs	265
A.37	Restrictions on Nesting of Regions	268
A.38	The <code>omp_set_dyn</code> . . . . . <code>amic</code> and <code>omp_set_num_threads</code> Routines	275
A.39	The <code>omp_get_num_threads</code> Routine	276
A.40	The <code>omp_init_lock</code> Routine	279
A.41	Ownership of Locks	280
A.42	Simple Lock Routines	281
A.43	Nestable Lock Routines	284
<b>B.</b>	<b>Stubs for Runtime Library Routines</b>	<b>287</b>
B.1	C/C++ Stub Routines	288
B.2	Fortran Stub Routines	294
<b>C.</b>	<b>OpenMP C and C++ Grammar</b>	<b>301</b>
C.1	Notation	301
C.2	Rules	302
<b>D.</b>	<b>Interface Declarations</b>	<b>311</b>
D.1	Example of the <code>omp.h</code> Header File	312
D.2	Example of an Interface Declaration <code>include</code> File	314
D.3	Example of a Fortran 90 Interface Declaration <code>module</code>	316
D.4	Example of a Generic Interface for a Library Routine	320
<b>E.</b>	<b>OpenMP Implementation-Defined Behaviors</b>	<b>321</b>
<b>F.</b>	<b>Features History</b>	<b>325</b>
F.1	Version 3.0 to 3.1 Differences	325
F.2	Version 2.5 to 3.0 Differences	326



# Introduction

1

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

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

10 `http://www.openmp.org`

11 The directives, library routines, and environment variables defined in this document  
12 allow users to create and manage parallel programs while permitting portability. The  
13 directives extend the C, C++ and Fortran base languages with single program multiple  
14 data (SPMD) constructs, tasking constructs, worksharing constructs, and  
15 synchronization constructs, and they provide support for sharing and privatizing data.  
16 The functionality to control the runtime environment is provided by library routines and  
17 environment variables. Compilers that support the OpenMP API often include a  
18 command line option to the compiler that activates and allows interpretation of all  
19 OpenMP directives.

---

## 20 1.1 Scope

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

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

---

## 4 **1.2 Glossary**

### 5 **1.2.1 Threading Concepts**

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

8 **OpenMP thread** A *thread* that is managed by the OpenMP runtime system.

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

### 11 **1.2.2 OpenMP language terminology**

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

14 COMMENT: See Section 1.6 on page 17 for a listing of current *base*  
15 *languages* for the OpenMP API.

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

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

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

21 COMMENTS:

22 For all base languages,

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

1		For C/C++:
2		• The point of entry must not be a call to <code>setjmp()</code> .
3		• <code>longjmp()</code> and <code>throw()</code> must not violate the entry/exit criteria.
4		• Calls to <code>exit()</code> are allowed in a <i>structured block</i> .
5		• An expression statement, iteration statement, selection statement,
6		or try block is considered to be a <i>structured block</i> if the
7		corresponding compound statement obtained by enclosing it in {
8		and } would be a <i>structured block</i> .
9		For Fortran:
10		• <code>STOP</code> statements are allowed in a <i>structured block</i> .
11	<b>enclosing context</b>	In C/C++, the innermost scope enclosing an OpenMP construct.
12		In Fortran, the innermost scoping unit enclosing an OpenMP construct.
13	<b>directive</b>	In C/C++, a <code>#pragma</code> , and in Fortran, a comment, that specifies <i>OpenMP</i>
14		<i>program</i> behavior.
15		COMMENT: See Section 2.1 on page 22 for a description of OpenMP
16		<i>directive</i> syntax.
17	<b>white space</b>	A non-empty sequence of space and/or horizontal tab characters.
18	<b>OpenMP program</b>	A program that consists of a <i>base program</i> , annotated with OpenMP <i>directives</i>
19		and runtime library routines.
20	<b>conforming program</b>	An <i>OpenMP program</i> that follows all the rules and restrictions of the
21		OpenMP specification.
22	<b>declarative directive</b>	An OpenMP <i>directive</i> that may only be placed in a declarative context. A
23		<i>declarative directive</i> has no associated executable user code, but instead has
24		one or more associated user declarations.
25		COMMENT: Only the <code>threadprivate</code> <i>directive</i> is a <i>declarative directive</i> .
26	<b>executable directive</b>	An OpenMP <i>directive</i> that is not declarative; i.e., it may be placed in an
27		executable context.
28		COMMENT: All <i>directives</i> except the <code>threadprivate</code> <i>directive</i> are
29		<i>executable directives</i> .
30	<b>stand-alone directive</b>	An OpenMP <i>executable directive</i> that has no associated executable user code.

1       **loop directive**   An OpenMP *executable directive* whose associated user code must be a loop  
2       nest that is a *structured block*.

3       COMMENTS:

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

5                       For Fortran, only the **do** *directive* and the optional **end do** *directive*  
6                       are *loop directives*.

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

8       COMMENT: If the *loop directive* contains a **collapse** clause then there  
9       may be more than one *associated loop*.

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

14       **region**   All code encountered during a specific instance of the execution of a given  
15       *construct* or of an OpenMP library routine. A *region* includes any code in  
16       called routines as well as any implicit code introduced by the OpenMP  
17       implementation. The generation of a *task* at the point where a **task** *directive*  
18       is encountered is a part of the *region* of the *encountering thread*, but the  
19       *explicit task region* associated with the **task** *directive* is not.

20       COMMENTS:

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

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

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

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

1	<b>sequential part</b>	All code encountered during the execution of an <i>OpenMP</i> program that is not part of a <b>parallel region</b> corresponding to a <b>parallel construct</b> or a <b>task region</b> corresponding to a <b>task construct</b> .
2		
3		
4		COMMENTS:
5		<i>The sequential part</i> executes as if it were enclosed by an <i>inactive parallel region</i> .
6		
7		Executable statements in called routines may be in both the <i>sequential part</i> and any number of explicit <b>parallel regions</b> at different points in the program execution.
8		
9		
10	<b>master thread</b>	The <i>thread</i> that encounters a <b>parallel construct</b> , creates a <i>team</i> , generates a set of <i>tasks</i> , then executes one of those <i>tasks</i> as <i>thread</i> number 0.
11		
12	<b>parent thread</b>	The <i>thread</i> that encountered the <b>parallel construct</b> and generated a <b>parallel region</b> is the <i>parent thread</i> of each of the <i>threads</i> in the <i>team</i> of that <b>parallel region</b> . The <i>master thread</i> of a <b>parallel region</b> is the same <i>thread</i> as its <i>parent thread</i> with respect to any resources associated with an <i>OpenMP thread</i> .
13		
14		
15		
16		
17	<b>ancestor thread</b>	For a given <i>thread</i> , its <i>parent thread</i> or one of its <i>parent thread's ancestor threads</i> .
18		
19	<b>team</b>	A set of one or more <i>threads</i> participating in the execution of a <b>parallel region</b> .
20		
21		COMMENTS:
22		For an <i>active parallel region</i> , the <i>team</i> comprises the <i>master thread</i> and at least one additional <i>thread</i> .
23		
24		For an <i>inactive parallel region</i> , the <i>team</i> comprises only the <i>master thread</i> .
25		
26	<b>initial thread</b>	The <i>thread</i> that executes the <i>sequential part</i> .
	<b>implicit parallel region</b>	The <i>inactive parallel region</i> that encloses the <i>sequential part</i> of an <i>OpenMP program</i> .
27		
28		
29	<b>nested construct</b>	A <i>construct</i> (lexically) enclosed by another <i>construct</i> .
30	<b>nested region</b>	A <i>region</i> (dynamically) enclosed by another <i>region</i> ; i.e., a <i>region</i> encountered during the execution of another <i>region</i> .
31		
32		COMMENT: Some nestings are <i>conforming</i> and some are not. See Section 2.10 on page 109 for the restrictions on nesting.
33		

1	<b>closely nested region</b>	A <i>region</i> nested inside another <i>region</i> with no <b>parallel</b> <i>region</i> nested
2		between them.
3	<b>all threads</b>	All OpenMP <i>threads</i> participating in the <i>OpenMP program</i> .
4	<b>current team</b>	All <i>threads</i> in the <i>team</i> executing the innermost enclosing <b>parallel</b> <i>region</i>
5	<b>encountering thread</b>	For a given <i>region</i> , the <i>thread</i> that encounters the corresponding <i>construct</i> .
6	<b>all tasks</b>	All <i>tasks</i> participating in the <i>OpenMP program</i> .
7	<b>current team tasks</b>	All <i>tasks</i> encountered during the execution of the innermost enclosing
8		<b>parallel</b> <i>region</i> by the <i>threads</i> of the corresponding <i>team</i> . Note that the
9		<i>implicit tasks</i> constituting the <b>parallel</b> <i>region</i> and any <i>descendant tasks</i>
10		encountered during the execution of these <i>implicit tasks</i> are included in this
11		<i>binding task set</i> .
12	<b>generating task</b>	For a given <i>region</i> the <i>task</i> whose execution by a <i>thread</i> generated the <i>region</i> .
13	<b>binding thread set</b>	The set of <i>threads</i> that are affected by, or provide the context for, the
14		execution of a <i>region</i> .
15		The <i>binding thread set</i> for a given <i>region</i> can be <i>all threads</i> , the <i>current team</i> ,
16		or the <i>encountering thread</i> .
17		COMMENT: The <i>binding thread set</i> for a particular <i>region</i> is described in its
18		corresponding subsection of this specification.
19	<b>binding task set</b>	The set of <i>tasks</i> that are affected by, or provide the context for, the execution
20		of a <i>region</i> .
21		The <i>binding task set</i> for a given <i>region</i> can be <i>all tasks</i> , the <i>current team</i>
22		<i>tasks</i> , or the <i>generating task</i> .
23		COMMENT: The <i>binding task set</i> for a particular <i>region</i> (if applicable) is
24		described in its corresponding subsection of this specification.

1	<b>binding region</b>	The enclosing <i>region</i> that determines the execution context and limits the scope of the effects of the bound <i>region</i> is called the <i>binding region</i> .
2		
3		<i>Binding region</i> is not defined for <i>regions</i> whose <i>binding thread set</i> is <i>all threads</i> or the <i>encountering thread</i> , nor is it defined for <i>regions</i> whose <i>binding task set</i> is <i>all tasks</i> .
4		
5		
6		COMMENTS:
7		The <i>binding region</i> for an <b>ordered</b> <i>region</i> is the innermost enclosing <i>loop region</i> .
8		
9		The <i>binding region</i> for a <b>taskwait</b> <i>region</i> is the innermost enclosing <i>task region</i> .
10		
11		For all other <i>regions</i> for which the <i>binding thread set</i> is the <i>current team</i> or the <i>binding task set</i> is the <i>current team tasks</i> , the <i>binding region</i> is the innermost enclosing <b>parallel</b> <i>region</i> .
12		
13		
14		For <i>regions</i> for which the <i>binding task set</i> is the generating <i>task</i> , the <i>binding region</i> is the <i>region</i> of the generating <i>task</i> .
15		
16		A <b>parallel</b> <i>region</i> need not be <i>active</i> nor explicit to be a <i>binding region</i> .
17		
18		A <i>task region</i> need not be explicit to be a <i>binding region</i> .
19		A <i>region</i> never binds to any <i>region</i> outside of the innermost enclosing <b>parallel</b> <i>region</i> .
20		
21	<b>orphaned construct</b>	A <i>construct</i> that gives rise to a <i>region</i> whose <i>binding thread set</i> is the <i>current team</i> , but that is not nested within another <i>construct</i> giving rise to the <i>binding region</i> .
22		
23		
24	<b>worksharing construct</b>	A <i>construct</i> that defines units of work, each of which is executed exactly once by one of the <i>threads</i> in the <i>team</i> executing the <i>construct</i> .
25		
26		For C, <i>worksharing constructs</i> are <b>for</b> , <b>sections</b> , and <b>single</b> .
27		For Fortran, <i>worksharing constructs</i> are <b>do</b> , <b>sections</b> , <b>single</b> and <b>workshare</b> .
28		
29	<b>sequential loop</b>	A loop that is not associated with any OpenMP <i>loop directive</i> .
30	<b>barrier</b>	A point in the execution of a program encountered by a <i>team</i> of <i>threads</i> , beyond which no <i>thread</i> in the team may execute until all <i>threads</i> in the <i>team</i> have reached the barrier and all <i>explicit tasks</i> generated by the <i>team</i> have executed to completion.
31		
32		
33		

## 1.2.3 Tasking Terminology

2	<b>task</b>	A specific instance of executable code and its data environment, generated when a <i>thread</i> encounters a <b>task construct</b> or a <b>parallel construct</b> .
4	<b>task region</b>	A <i>region</i> consisting of all code encountered during the execution of a <i>task</i> . COMMENT: A <b>parallel region</b> consists of one or more implicit <i>task regions</i> .
7	<b>explicit task</b>	A <i>task</i> generated when a <b>task construct</b> is encountered during execution.
8	<b>implicit task</b>	A <i>task</i> generated by the <i>implicit parallel region</i> or generated when a <b>parallel construct</b> is encountered during execution.
10	<b>initial task</b>	The <i>implicit task</i> associated with the <i>implicit parallel region</i> .
11	<b>current task</b>	For a given <i>thread</i> , the <i>task</i> corresponding to the <i>task region</i> in which it is executing.
13	<b>child task</b>	A <i>task</i> is a <i>child task</i> of the <i>region</i> of its generating <i>task</i> . A <i>child task region</i> is not part of its generating <i>task region</i> .
15	<b>descendant task</b>	A <i>task</i> that is the <i>child task</i> of a <i>task region</i> or of one of its <i>descendant task regions</i> .
17	<b>task completion</b>	<i>Task completion</i> occurs when the end of the <i>structured block</i> associated with the <i>construct</i> that generated the <i>task</i> is reached. COMMENT: Completion of the <i>initial task</i> occurs at program exit.
20	<b>task scheduling point</b>	A point during the execution of the current <i>task region</i> at which it can be suspended to be resumed later; or the point of <i>task completion</i> , after which the executing thread may switch to a different <i>task region</i> . COMMENT: Within tied task regions, task scheduling points only appear in the following: <ul style="list-style-type: none"><li>• encountered <b>task constructs</b></li><li>• encountered <b>taskyield constructs</b></li><li>• encountered <b>taskwait constructs</b></li><li>• encountered <b>barrier directives</b></li><li>• implicit <b>barrier regions</b></li><li>• at the end of the <i>tied task region</i></li></ul>
32	<b>task switching</b>	The act of a <i>thread</i> switching from the execution of one <i>task</i> to another <i>task</i> .



1	<b>tied task</b>	A <i>task</i> that, when its <i>task region</i> is suspended, can be resumed only by the same <i>thread</i> that suspended it; that is, the <i>task</i> is tied to that <i>thread</i> .
2		
3	<b>untied task</b>	A <i>task</i> that, when its <i>task region</i> is suspended, can be resumed by any <i>thread</i> in the team; that is, the <i>task</i> is not tied to any <i>thread</i> .
4		
5	<b>undelayed task</b>	A <i>task</i> for which execution is not deferred with respect to its generating <i>task region</i> ; that is, its generating <i>task region</i> is suspended until execution of the <i>undelayed task</i> is completed.
6		
7		
8	<b>included task</b>	A <i>task</i> for which execution is sequentially included in the generating <i>task region</i> ; that is, it is <i>undelayed</i> and executed immediately by the encountering thread.
9		
10		
11	<b>merged task</b>	A <i>task</i> whose data environment, inclusive of ICVs, is the same as that of its generating <i>task region</i> .
12		
13	<b>final task</b>	A <i>task</i> that forces all of its descendant tasks to become <i>included</i> tasks.
	<b>task synchronization construct</b>	
14		A <b>taskwait</b> or a <b>barrier construct</b> .

## 15 1.2.4 Data Terminology

16	<b>variable</b>	A named data storage block, whose value can be defined and redefined during the execution of a program.
17		
18		Array sections and substrings are not considered <i>variables</i> .
19	<b>private variable</b>	With respect to a given set of <i>task regions</i> that bind to the same <b>parallel region</b> , a <i>variable</i> whose name provides access to a different block of storage for each <i>task region</i> .
20		
21		
22		A <i>variable</i> that is part of another variable (as an array or structure element) cannot be made private independently of other components.
23		
24	<b>shared variable</b>	With respect to a given set of <i>task regions</i> that bind to the same <b>parallel region</b> , a <i>variable</i> whose name provides access to the same block of storage for each <i>task region</i> .
25		
26		
27		A <i>variable</i> that is part of another variable (as an array or structure element) cannot be <i>shared</i> independently of the other components, except for static data members of C++ classes.
28		
29		

1	<b>threadprivate</b>	
2	<b>variable</b>	A <i>variable</i> that is replicated, one instance per <i>thread</i> , by the OpenMP implementation, so that its name provides access to a different block of storage for each <i>thread</i> .
3		
4		A <i>variable</i> that is part of another variable (as an array or structure element) cannot be made <i>threadprivate</i> independently of the other components, except for static data members of C++ classes.
5		
6		
7	<b>threadprivate</b>	
	<b>memory</b>	The set of <i>threadprivate variables</i> associated with each <i>thread</i> .
8	<b>data environment</b>	All the variables associated with the execution of a given <i>task</i> . The <i>data environment</i> for a given <i>task</i> is constructed from the <i>data environment</i> of the <i>generating task</i> at the time the <i>task</i> is generated.
9		
10		
11	<b>defined</b>	For <i>variables</i> , the property of having a valid value.
12		For C:
13		For the contents of <i>variables</i> , the property of having a valid value.
14		For C++:
15		For the contents of <i>variables</i> of POD (plain old data) type, the property of having a valid value.
16		
17		For <i>variables</i> of non-POD class type, the property of having been constructed but not subsequently destructed.
18		
19		For Fortran:
20		For the contents of <i>variables</i> , the property of having a valid value. For the allocation or association status of <i>variables</i> , the property of having a valid status.
21		
22		
23		COMMENT: Programs that rely upon <i>variables</i> that are not <i>defined</i> are <i>non-conforming programs</i> .
24		
25	<b>class type</b>	For C++: Variables declared with one of the <b>class</b> , <b>struct</b> , or <b>union</b> keywords.

## 1.2.5 Implementation Terminology

2	<b>supporting <math>n</math> levels of parallelism</b>	Implies allowing an <i>active parallel region</i> to be enclosed by $n-1$ <i>active parallel regions</i> .
4	<b>supporting the OpenMP API</b>	Supporting at least one level of parallelism.
5	<b>supporting nested parallelism</b>	Supporting more than one level of parallelism.
6	<b>internal control variable</b>	A conceptual variable that specifies run-time behavior of a set of <i>threads</i> or <i>tasks</i> in an <i>OpenMP program</i> .
8		COMMENT: The acronym ICV is used interchangeably with the term <i>internal control variable</i> in the remainder of this specification.
10	<b>compliant implementation</b>	An implementation of the OpenMP specification that compiles and executes any <i>conforming program</i> as defined by the specification.
12		COMMENT: A <i>compliant implementation</i> may exhibit <i>unspecified behavior</i> when compiling or executing a <i>non-conforming program</i> .
14	<b>unspecified behavior</b>	A behavior or result that is not specified by the OpenMP specification or not known prior to the compilation or execution of an OpenMP program.
16		Such unspecified behavior may result from:
17		• Issues documented by the OpenMP specification as having <i>unspecified behavior</i> .
18		• A <i>non-conforming program</i> .
19		• A <i>conforming program</i> exhibiting an <i>implementation defined</i> behavior.
21	<b>implementation defined</b>	Behavior that must be documented by the implementation, and which is allowed to vary among different <i>compliant implementations</i> . An implementation is allowed to define this behavior as <i>unspecified</i> .
24		COMMENT: All features that have <i>implementation defined</i> behavior are documented in Appendix E.

---

## 1.3 Execution Model

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

An OpenMP program begins as a single thread of execution, called the initial thread. The initial thread executes sequentially, as if enclosed in an implicit task region, called the initial task region, that is defined by an implicit inactive **parallel** region surrounding the whole program.

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

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

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

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

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

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

---

## 23 **1.4       Memory Model**

### 24 **1.4.1     Structure of the OpenMP Memory Model**

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

1 A directive that accepts data-sharing attribute clauses determines two kinds of access to  
2 variables used in the directive's associated structured block: shared and private. Each  
3 variable referenced in the structured block has an original variable, which is the variable  
4 by the same name that exists in the program immediately outside the construct. Each  
5 reference to a shared variable in the structured block becomes a reference to the original  
6 variable. For each private variable referenced in the structured block, a new version of  
7 the original variable (of the same type and size) is created in memory for each task that  
8 contains code associated with the directive. Creation of the new version does not alter  
9 the value of the original variable. However, the impact of attempts to access the original  
10 variable during the region associated with the directive is unspecified; see  
11 Section 2.9.3.3 on page 94 for additional details. References to a private variable in the  
12 structured block refer to the current task's private version of the original variable. The  
13 relationship between the value of the original variable and the initial or final value of the  
14 private version depends on the exact clause that specifies it. Details of this issue, as well  
15 as other issues with privatization, are provided in Section 2.9 on page 82.

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

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

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

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

## 1 1.4.2 The Flush Operation

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

8 The flush operation is applied to a set of variables called the *flush-set*. The flush  
9 operation restricts reordering of memory operations that an implementation might  
10 otherwise do. Implementations must not reorder the code for a memory operation for a  
11 given variable, or the code for a flush operation for the variable, with respect to a flush  
12 operation that refers to the same variable.

13 If a thread has performed a write to its temporary view of a shared variable since its last  
14 flush of that variable, then when it executes another flush of the variable, the flush does  
15 not complete until the value of the variable has been written to the variable in memory.  
16 If a thread performs multiple writes to the same variable between two flushes of that  
17 variable, the flush ensures that the value of the last write is written to the variable in  
18 memory. A flush of a variable executed by a thread also causes its temporary view of the  
19 variable to be discarded, so that if its next memory operation for that variable is a read,  
20 then the thread will read from memory when it may again capture the value in the  
21 temporary view. When a thread executes a flush, no later memory operation by that  
22 thread for a variable involved in that flush is allowed to start until the flush completes.  
23 The completion of a flush of a set of variables executed by a thread is defined as the  
24 point at which all writes to those variables performed by the thread before the flush are  
25 visible in memory to all other threads and that thread's temporary view of all variables  
26 involved is discarded.

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

- 33 1. The value is written to the variable by the first thread.
- 34 2. The variable is flushed by the first thread.
- 35 3. The variable is flushed by the second thread.
- 36 4. The value is read from the variable by the second thread.

1 **Note** – OpenMP synchronization operations, described in Section 2.8 on page 65 and in  
2 Section 3.3 on page 139, are recommended for enforcing this order. Synchronization  
3 through variables is possible; however, it is not recommended since proper timing of  
4 flushes is difficult as shown in Section A.2 on page 160.

### 5 1.4.3 OpenMP Memory Consistency

6 The restrictions in Section 1.4.2 on page 15 on reordering with respect to flush  
7 operations guarantee the following:

- 8 • If the intersection of the flush-sets of two flushes performed by two different threads  
9 is non-empty, then the two flushes must be completed as if in some sequential order,  
10 seen by all threads.
- 11 • If two operations performed by the same thread either access, modify, or flush the  
12 same variable, then they must be completed as if in that thread's program order, as  
13 seen by all threads.
- 14 • If the intersection of the flush-sets of two flushes is empty, the threads can observe  
15 these flushes in any order.

16 The flush operation can be specified using the **flush** directive, and is also implied at  
17 various locations in an OpenMP program: see Section 2.8.6 on page 76 for details. For  
18 an example illustrating the memory model, see Section A.2 on page 160.

19 **Note** – Since flush operations by themselves cannot prevent data races, explicit flush  
20 operations are only useful in combination with atomic directives.

21 OpenMP programs that:

- 22 • do not use atomic directives,
- 23 • do not rely on the accuracy of a false result from **omp\_test\_lock** and  
24 **omp\_test\_nest\_lock**, and
- 25 • correctly avoid data races as required in Section 1.4.1 on page 13

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



1 Implementations are allowed to relax the ordering imposed by implicit flush operations  
2 when the result is only visible to programs using atomic directives.

---

## 3 **1.5 OpenMP Compliance**

4 An implementation of the OpenMP API is compliant if and only if it compiles and  
5 executes all conforming programs according to the syntax and semantics laid out in  
6 Chapters 1, 2, 3 and 4. Appendices A, B, C, D, E and F and sections designated as Notes  
7 (see Section 1.7 on page 18) are for information purposes only and are not part of the  
8 specification.

9 The OpenMP API defines constructs that operate in the context of the base language that  
10 is supported by an implementation. If the base language does not support a language  
11 construct that appears in this document, a compliant OpenMP implementation is not  
12 required to support it, with the exception that for Fortran, the implementation must  
13 allow case insensitivity for directive and API routines names, and must allow identifiers  
14 of more than six characters.

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

21 In both Fortran 90 and Fortran 95, variables with explicit initialization have the **SAVE**  
22 attribute implicitly. This is not the case in Fortran 77. However, a compliant OpenMP  
23 Fortran implementation must give such a variable the **SAVE** attribute, regardless of the  
24 underlying base language version.

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

---

## 28 **1.6 Normative References**

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

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

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

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

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

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

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

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

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

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

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

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

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

---

## 13 **1.7 Organization of this document**

14 The remainder of this document is structured as follows:

- 15 • Chapter 2: Directives
- 16 • Chapter 3: Runtime Library Routines
- 17 • Chapter 4: Environment Variables
- 18 • Appendix A: Examples
- 19 • Appendix B: Stubs for Runtime Library Routines
- 20 • Appendix C: OpenMP C and C++ Grammar
- 21 • Appendix D: Interface Declarations
- 22 • Appendix E: OpenMP Implementation-Defined Behaviors
- 23 • Appendix F: Features History

1 Some sections of this document only apply to programs written in a certain base  
2 language. Text that applies only to programs whose base language is C or C++ is shown  
3 as follows:

4  C/C++ specific text....

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

6  Fortran specific text.....

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

 Fortran (cont.)

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

11  **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 22)

- Mechanisms to control conditional compilation (Section 2.2 on page 26)

- Control of OpenMP API ICVs (Section 2.3 on page 28)

- Details of each OpenMP directive (Section 2.4 on page 32 to Section 2.10 on page 109)

## ▼ C/C++ ▲

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

## ▲ C/C++ ▼

## ▼ Fortran ▲

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

## ▲ Fortran ▼

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

## Restrictions

The following restriction applies to all OpenMP directives:

- OpenMP directives may not appear in **PURE** or **ELEMENTAL** procedures.

## 2.1 Directive Format

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

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 23 and Section 2.1.2 on page 24.

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

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

### Fortran

3 Only one *directive-name* can be specified per directive (note that this includes combined  
4 directives, see Section 2.6 on page 54). The order in which clauses appear on directives  
5 is not significant. Clauses on directives may be repeated as needed, subject to the  
6 restrictions listed in the description of each clause.

7 Some data-sharing attribute clauses (Section 2.9.3 on page 90), data copying clauses  
8 (Section 2.9.4 on page 105), the **threadprivate** directive (Section 2.9.2 on page 86)  
9 and the **flush** directive (Section 2.8.6 on page 76) accept a *list*. A *list* consists of a  
10 comma-separated collection of one or more *list items*.

### C/C++

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

### C/C++

### Fortran

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

### Fortran

### Fortran

## 16 2.1.1 Fixed Source Form Directives

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

<code>!\$omp</code>   <code>c\$omp</code>   <code>*\$omp</code>
---

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

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

```

1  Note – in the following example, the three formats for specifying the directive are
2  equivalent (the first line represents the position of the first 9 columns):
3  c23456789
4  !$omp parallel do shared(a,b,c)
5
6  c$omp parallel do
7  c$omp+shared(a,b,c)
8
9  c$omp paralleldoshared(a,b,c)

```

## 8.2.1.2 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:



```
end critical
end do
end master
end ordered
end parallel
end sections
end single
end task
end workshare
parallel do
parallel sections
parallel workshare
```

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

```
3 !23456789
4     !$omp parallel do &
5         !$omp shared(a,b,c)
6
6     !$omp parallel &
7     !$omp&do shared(a,b,c)
8
8     !$omp paralleldo shared(a,b,c)
```

Fortran

---

## 2.2 Conditional Compilation

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

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

For examples of conditional compilation, see Section A.3 on page 167.

### Fortran

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

### 2.2.1 Fixed Source Form Conditional Compilation Sentinels

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

<code>!\$</code>   <code>*\$</code>   <code>c\$</code>
--

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):

```

1      c23456789
2      !$ 10 iam = omp_get_thread_num() +
3
4      !$      &          index
5
6
7      #ifdef _OPENMP
8          10 iam = omp_get_thread_num() +
9              &          index
10     #endif

```

## 2.2.2 Free Source Form Conditional Compilation Sentinel

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

```
!$
```

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

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

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

```

1   Note – in the following example, the two forms for specifying conditional compilation
2   in free source form are equivalent (the first line represents the position of the first 9
3   columns):
4   c23456789
5       !$ iam = omp_get_thread_num() +      &
6       !$&    index
7
7   #ifdef _OPENMP
8       iam = omp_get_thread_num() +      &
9       index
10  #endif

```

Fortran

## 11 2.3 Internal Control Variables

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

20 For purposes of exposition, this document refers to the ICVs by certain names, but an  
21 implementation is not required to use these names or to offer any way to access the  
22 variables other than through the ways shown in Section 2.3.2 on page 29.

### 23 2.3.1 ICV Descriptions

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

- 25 • *dyn-var* - controls whether dynamic adjustment of the number of threads is enabled  
26 for encountered **parallel** regions. There is one copy of this ICV per task.

- 1 • *nest-var* - controls whether nested parallelism is enabled for encountered **parallel**
- 2 regions. There is one copy of this ICV per task.
- 3 • *nthreads-var* - controls the number of threads requested for encountered **parallel**
- 4 regions. There is one copy of this ICV per task.
- 5 • *thread-limit-var* - controls the maximum number of threads participating in the
- 6 OpenMP program. There is one copy of this ICV for the whole program.
- 7 • *max-active-levels-var* - controls the maximum number of nested active **parallel**
- 8 regions. There is one copy of this ICV for the whole program.

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

- 10 • *run-sched-var* - controls the schedule that the **runtime** schedule clause uses for
- 11 loop regions. There is one copy of this ICV per task.
- 12 • *def-sched-var* - controls the implementation defined default scheduling of loop
- 13 regions. There is one copy of this ICV for the whole program.

14 The following ICVs store values that affect the program execution.

- 15 • *bind-var* - controls the binding of threads to processors. If binding is enabled, the
- 16 execution environment is advised not to move OpenMP threads between processors.
- 17 There is one copy of this ICV for the whole program.
- 18 • *stacksize-var* - controls the stack size for threads that the OpenMP implementation
- 19 creates. There is one copy this ICV for the whole program.
- 20 • *wait-policy-var* - controls the desired behavior of waiting threads. There is one copy
- 21 of this ICV for the whole program.

## 22 2.3.2 Modifying and Retrieving ICV Values

23 The following table shows the methods for retrieving the values of the ICVs as well as  
 24 their initial values:

ICV	Scope	Ways to modify value	Way to retrieve value	Initial value
<i>dyn-var</i>	data environment	<b>OMP_DYNAMIC</b> <code>omp_set_dynamic()</code>	<code>omp_get_dynamic()</code>	See comments below
<i>nest-var</i>	data environment	<b>OMP_NESTED</b> <code>omp_set_nested()</code>	<code>omp_get_nested()</code>	<i>false</i>
<i>nthreads-var</i>	data environment	<b>OMP_NUM_THREADS</b> <code>omp_set_num_threads()</code>	<code>omp_get_max_threads()</code>	Implementation defined
<i>run-sched-var</i>	data environment	<b>OMP_SCHEDULE</b> <code>omp_set_schedule()</code>	<code>omp_get_schedule()</code>	Implementation defined
<i>def-sched-var</i>	global	(none)	(none)	Implementation defined

ICV	Scope	Ways to modify value	Way to retrieve value	Initial value
<i>bind-var</i>	global	OMP_PROC_BIND	(none)	Implementation defined
<i>stacksize-var</i>	global	OMP_STACKSIZE	(none)	Implementation defined
<i>wait-policy-var</i>	global	OMP_WAIT_POLICY	(none)	Implementation defined
<i>thread-limit-var</i>	global	OMP_THREAD_LIMIT	<code>omp_get_thread_limit()</code>	Implementation defined
<i>max-active-level-s-var</i>	global	OMP_MAX_ACTIVE_LEVELS <code>omp_set_max_active_levels()</code>	<code>omp_get_max_active_levels()</code>	See comments below

1           Comments:

- 2           • The value of the *nthreads-var* ICV is a list. The runtime call  
3            `omp_set_num_threads()` sets the value of the first element of this list, and  
4            `omp_get_max_threads()` retrieves the value of the first element of this list.  
5           • The initial value of *dyn-var* is implementation defined if the implementation supports  
6           dynamic adjustment of the number of threads; otherwise, the initial value is *false*.  
7           • The initial value of *max-active-levels-var* is the number of levels of parallelism that  
8           the implementation supports. See the definition of *supporting n levels of parallelism*  
9           in Section 1.2.5 on page 11 for further details.

10           After the initial values are assigned, but before any OpenMP construct or OpenMP API  
11           routine executes, the values of any OpenMP environment variables that were set by the  
12           user are read and the associated ICVs are modified accordingly. After this point, no  
13           changes to any OpenMP environment variables will affect the ICVs.

14           Clauses on OpenMP constructs do not modify the values of any of the ICVs.

## 15 2.3.3           How the Per-Data Environment ICVs Work

16           Each data environment has its own copies of internal variables *dyn-var*, *nest-var*,  
17           *nthreads-var*, and *run-sched-var*.

18           Calls to `omp_set_num_threads()`, `omp_set_dynamic()`,  
19           `omp_set_nested()`, and `omp_set_schedule()` modify only the ICVs in the  
20           data environment of their binding task.

21           When a **task** construct or **parallel** construct is encountered, the generated task(s)  
22           inherit the values of *dyn-var*, *nest-var*, and *run-sched-var* from the generating task's ICV  
23           values.

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

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

## 11 2.3.4 ICV Override Relationships

12 The override relationships among various construct clauses, OpenMP API routines,  
 13 environment variables, and the initial values of ICVs are shown in the following table:

construct clause, if used	overrides call to API routine	overrides setting of environment variable	overrides initial value of
(none)	<code>omp_set_dynamic()</code>	OMP_DYNAMIC	<i>dyn-var</i>
(none)	<code>omp_set_nested()</code>	OMP_NESTED	<i>nest-var</i>
<b>num_threads</b>	<code>omp_set_num_threads()</code>	OMP_NUM_THREADS	<i>nthreads-var</i> *
<b>schedule</b>	<code>omp_set_schedule()</code>	OMP_SCHEDULE	<i>run-sched-var</i>
(none)	(none)	OMP_PROC_BIND	<i>bind-var</i>
<b>schedule</b>	(none)	(none)	<i>def-sched-var</i>
(none)	(none)	OMP_STACKSIZE	<i>stacksize-var</i>
(none)	(none)	OMP_WAIT_POLICY	<i>wait-policy-var</i>
(none)	(none)	OMP_THREAD_LIMIT	<i>thread-limit-var</i>
(none)	<code>omp_set_max_active_levels()</code>	OMP_MAX_ACTIVE_LEVELS	<i>max-active-levels-var</i>

14 \* The **num\_threads** clause and `omp_set_num_threads()` override the value of  
 15 the OMP\_NUM\_THREADS environment variable and the initial value of the first element  
 16 of the *nthreads-var* ICV.

### 17 Cross References:

- 18 • **parallel** construct, see Section 2.4 on page 32.
- 19 • **num\_threads** clause, see Section 2.4.1 on page 36.

- 1 • `schedule` clause, see Section 2.5.1.1 on page 46.
- 2 • Loop construct, see Section 2.5.1 on page 38.
- 3 • `omp_set_num_threads` routine, see Section 3.2.1 on page 114.
- 4 • `omp_get_max_threads` routine, see Section 3.2.3 on page 116.
- 5 • `omp_set_dynamic` routine, see Section 3.2.7 on page 121.
- 6 • `omp_get_dynamic` routine, see Section 3.2.8 on page 122.
- 7 • `omp_set_nested` routine, see Section 3.2.9 on page 123.
- 8 • `omp_get_nested` routine, see Section 3.2.10 on page 124.
- 9 • `omp_set_schedule` routine, see Section 3.2.11 on page 125.
- 10 • `omp_get_schedule` routine, see Section 3.2.12 on page 127.
- 11 • `omp_get_thread_limit` routine, see Section 3.2.13 on page 129.
- 12 • `omp_set_max_active_levels` routine, see Section 3.2.14 on page 130.
- 13 • `omp_get_max_active_levels` routine, see Section 3.2.15 on page 131.
- 14 • `OMP_SCHEDULE` environment variable, see Section 4.1 on page 152.
- 15 • `OMP_NUM_THREADS` environment variable, see Section 4.2 on page 153.
- 16 • `OMP_DYNAMIC` environment variable, see Section 4.3 on page 154.
- 17 • `OMP_NESTED` environment variable, see Section 4.5 on page 155.
- 18 • `OMP_STACKSIZE` environment variable, see Section 4.6 on page 155.
- 19 • `OMP_WAIT_POLICY` environment variable, see Section 4.7 on page 156.
- 20 • `OMP_MAX_ACTIVE_LEVELS` environment variable, see Section 4.8 on page 157.
- 21 • `OMP_THREAD_LIMIT` environment variable, see Section 4.9 on page 157.

---

## 22 **2.4 parallel Construct**

### 23 **Summary**

24 This fundamental construct starts parallel execution. See Section 1.3 on page 12 for a  
25 general description of the OpenMP execution model.



1

## Syntax

C/C++

2

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

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

3

where *clause* is one of the following:

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

C/C++

Fortran

4

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

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

5

where *clause* is one of the following:

```
if (scalar-logical-expression)
num_threads (scalar-integer-expression)
default (private | firstprivate | shared | none)
private (list)
firstprivate (list)
```

```
shared(list)
copyin(v)
reduction({operator|intrinsic_procedure_name}:list)
```

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

Fortran

## 2 Binding

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

## 5 Description

6 When a thread encounters a **parallel** construct, a team of threads is created to  
7 execute the **parallel** region (see Section 2.4.1 on page 36 for more information about  
8 how the number of threads in the team is determined, including the evaluation of the **if**  
9 and **num\_threads** clauses). The thread that encountered the **parallel** construct  
10 becomes the master thread of the new team, with a thread number of zero for the  
11 duration of the new **parallel** region. All threads in the new team, including the  
12 master thread, execute the region. Once the team is created, the number of threads in the  
13 team remains constant for the duration of that **parallel** region.

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

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

25 The implementation may cause any thread to suspend execution of its implicit task at a  
26 task scheduling point, and switch to execute any explicit task generated by any of the  
27 threads in the team, before eventually resuming execution of the implicit task (for more  
28 details see Section 2.7 on page 59).

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

4 If a thread in a team executing a **parallel** region encounters another **parallel**  
5 directive, it creates a new team, according to the rules in Section 2.4.1 on page 36, and  
6 it becomes the master of that new team.



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





12 For an example of the **parallel** construct, see Section A.5 on page 170. For an  
13 example of the **num\_threads** clause, see Section A.6 on page 173.

## 14 Restrictions

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

- 16 • A program that branches into or out of a **parallel** region is non-conforming.
- 17 • A program must not depend on any ordering of the evaluations of the clauses of the  
18 **parallel** directive, or on any side effects of the evaluations of the clauses.
- 19 • At most one **if** clause can appear on the directive.
- 20 • At most one **num\_threads** clause can appear on the directive. The **num\_threads**  
21 expression must evaluate to a positive integer value.

22  C/C++   
23 • A **throw** executed inside a **parallel** region must cause execution to resume  
24 within the same **parallel** region, and the same thread that threw the exception  
must catch it.

25  C/C++   
26  Fortran   
• Unsynchronized use of Fortran I/O statements by multiple threads on the same unit  
has unspecified behavior.

27  Fortran 

## 27 Cross References

- 28 • **default**, **shared**, **private**, **firstprivate**, and **reduction** clauses, see  
29 Section 2.9.3 on page 90.
- 30 • **copyin** clause, see Section 2.9.4 on page 105.

1

- `omp_get_thread_num` routine, see Section 3.2.4 on page 117.

## 2 2.4.1

# 3 Determining the Number of Threads for a 4 parallel Region

3

4

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

8

9 Note that using a variable in an `if` or `num_threads` clause expression of a  
10 `parallel` construct causes an implicit reference to the variable in all enclosing  
11 constructs. The `if` clause expression and the `num_threads` clause expression are  
12 evaluated in the context outside of the `parallel` construct, and no ordering of those  
13 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.

14

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

15

---

### Algorithm 2.1

---

```
let ThreadsBusy be the number of OpenMP threads currently executing;  
let ActiveParRegions be the number of enclosing active parallel regions;  
if an if clause exists  
then let IfClauseValue be the value of the if clause expression;  
else let IfClauseValue = true;  
if a num_threads clause exists  
then let ThreadsRequested be the value of the num_threads clause  
expression;  
else let ThreadsRequested = value of the first element of nthreads-var;  
let ThreadsAvailable = (thread-limit-var - ThreadsBusy + 1);  
if (IfClauseValue = false)  
then number of threads = 1;  
else if (ActiveParRegions >= 1) and (nest-var = false)  
then number of threads = 1;  
else if (ActiveParRegions = max-active-levels-var)
```

---

---

## Algorithm 2.1

---

```
then number of threads = 1;  
else if (dyn-var = true) and (ThreadsRequested <= ThreadsAvailable)  
then number of threads = [ 1 : ThreadsRequested ];  
else if (dyn-var = true) and (ThreadsRequested > ThreadsAvailable)  
then number of threads = [ 1 : ThreadsAvailable ];  
else if (dyn-var = false) and (ThreadsRequested <= ThreadsAvailable)  
then number of threads = ThreadsRequested;  
else if (dyn-var = false) and (ThreadsRequested > ThreadsAvailable)  
then behavior is implementation defined;
```

---

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

### 4 **Cross References**

- 5 • *nthreads-var*, *dyn-var*, *thread-limit-var*, *max-active-level-var*, and *nest-var* ICVs, see  
6 Section 2.3 on page 28.

---

## 7 **2.5 Worksharing Constructs**

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

12 A worksharing region has no barrier on entry; however, an implied barrier exists at the  
13 end of the worksharing region, unless a **nowait** clause is specified. If a **nowait**  
14 clause is present, an implementation may omit the barrier at the end of the worksharing  
15 region. In this case, threads that finish early may proceed straight to the instructions  
16 following the worksharing region without waiting for the other members of the team to  
17 finish the worksharing region, and without performing a flush operation (see  
18 Section A.9 on page 178 for an example.)

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

- 3 • loop construct
- 4 • **sections** construct
- 5 • **single** construct
- 6 • **workshare** construct

## 7 **Restrictions**

8 The following restrictions apply to worksharing constructs:

- 9 • Each worksharing region must be encountered by all threads in a team or by none at  
10 all.
- 11 • The sequence of worksharing regions and **barrier** regions encountered must be the  
12 same for every thread in a team.


## 13 **2.5.1 Loop Construct**

### 14 **Summary**

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

### 19 **Syntax**

20 The syntax of the loop construct is as follows:

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

1

where *clause* is one of the following:

```

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

```

▼----- C/C++ (cont.) -----▼

2

The **for** directive places restrictions on the structure of all associated *for-loops*.

3

Specifically, all associated *for-loops* must have the following canonical form:

---

**for** (*init-expr*; *test-expr*; *incr-expr*) *structured-block*

---

<i>init-expr</i>	One of the following: <i>var</i> = <i>lb</i> <i>integer-type var</i> = <i>lb</i> <i>random-access-iterator-type var</i> = <i>lb</i> <i>pointer-type var</i> = <i>lb</i>
<i>test-expr</i>	One of the following: <i>var relational-op b</i> <i>b relational-op var</i>
<i>incr-expr</i>	One of the following: <b>++</b> <i>var</i> <i>var</i> <b>++</b> <b>--</b> <i>var</i> <i>var</i> <b>--</b> <b>v +=</b> <i>incr</i> <i>var -=</i> <i>incr</i> <i>var</i> = <i>var</i> + <i>incr</i> <i>var</i> = <i>incr</i> + <i>var</i> <i>var</i> = <i>var</i> - <i>incr</i>

---

---

<i>var</i>	<p>One of the following:</p> <ul style="list-style-type: none"> <li>A variable of a signed or unsigned integer type.</li> <li>For C++, a variable of a random access iterator type.</li> <li>For C, a variable of a pointer type.</li> </ul> <p>If this variable would otherwise be shared, it is implicitly made private in the loop construct. This variable must not be modified during the execution of the <i>for-loop</i> other than in <i>incr-expr</i>. Unless the variable is specified <b>lastprivate</b> on the loop construct, its value after the loop is unspecified.</p>
<i>relational-op</i>	<p>One of the following:</p> <ul style="list-style-type: none"> <li>&lt;</li> <li>&lt;=</li> <li>&gt;</li> <li>&gt;=</li> </ul>
<i>lb</i> and <i>b</i>	<p>Loop invariant expressions of a type compatible with the type of <i>var</i>.</p>
<i>incr</i>	<p>A loop invariant integer expression.</p>

---

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

- 4 • If *var* is of an integer type, then the type is the type of *var*.
- 5 • For C++, if *var* is of a random access iterator type, then the type is the type that  
6 would be used by *std::distance* applied to variables of the type of *var*.
- 7 • For C, if *var* is of a pointer type, then the type is *ptrdiff\_t*.

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

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

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





1 The syntax of the loop construct is as follows:

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

2 where *clause* is one of the following:

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

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

5 All associated *do-loops* must be *do-constructs* as defined by the Fortran standard. If an  
6 **end do** directive follows a *do-construct* in which several loop statements share a **DO**  
7 termination statement, then the directive can only be specified for the outermost of these  
8 **DO** statements. See Section A.7 on page 175 for examples.

9 If any of the loop iteration variables would otherwise be shared, they are implicitly  
10 made private on the loop construct. See Section A.8 on page 177 for examples. Unless  
11 the loop iteration variables are specified **lastprivate** on the loop construct, their  
12 values after the loop are unspecified.

## 13 Binding

14 The binding thread set for a loop region is the current team. A loop region binds to the  
15 innermost enclosing **parallel** region. Only the threads of the team executing the  
16 binding **parallel** region participate in the execution of the loop iterations and  
17 (optional) implicit barrier of the loop region.

## Description

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

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

The **collapse** clause may be used to specify how many loops are associated with the loop construct. The parameter of the **collapse** clause must be a constant positive integer expression. If no **collapse** clause is present, the only loop that is associated with the loop construct is the one that immediately follows the construct.

If more than one loop is associated with the loop construct, then the iterations of all associated loops are collapsed into one larger iteration space that is then divided according to the **schedule** clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

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

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

A worksharing loop has logical iterations numbered 0,1,...,N-1 where N is the number of loop iterations, and the logical numbering denotes the sequence in which the iterations would be executed if the associated loop(s) were executed by a single thread. The **schedule** clause specifies how iterations of the associated loops are divided into contiguous non-empty subsets, called chunks, and how these chunks are distributed among threads of the team. Each thread executes its assigned chunk(s) in the context of its implicit task. The *chunk\_size* expression is evaluated using the original list items of any variables that are made private in the loop construct. It is unspecified whether, in what order, or how many times, any side-effects of the evaluation of this expression occur. The use of a variable in a **schedule** clause expression of a loop construct causes an implicit reference to the variable in all enclosing constructs.

Different loop regions with the same schedule and iteration count, even if they occur in the same parallel region, can distribute iterations among threads differently. The only exception is for the **static** schedule as specified in Table 2-1. Programs that depend on which thread executes a particular iteration under any other circumstances are non-conforming.

See Section 2.5.1.1 on page 46 for details of how the schedule for a worksharing loop is determined.

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

TABLE 2-1 `schedule` clause *kind* values

---

<b>static</b>	<p>When <code>schedule(static, chunk_size)</code> is specified, iterations are divided into chunks of size <code>chunk_size</code>, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number.</p> <p>When no <code>chunk_size</code> is specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread. Note that the size of the chunks is unspecified in this case.</p> <p>A compliant implementation of <b>static</b> schedule must ensure that the same assignment of logical iteration numbers to threads will be used in two loop regions if the following conditions are satisfied: 1) both loop regions have the same number of loop iterations, 2) both loop regions have the same value of <code>chunk_size</code> specified, or both loop regions have no <code>chunk_size</code> specified, and 3) both loop regions bind to the same parallel region. A data dependence between the same logical iterations in two such loops is guaranteed to be satisfied allowing safe use of the <b>nowait</b> clause (see Section A.9 on page 178 for examples).</p>
<b>dynamic</b>	<p>When <code>schedule(dynamic, chunk_size)</code> is specified, the iterations are distributed to threads in the team in chunks as the threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed.</p> <p>Each chunk contains <code>chunk_size</code> iterations, except for the last chunk to be distributed, which may have fewer iterations.</p> <p>When no <code>chunk_size</code> is specified, it defaults to 1.</p>
<b>guided</b>	<p>When <code>schedule(guided, chunk_size)</code> is specified, the iterations are assigned to threads in the team in chunks as the executing threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned.</p> <p>For a <code>chunk_size</code> of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to 1. For a <code>chunk_size</code> with value <i>k</i> (greater than 1), the size of each chunk is determined in the same way, with the restriction that the chunks do not contain fewer than <i>k</i> iterations (except for the last chunk to be assigned, which may have fewer than <i>k</i> iterations).</p> <p>When no <code>chunk_size</code> is specified, it defaults to 1.</p>
<b>auto</b>	<p>When <code>schedule(auto)</code> is specified, the decision regarding scheduling is delegated to the compiler and/or runtime system. The programmer gives the implementation the freedom to choose any possible mapping of iterations to threads in the team.</p>

**runtime** When `schedule(runtime)` is specified, the decision regarding scheduling is deferred until run time, and the schedule and chunk size are taken from the *run-sched-var* ICV. If the ICV is set to **auto**, the schedule is implementation defined.

---

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

---

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

---

## Restrictions

Restrictions to the loop construct are as follows:

- All loops associated with the loop construct must be perfectly nested; that is, there must be no intervening code nor any OpenMP directive between any two loops.
- The values of the loop control expressions of the loops associated with the loop directive must be the same for all the threads in the team.
- Only one **schedule** clause can appear on a loop directive.
- Only one **collapse** clause can appear on a loop directive.
- *chunk\_size* must be a loop invariant integer expression with a positive value.
- The value of the *chunk\_size* expression must be the same for all threads in the team.
- The value of the *run-sched-var* ICV must be the same for all threads in the team.
- When **schedule(runtime)** or **schedule(auto)** is specified, *chunk\_size* must not be specified.
- Only a single **ordered** clause can appear on a loop directive.
- The **ordered** clause must be present on the loop construct if any **ordered** region ever binds to a loop region arising from the loop construct.

- 1 • The loop iteration variable may not appear in a **threadprivate** directive.

C/C++

- 2 • The associated *for-loops* must be structured blocks.
- 3 • Only an iteration of the innermost associated loop may be curtailed by a **continue**
- 4 statement.
- 5 • No statement can branch to any associated **for** statement.
- 6 • Only one **nowait** clause can appear on a **for** directive.
- 7 • If *test-expr* is of the form *var relational-op b* and *relational-op* is **<** or **<=** then
- 8 *incr-expr* must cause *var* to increase on each iteration of the loop. If *test-expr* is of
- 9 the form *var relational-op b* and *relational-op* is **>** or **>=** then *incr-expr* must cause
- 10 *var* to decrease on each iteration of the loop.
- 11 • If *test-expr* is of the form *b relational-op var* and *relational-op* is **<** or **<=** then
- 12 *incr-expr* must cause *var* to decrease on each iteration of the loop. If *test-expr* is of
- 13 the form *b relational-op var* and *relational-op* is **>** or **>=** then *incr-expr* must cause
- 14 *var* to increase on each iteration of the loop.
- 15 • A throw executed inside a loop region must cause execution to resume within the
- 16 same iteration of the loop region, and the same thread that threw the exception must
- 17 catch it.

C/C++

Fortran

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

Fortran

## Cross References

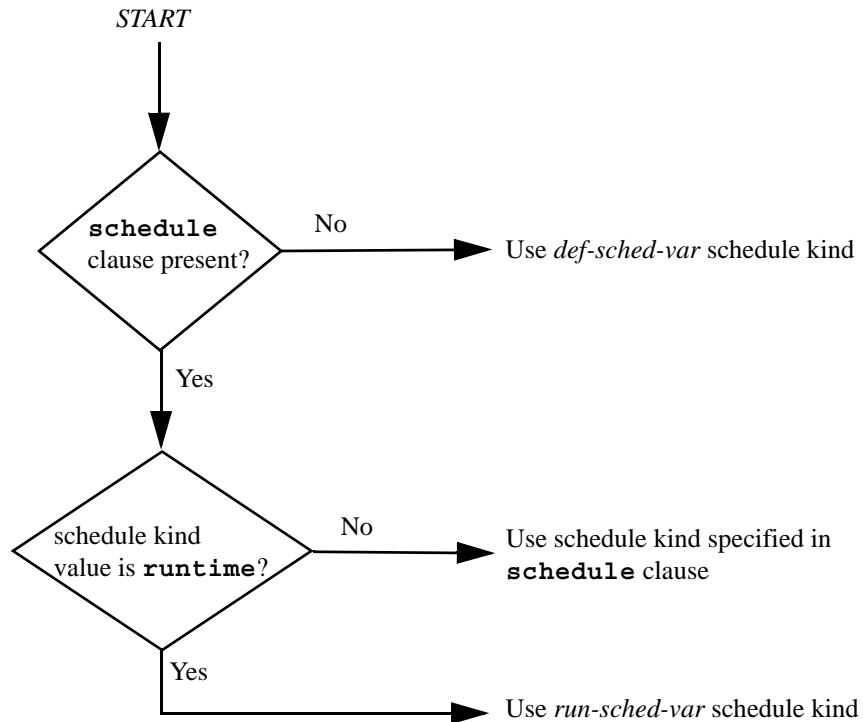
- 25 • **private**, **firstprivate**, **lastprivate**, and **reduction** clauses, see
- 26 Section 2.9.3 on page 90.
- 27
- 28 • **OMP\_SCHEDULE** environment variable, see Section 4.1 on page 152.
- 29 • **ordered** construct, see Section 2.8.7 on page 81.

## 1 2.5.1.1 Determining the Schedule of a Worksharing Loop

2 When execution encounters a loop directive, the **schedule** clause (if any) on the  
3 directive, and the *run-sched-var* and *def-sched-var* ICVs are used to determine how loop  
4 iterations are assigned to threads. See Section 2.3 on page 28 for details of how the  
5 values of the ICVs are determined. If the loop directive does not have a **schedule**  
6 clause then the current value of the *def-sched-var* ICV determines the schedule. If the  
7 loop directive has a **schedule** clause that specifies the **runtime** schedule kind then  
8 the current value of the *run-sched-var* ICV determines the schedule. Otherwise, the  
9 value of the **schedule** clause determines the schedule. Figure 2-1 describes how the  
10 schedule for a worksharing loop is determined.

### 11 Cross References

- 12 • ICVs, see Section 2.3 on page 28.



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

## 1 2.5.2 sections Construct

### 2 Summary

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

### 7 Syntax

C/C++

8 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 ]
  ...
}
```

9 where *clause* is one of the following:

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

C/C++

1 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]
    
```

2 where *clause* is one of the following:

```

private (list)
firstprivate (list)
lastprivate (list)
reduction ({operator|intrinsic_procedure_name} :list)
    
```

### 3 **Binding**

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

### 8 **Description**

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

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

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



1

## Restrictions

2

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

3

- Orphaned **section** directives are prohibited; i.e., the **section** directives must appear within the **sections** construct and may not be encountered elsewhere in the **sections** region.

4

5

6

- The code enclosed in a **sections** construct must be a structured block.

7

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

C/C++

8

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

9

10

C/C++

11

## Cross References

12

- **private**, **firstprivate**, **lastprivate**, and **reduction** clauses, see Section 2.9.3 on page 90.

13

## 2.5.3

## single Construct

15

### Summary

16

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

17

18

19

20

### Syntax

C/C++

21

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

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

1 where *clause* is one of the following:

```
private (list)
firstprivate (list)
copyprivate (list)
nowait
```

▲──────────────────────────────── C/C++ ─────────────────────────────────▲

▼──────────────────────────────── Fortran ─────────────────────────────────▼

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

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

3 where *clause* is one of the following:

```
private (list)
firstprivate (list)
```

4 and *end\_clause* is one of the following:

```
copyprivate (list)
nowait
```

▲──────────────────────────────── Fortran ─────────────────────────────────▲

## 5 **Binding**

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

1

## Description

2

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

3

4

For an example of the **single** construct, see Section A.12 on page 186.

5

## Restrictions

6

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

7

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

8

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

9

▼ C/C++ ▼

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

▲ C/C++ ▲

10

11

12

## Cross References

13

- **private** and **firstprivate** clauses, see Section 2.9.3 on page 90.

14

- **copyprivate** clause, see Section 2.9.4.2 on page 107.

▼ Fortran ▼

## 2.5.4

### workshare Construct

15

## Summary

16

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

17

18

19

## Syntax

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

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

The enclosed structured block must consist of only the following:

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

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

## Binding

The binding thread set for a **workshare** region is the current team. A **workshare** region binds to the innermost enclosing **parallel** region. Only the threads of the team executing the binding **parallel** region participate in the execution of the units of work and the (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 complete prior to the effects of assigning to the left hand side.

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

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

23 The transformational array intrinsic functions are **MATMUL**, **DOT\_PRODUCT**, **SUM**,  
 24 **PRODUCT**, **MAXVAL**, **MINVAL**, **COUNT**, **ANY**, **ALL**, **SPREAD**, **PACK**, **UNPACK**,  
 25 **RESHAPE**, **TRANSPOSE**, **EOSHIFT**, **CSHIFT**, **MINLOC**, and **MAXLOC**.

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

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

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

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

35 For examples of the **workshare** construct, see Section A.14 on page 201.

## Restrictions

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

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

---

## 2.6 Combined Parallel Worksharing Constructs

Combined parallel worksharing constructs are shortcuts for specifying a worksharing construct nested immediately inside a **parallel** construct. The semantics of these directives are identical to that of explicitly specifying a **parallel** construct containing one worksharing construct and no other statements.

The combined parallel worksharing constructs allow certain clauses that are permitted both on **parallel** constructs and on worksharing constructs. If a program would have different behavior depending on whether the clause were applied to the **parallel** construct or to the worksharing construct, then the program's behavior is unspecified.

The following sections describe the combined parallel worksharing constructs:

- The **parallel** loop construct.
- The **parallel sections** construct.
- The **parallel workshare** construct.

### 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.

1

## Syntax

C/C++

2

The syntax of the parallel loop construct is as follows:

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

3

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

4

C/C++

Fortran

5

The syntax of the parallel loop construct is as follows:

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

6

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

7

8

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

9

10

Fortran

11

## Description

C/C++

12

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

13

C/C++

Fortran

14

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.

15

16

Fortran

## 1           **Restrictions**

2           The restrictions for the **parallel** construct and the loop construct apply.

## 3           **Cross References**


- 4           • **parallel** construct, see Section 2.4 on page 32.
- 5           • loop construct, see Section 2.5.1 on page 38.
- 6           • Data attribute clauses, see Section 2.9.3 on page 90.

## 7 **2.6.2           parallel sections Construct**

### 8           **Summary**

9           The **parallel sections** construct is a shortcut for specifying a **parallel**  
10          construct containing one **sections** construct and no other statements.

### 11          **Syntax**

12           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 ]
...
}
```

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

 C/C++



## Fortran

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

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

2 where *clause* can be any of the clauses accepted by the **parallel** or **sections**  
3 directives, with identical meanings and restrictions.

4 The last section ends at the **end parallel sections** directive. **nowait** cannot be  
5 specified on an **end parallel sections** directive.

## Fortran

### 6 Description

#### C/C++

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

#### C/C++

#### Fortran

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

#### Fortran

12 For an example of the parallel sections construct, see Section A.11 on page 185.

### 13 Restrictions

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

### 15 Cross References:

- 16 • **parallel** construct, see Section 2.4 on page 32.

- 1 • **sections** construct, see Section 2.5.2 on page 47.
- 2 • Data attribute clauses, see Section 2.9.3 on page 90.

### 3 2.6.3 parallel workshare Construct

#### 4 Summary

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

#### 7 Syntax

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

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

9 where *clause* can be any of the clauses accepted by the **parallel** directive, with  
10 identical meanings and restrictions. **nowait** may not be specified on an **end**  
11 **parallel workshare** directive.

#### 12 Description

13 The semantics are identical to explicitly specifying a **parallel** directive immediately  
14 followed by a **workshare** directive, and an **end workshare** directive immediately  
15 followed by an **end parallel** directive.

#### 16 Restrictions

17 The restrictions for the **parallel** construct and the **workshare** construct apply.

#### 18 Cross References

- 19 • **parallel** construct, see Section 2.4 on page 32.
- 20 • **workshare** construct, see Section 2.5.4 on page 51.

- 1 • Data attribute clauses, see Section 2.9.3 on page 90.

Fortran

---

## 2 2.7 Tasking Constructs

### 3 2.7.1 task Construct

#### 4 Summary

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

#### 6 Syntax

C/C++

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

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

8 where *clause* is one of the following:

```
if (scalar-expression)
final (scalar-expression)
untied
default (shared | none)
mergeable
private (list)
firstprivate (list)
shared (list)
```

## C/C++

## Fortran

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

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

2 where *clause* is one of the following:

```
    if (scalar-logical-expression)  
    final (scalar-logical-expression)  
    untied  
    default (private | firstprivate | shared | none)  
    mergeable  
    private (list)  
    firstprivate (list)  
    shared (list)
```

## Fortran

### 3 **Binding**

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

### 6 **Description**

7 When a thread encounters a **task** construct, a task is generated from the code for the  
8 associated structured block. The data environment of the task is created according to the  
9 data-sharing attribute clauses on the **task** construct, per-data environment ICVs, and  
10 any defaults that apply.

1 The encountering thread may immediately execute the task, or defer its execution. In the  
2 latter case, any thread in the team may be assigned the task. Completion of the task can  
3 be guaranteed using task synchronization constructs. A **task** construct may be nested  
4 inside an outer task, but the **task** region of the inner task is not a part of the **task**  
5 region of the outer task.

6 When an **if** clause is present on a **task** construct, and the **if** clause expression  
7 evaluates to **false**, an undeferred task is generated, and the encountering thread must  
8 suspend the current task region, for which execution cannot be resumed until the  
9 generated task is completed. Note that the use of a variable in an **if** clause expression  
10 of a **task** construct causes an implicit reference to the variable in all enclosing  
11 constructs.

12 When a **final** clause is present on a **task** construct and the **final** clause expression  
13 evaluates to **true**, the generated task will be a final task. All **task** constructs  
14 encountered during execution of a **final** task will generate included tasks. Note that  
15 the use of a variable in a **final** clause expression of a **task** construct causes an  
16 implicit reference to the variable in all enclosing constructs.

17 A thread that encounters a task scheduling point within the **task** region may  
18 temporarily suspend the **task** region. By default, a task is tied and its suspended task  
19 region can only be resumed by the thread that started its execution. If the **untied**  
20 clause is present on a **task** construct, any thread in the team can resume the **task**  
21 region after a suspension. The **untied** clause is ignored if a **final** clause is present  
22 on the same **task** construct and the **final** clause expression evaluates to **true**, or if  
23 a task is an included task.

24 The **task** construct includes a task scheduling point in the task region of its generating  
25 task, immediately following the generation of the explicit task. Each explicit **task**  
26 region includes a task scheduling point at its point of completion. An implementation  
27 may add task scheduling points anywhere in untied **task** regions.

28 When an **mergeable** clause is present on a **task** construct, and the generated task is  
29 an undeferred task or an included task, the implementation may generate a merged task  
30 instead.

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

## 34 **Restrictions**

35 Restrictions to the **task** construct are as follows:

- 1 • A program that branches into or out of a **task** region is non-conforming.
- 2 • A program must not depend on any ordering of the evaluations of the clauses of the
- 3 **task** directive, or on any side effects of the evaluations of the clauses.
- 4 • At most one **if** clause can appear on the directive.
- 5 • At most one **final** clause can appear on the directive.

C/C++

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

C/C++

Fortran

- 8 • Unsynchronized use of Fortran I/O statements by multiple tasks on the same unit has
- 9 unspecified behavior.

Fortran

## 10 2.7.2 **taskyield Construct**

### 11 **Summary**

12 The **taskyield** construct specifies that the current task can be suspended in favor of  
13 execution of a different task.

### 14 **Syntax**

C/C++

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

```
#pragma omp taskyield new-line
```

16 Because the **taskyield** construct is a stand-alone directive, there are some  
17 restrictions on its placement within a program. The **taskyield** directive may be  
18 placed only at a point where a base language statement is allowed. The **taskyield**  
19 directive may not be used in place of the statement following an **if**, **while**, **do**,  
20 **switch**, or **label**. See Appendix C “OpenMP C and C++ Grammar” for the formal  
21 grammar. The examples in Appendix A.22 illustrate these restrictions.

C/C++

1 The syntax of the `taskyield` construct is as follows:

```
!$omp taskyield
```

2 Because the `taskyield` construct is a stand-alone directive, there are some  
 3 restrictions on its placement within a program. The `taskyield` directive may be  
 4 placed only at a point where a Fortran executable statement is allowed. The  
 5 `taskyield` directive may not be used as the action statement in an `if` statement or as  
 6 the executable statement following a label if the label is referenced in the program. The  
 7 examples in Section A.22 on page 222 illustrate these restrictions.

## 8 Binding

9 A `taskyield` region binds to the current task region. The binding thread set of the  
 10 `taskyield` region is the current team.

## 11 Description

12 The `taskyield` region includes an explicit task scheduling point in the current task  
 13 region.

## 14 Restrictions

## 15 Cross References

- 16 • Task scheduling, see Section 2.7.3 on page 63.

## 17 2.7.3 Task Scheduling

18 Whenever a thread reaches a task scheduling point, the implementation may cause it to  
 19 perform a task switch, beginning or resuming execution of a different task bound to the  
 20 current team. Task scheduling points are implied at the following locations:

- 21 • the point immediately following the generation of an explicit task

- 1 • after the last instruction of a **task** region
- 2 • in **taskyield** regions
- 3 • in **taskwait** regions
- 4 • in implicit and explicit *barrier* regions.

5 In addition, implementations may insert task scheduling points in untied tasks anywhere  
6 that they are not specifically prohibited in this specification.

7 When a thread encounters a task scheduling point it may do one of the following,  
8 subject to the *Task Scheduling Constraints* (below):

- 9 • begin execution of a tied task bound to the current team.
- 10 • resume any suspended task region, bound to the current team, to which it is tied.
- 11 • begin execution of an untied task bound to the current team.
- 12 • resume any suspended untied task region bound to the current team.

13 If more than one of the above choices is available, it is unspecified as to which will be  
14 chosen.

#### 15 *Task Scheduling Constraints*

- 16 1. An included task is executed immediately after generation of the task.
- 17 2. Scheduling of new tied tasks is constrained by the set of task regions that are currently  
18 tied to the thread, and that are not suspended in a **barrier** region. If this set is empty,  
19 any new tied task may be scheduled. Otherwise, a new tied task may be scheduled only  
20 if it is a descendant of every task in the set.
- 21 3. When an explicit task is generated by a construct containing an **if** clause for which the  
22 expression evaluated to *false*, and the previous constraint is already met, the task is  
23 executed immediately after generation of the task.

24 A program relying on any other assumption about task scheduling is non-conforming.

---

25 **Note** – Task scheduling points dynamically divide task regions into parts. Each part is  
26 executed uninterruptedly from start to end. Different parts of the same task region are  
27 executed in the order in which they are encountered. In the absence of task  
28 synchronization constructs, the order in which a thread executes parts of different  
29 schedulable tasks is unspecified.

30 A correct program must behave correctly and consistently with all conceivable  
31 scheduling sequences that are compatible with the rules above.



1 For example, if **threadprivate** storage is accessed (explicitly in the source code or  
2 implicitly in calls to library routines) in one part of a task region, its value cannot be  
3 assumed to be preserved into the next part of the same task region if another schedulable  
4 task exists that modifies it (see Example A.13.7c on page 196, Example A.13.7f on page  
5 196, Example A.13.8c on page 197 and Example A.13.8f on page 197).

6 As another example, if a lock acquire and release happen in different parts of a task  
7 region, no attempt should be made to acquire the same lock in any part of another task  
8 that the executing thread may schedule; otherwise, a deadlock is possible. A similar  
9 situation can occur when a critical region spans multiple parts of a task and another  
10 schedulable task contains a critical region with the same name (see Example A.13.9c on  
11 page 198, Example A.13.9f on page 199, Example A.13.10c on page 200 and Example  
12 A.13.10f on page 201).

13 The use of **threadprivate** variables and the use of locks or critical sections in an explicit  
14 task with an **if** clause must take into account that when the **if** clause evaluates to  
15 false, the task is executed immediately, without regard to *Task Scheduling Constraint 2*.



## 16 **2.8 Master and Synchronization Constructs**

17 The following sections describe :

- 18 • the **master** construct.
- 19 • the **critical** construct.
- 20 • the **barrier** construct.
- 21 • the **taskwait** construct.
- 22 • the **atomic** construct.
- 23 • the **flush** construct.
- 24 • the **ordered** construct.

### 25 **2.8.1 master Construct**

#### 26 **Summary**

27 The **master** construct specifies a structured block that is executed by the master thread  
28 of the team.

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15

## 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.15 on page 205.

## Restrictions

C/C++

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

C/C++

## 1 2.8.2 critical Construct

### 2 Summary

3 The **critical** construct restricts execution of the associated structured block to a  
4 single thread at a time.

### 5 Syntax

#### C/C++

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

```
#pragma omp critical [(name)] new-line  
    structured-block
```

#### C/C++

#### Fortran

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

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

#### Fortran



### 8 Binding

9 The binding thread set for a **critical** region is all threads. Region execution is  
10 restricted to a single thread at a time among all the threads in the program, without  
11 regard to the team(s) to which the threads belong.



### 12 Description

13 An optional *name* may be used to identify the **critical** construct. All **critical**  
14 constructs without a name are considered to have the same unspecified name. A thread  
15 waits at the beginning of a **critical** region until no thread is executing a **critical**



1 region with the same name. The **critical** construct enforces exclusive access with  
2 respect to all **critical** constructs with the same name in all threads, not just those  
3 threads in the current team.

4  C/C++   
5 Identifiers used to identify a **critical** construct have external linkage and are in a  
6 name space that is separate from the name spaces used by labels, tags, members, and  
ordinary identifiers.

7  Fortran   
8 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.

9  Fortran   
10 For an example of the **critical** construct, see Section A.16 on page 207.

## 10 Restrictions

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

14  C/C++   
15  Fortran 

16 The following restrictions apply to the **critical** construct:

- 17 • If a *name* is specified on a **critical** directive, the same *name* must also be  
18 specified on the **end critical** directive.
- If no *name* appears on the **critical** directive, no *name* can appear on the **end  
critical** directive.

19  Fortran 

## 19 2.8.3 barrier Construct

### 20 Summary

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

1

## Syntax

C/C++

2

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

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

3

4

5

6

7

8

Because the **barrier** construct is a stand-alone directive, there are some restrictions on its placement within a program. The **barrier** directive may be placed only at a point where a base language statement is allowed. The **barrier** directive may not be used in place of the statement following an *if*, *while*, *do*, *switch*, or *label*. See Appendix C for the formal grammar. The examples in Section A.22 on page 222 illustrate these restrictions.

C/C++

Fortran

9

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

```
!$omp barrier
```

10

11

12

13

14

15

Because the **barrier** construct is a stand-alone directive, there are some restrictions on its placement within a program. The **barrier** directive may be placed only at a point where a Fortran executable statement is allowed. The **barrier** directive may not be used as the action statement in an **if** statement or as the executable statement following a label if the label is referenced in the program. The examples in Section A.22 on page 222 illustrate these restrictions.

Fortran

16

## Binding

17

18

19

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.18 on page 210 for examples.

20

## Description

21

22

23

24

All threads of the team executing the binding **parallel** region must execute the **barrier** region and complete execution of all explicit tasks generated in the binding **parallel** region up to this point before any are allowed to continue execution beyond the barrier.

1 The **barrier** region includes an implicit task scheduling point in the current task  
2 region.

### 3 **Restrictions**

4 The following restrictions apply to the **barrier** construct:


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

## 8 **2.8.4 taskwait Construct**

### 9 **Summary**

10 The **taskwait** construct specifies a wait on the completion of child tasks generated  
11 since the beginning of the current task.

### 12 **Syntax**

13  C/C++  
The syntax of the **taskwait** construct is as follows:

```
#pragma omp taskwait newline
```

14 Because the **taskwait** construct is a stand-alone directive, there are some restrictions  
15 on its placement within a program. The **taskwait** directive may be placed only at a  
16 point where a base language statement is allowed. The **taskwait** directive may not be  
17 used in place of the statement following an *if*, *while*, *do*, *switch*, or *label*. See Appendix  
18 C for the formal grammar. The examples in Section A.22 on page 222 illustrate these  
19 restrictions.

14  C/C++

15  Fortran

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

```
!$omp taskwait
```

1 Because the `taskwait` construct is a stand-alone directive, there are some restrictions  
2 on its placement within a program. The `taskwait` directive may be placed only at a  
3 point where a Fortran executable statement is allowed. The `taskwait` directive may  
4 not be used as the action statement in an `if` statement or as the executable statement  
5 following a label if the label is referenced in the program. The examples in Section A.22  
6 on page 222 illustrate these restrictions.

---

Fortran

## 7 Binding

8 A `taskwait` region binds to the current task region. The binding thread set of the  
9 `taskwait` region is the current team.

## 10 Description

11 The `taskwait` region includes an implicit task scheduling point in the current task  
12 region. The current task region is suspended at the task scheduling point until execution  
13 of all its child tasks generated before the `taskwait` region are completed.

## 14 2.8.5 atomic Construct

### 15 Summary

16 The `atomic` construct ensures that a specific storage location is accessed atomically,  
17 rather than exposing it to the possibility of multiple, simultaneous reading and writing  
18 threads that may result in indeterminate values.

### 19 Syntax

---

C/C++

20 The syntax of the `atomic` construct takes either of the following forms:

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

1

or:

```
#pragma omp atomic capture new-line
      structured-block
```

2

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

3

- If clause is **read**:

4

```
v = x;
```

5

- If clause is **write**:

6

```
x = expr;
```

7

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

8

```
x++;
```

9

```
x--;
```

10

```
++x;
```

11

```
--x;
```

12

```
x binop= expr;
```

13

- If clause is **capture**:

14

```
v = x++;
```

15

```
v = x--;
```

16

```
v = ++x;
```

17

```
v = --x;
```

18

```
v = x binop= expr;
```

19

and where *structured-block* is a structured block with one of the following forms:

20

```
{v = x; x binop= expr}
```

21

```
{x binop= expr; v = x}
```

22

In the preceding expressions:

23

- *x* and *v* (as applicable) are both *l-value* expressions with scalar type.

24

- During the execution of an atomic region, multiple syntactic occurrences of *x* must designate the same storage location.

25

26

- Neither of *v* and *expr* (as applicable) may access the storage location designated by *x*.

27

- *expr* is an expression with scalar type.

28

- *binop* is one of +, \*, -, /, &, ^, |, <<, or >>.

29

- *binop*, *binop* =, ++, and -- are not overloaded operators.



C/C++



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

```

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

2 or

```

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

3 or

```

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

4 or

```

!$omp atomic capture
    update-statement
    capture-statement
!$omp end atomic
```

5 or

```

!$omp atomic capture
    capture-statement
    update-statement
!$omp end atomic
```

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

7  $x = \textit{expr}$

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

9  $v = x$

10 and where *update-statement* has one of the following forms (if clause is **update**,  
11 **capture**, or not present):

1  $x = x \text{ operator } expr$   
2  $x = expr \text{ operator } x$   
3  $x = \text{intrinsic\_procedure\_name} (x, \text{expr\_list})$   
4  $x = \text{intrinsic\_procedure\_name} (\text{expr\_list}, x)$

5 In the preceding statements:

- 6 •  $x$  and  $v$  (as applicable) are both scalar variables of intrinsic type.
- 7 • During the execution of an atomic region, multiple syntactic occurrences of  $x$  must  
8 designate the same storage location.
- 9 • None of  $v$ ,  $expr$  and  $expr\_list$  (as applicable) may access the same storage location as  
10  $x$ .
- 11 •  $expr$  is a scalar expression.
- 12 •  $expr\_list$  is a comma-separated, non-empty list of scalar expressions. If  
13  $\text{intrinsic\_procedure\_name}$  refers to **IAND**, **IOR**, or **IEOR**, exactly one expression  
14 must appear in  $expr\_list$ .
- 15 •  $\text{intrinsic\_procedure\_name}$  is one of **MAX**, **MIN**, **IAND**, **IOR**, or **IEOR**.
- 16 •  $operator$  is one of **+**, **\***, **-**, **/**, **.AND.**, **.OR.**, **.EQV.**, or **.NEQV.**
- 17 • The operators in  $expr$  must have precedence equal to or greater than the precedence  
18 of  $operator$ ,  $x \text{ operator } expr$  must be mathematically equivalent to  $x \text{ operator } (expr)$ ,  
19 and  $expr \text{ operator } x$  must be mathematically equivalent to  $(expr) \text{ operator } x$ .
- 20 •  $\text{intrinsic\_procedure\_name}$  must refer to the intrinsic procedure name and not to other  
21 program entities.
- 22 •  $operator$  must refer to the intrinsic operator and not to a user-defined operator.
- 23 • All assignments must be intrinsic assignments.



Fortran

## 24 **Binding**

25 The binding thread set for an atomic region is all threads. **atomic** regions enforce  
26 exclusive access with respect to other **atomic** regions that access the same storage  
27 location  $x$  among all the threads in the program without regard to the teams to which the  
28 threads belong.

## 29 **Description**

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

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

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

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

20          The remaining description applies to all forms of the **atomic** construct: Any  
21          combination of two or more of these **atomic** constructs enforces mutually exclusive  
22          access to the locations designated by  $x$ . To avoid race conditions, all accesses of the  
23          locations designated by  $x$  that could potentially occur in parallel must be protected with  
24          an **atomic** construct.

25          **atomic** regions do not guarantee exclusive access with respect to any accesses outside  
26          of **atomic** regions to the same storage location  $x$  even if those accesses occur during a  
27          **critical** or **ordered** region, while an OpenMP lock is owned by the executing  
28          task, or during the execution of a **reduction** clause.

29          However, other OpenMP synchronization can ensure the desired exclusive access. For  
30          example, a barrier following a series of atomic updates to  $x$  guarantees that subsequent  
31          accesses do not form a race with the atomic accesses.

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

35          For an example of the **atomic** construct, see Section A.19 on page 212.

## Restrictions

### C/C++

The following restriction applies to the `atomic` construct:

- All atomic accesses to the storage locations designated by  $x$  throughout the program are required to have a compatible type. See Section A.20 on page 215 for examples.

### C/C++

### Fortran

The following restriction applies to the `atomic` construct:

- All atomic accesses to the storage locations designated by  $x$  throughout the program are required to have the same type and type parameters. See Section A.20 on page 215 for examples.

### Fortran

## Cross References

- `critical` construct, see Section 2.8.2 on page 67.
- `barrier` construct, see Section 2.8.3 on page 68.
- `flush` construct, see Section 2.8.6 on page 76.
- `ordered` construct, see Section 2.8.7 on page 81.
- `reduction` clause, see Section 2.9.3.6 on page 101.
- lock routines, see Section 3.3 on page 139.

## 2.8.6 `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 13 for more details.

1

## Syntax

C/C++

2

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

```
#pragma omp flush [(list)] new-line
```

3

4

5

6

7

8

Because the **flush** construct is a stand-alone directive, there are some restrictions on its placement within a program. The **flush** directive may be placed only at a point where a base language statement is allowed. The **flush** directive may not be used in place of the statement following an *if*, *while*, *do*, *switch*, or *label*. See Appendix C for the formal grammar. See Section A.22 on page 222 for an example that illustrates these placement restrictions.

C/C++

Fortran

9

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

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

10

11

12

13

14

15

Because the **flush** construct is a stand-alone directive, there are some restrictions on its placement within a program. The **flush** directive may be placed only at a point where a Fortran executable statement is allowed. The **flush** directive may not be used as the action statement in an **if** statement or as the executable statement following a label if the label is referenced in the program. The examples in Section Section A.22 on page 222 illustrate these restrictions.

Fortran

16

## Binding

17

18

19

20

21

The binding thread set for a **flush** region is the encountering thread. Execution of a **flush** region affects the memory and the temporary view of memory of only the thread that executes the region. It does not affect the temporary view of other threads. Other threads must themselves execute a flush operation in order to be guaranteed to observe the effects of the encountering thread's flush operation.

## Description

A **flush** construct without a list, executed on a given thread, operates as if the whole thread-visible data state of the program, as defined by the base language, is flushed. A **flush** construct with a list applies the flush operation to the items in the list, and does not return until the operation is complete for all specified list items. Use of a **flush** construct with a list is extremely error prone and users are strongly discouraged from attempting it. An implementation may implement a **flush** with a list by ignoring the list, and treating it the same as a **flush** without a list.

### C/C++

If a pointer is present in the list, the pointer itself is flushed, not the memory block to which the pointer refers.

### C/C++

### Fortran

If the list item or a subobject of the list item has the **POINTER** attribute, the allocation or association status of the **POINTER** item is flushed, but the pointer target is not. If the list item is a Cray pointer, the pointer is flushed, but the object to which it points is not. If the list item 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.22 on page 222.

1  
2  
3  
4  
5  
6  
7  
8  
9

**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 protected section by the two threads, but the program does not work properly because it does not enforce the proper ordering of the operations on variables **a** and **b**. Any shared data accessed in the protected section is not guaranteed to be current or consistent during or after the protected section. The atomic notation in the pseudo-code in the following two examples indicates that the accesses to **a** and **b** are **ATOMIC** writes and captures. Otherwise both examples would contain data races and automatically result in unspecified behavior.

```
Incorrect example:

                                a = b = 0

thread 1                          thread 2

atomic(b = 1)                     atomic(a = 1)
flush(b)                           flush(a)
flush(a)                           flush(b)
atomic(tmp = a)                   atomic(tmp = b)
if (tmp == 0) then                 if (tmp == 0) then
    protected section              protected section
end if                             end if
```

10  
11  
12  
13  
14  
15

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

1 The following correct pseudocode example correctly ensures that the protected section is  
2 executed by not more than one of the two threads at any one time. Notice that execution of  
3 the protected section by neither thread is considered correct in this example. This occurs if  
4 both flushes complete prior to either thread executing its `if` statement.

```
Correct example:

                                a = b = 0

thread 1                                thread 2

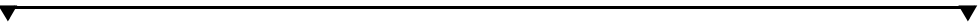
atomic(b = 1)                          atomic(a = 1)
flush(a,b)                               flush(a,b)
atomic(tmp = a)                          atomic(tmp = b)
if (tmp == 0) then                        if (tmp == 0) then
    protected section                      protected section
end if                                    end if
```

5 The compiler is prohibited from moving the flush at all for either thread, ensuring that the  
6 respective assignment is complete and the data is flushed before the `if` statement is  
7 executed.



- 8 A **flush** region without a list is implied at the following locations:
- 9 • During a barrier region.
  - 10 • At entry to and exit from **parallel**, **critical**, and **ordered** regions.
  - 11 • At exit from worksharing regions unless a **nowait** is present.
  - 12 • At entry to and exit from combined parallel worksharing regions.
  - 13 • During **omp\_set\_lock** and **omp\_unset\_lock** regions.
  - 14 • During **omp\_test\_lock**, **omp\_set\_nest\_lock**, **omp\_unset\_nest\_lock**  
15 and **omp\_test\_nest\_lock** regions, if the region causes the lock to be set or  
16 unset.
  - 17 • Immediately before and immediately after every task scheduling point.

- 18 A **flush** region with a list is implied at the following locations:
- 19 • At entry to and exit from the **atomic** operation (read, write, update, or capture)  
20 performed in an atomic region, where the list contains only the storage location  
21 designated as *x* according to the description of the syntax of the atomic construct in  
22 Section 2.8.5 on page 71.



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




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

## 3 2.8.7 ordered Construct



### 4 Summary

5 The **ordered** construct specifies a structured block in a loop region that will be  
6 executed in the order of the loop iterations. This sequentializes and orders the code  
7 within an **ordered** region while allowing code outside the region to run in parallel.

### 8 Syntax

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

```
#pragma omp ordered new-line  
                  structured-block
```

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

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



### 11 Binding

12 The binding thread set for an **ordered** region is the current team. An **ordered** region  
13 binds to the innermost enclosing loop region. **ordered** regions that bind to different  
14 loop regions execute 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 execution of all the **ordered** regions belonging to all previous iterations have completed.

For examples of the **ordered** construct, see Section A.23 on page 225.

## 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 or a loop nest within a loop region, a thread must not execute more than one **ordered** region that binds to the same loop region.

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

## Cross References

- loop construct, see Section 2.5.1 on page 38.
- parallel loop construct, see Section 2.6.1 on page 54.

---

## 2.9 Data Environment

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

- Section 2.9.1 on page 83 describes how the data-sharing attributes of variables referenced in **parallel**, **task**, and worksharing regions are determined.
- The **threadprivate** directive, which is provided to create threadprivate memory, is described in Section 2.9.2 on page 86.

- 1 • Clauses that may be specified on directives to control the data-sharing attributes of  
2 variables referenced in **parallel**, **task**, or worksharing constructs are described  
3 in Section 2.9.3 on page 90.
- 4 • Clauses that may be specified on directives to copy data values from private or  
5 threadprivate variables on one thread to the corresponding variables on other threads  
6 in the team are described in Section 2.9.4 on page 105.

## 7 2.9.1 Data-sharing Attribute Rules

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





- 11 • Section 2.9.1.1 on page 83 describes the data-sharing attribute rules for variables  
12 referenced in a construct.
- 13 • Section 2.9.1.2 on page 85 describes the data-sharing attribute rules for variables  
14 referenced in a region, but outside any construct.

### 15 2.9.1.1 Data-sharing Attribute Rules for Variables Referenced in a 16 Construct

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

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

22 The following variables have predetermined data-sharing attributes:

- 23  C/C++ 
- 24 • Variables appearing in **threadprivate** directives are threadprivate.
  - 25 • Variables with automatic storage duration that are declared in a scope inside the  
26 construct are private.
  - 27 • Objects with dynamic storage duration are shared.
  - 28 • Static data members are shared.
  - 29 • The loop iteration variable(s) in the associated *for-loop(s)* of a **for** or **parallel  
30 for** construct is(are) private.
  - Static variables that are declared in a scope inside the construct are shared.
-  C/C++ 

## Fortran

- Variables and common blocks appearing in **threadprivate** directives are **threadprivate**.
- The loop iteration variable(s) in the associated *do-loop(s)* of a **do** or **parallel do** construct is(are) **private**.
- A loop iteration variable for a sequential loop in a **parallel** or **task** construct is **private** in the innermost such construct that encloses the loop.
- **implied-do** and **forall** indices are **private**.
- Cray pointers inherit the data-sharing attribute of the storage with which their Cray pointers are associated.
- Assumed-size arrays are shared.

## Fortran

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

## C/C++

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

## C/C++

## Fortran

- The loop iteration variable(s) in the associated *do-loop(s)* of a **do** or **parallel do** construct may be listed in a **private** or **lastprivate** clause.
- Variables used as loop iteration variables in sequential loops in a **parallel** or **task** construct may be listed in data-sharing clauses on the construct itself, and on enclosed constructs, subject to other restrictions.
- Assumed-size arrays may be listed in a **shared** clause.

## Fortran

Additional restrictions on the variables which may appear in individual clauses are described with each clause in Section 2.9.3 on page 90.

Variables with explicitly determined data-sharing attributes are those that are referenced in a given construct and are listed in a data-sharing attribute clause on the construct.

Variables with implicitly determined data-sharing attributes are those that are referenced in a given construct, do not have predetermined data-sharing attributes, and are not listed in a data-sharing attribute clause on the construct.

Rules for variables with implicitly determined data-sharing attributes are as follows:

- 1 • In a **parallel** or **task** construct, the data-sharing attributes of these variables are  
2 determined by the **default** clause, if present (see Section 2.9.3.1 on page 91).
- 3 • In a **parallel** construct, if no **default** clause is present, these variables are  
4 shared.
- 5 • For constructs other than **task**, if no **default** clause is present, these variables  
6 inherit their data-sharing attributes from the enclosing context.
- 7 • In a **task** construct, if no **default** clause is present, a variable that in the  
8 enclosing context is determined to be shared by all implicit tasks bound to the current  
9 team is shared.

▼ Fortran ▲

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

▲ Fortran ▼

- 12 • In a **task** construct, if no **default** clause is present, a variable whose data-sharing  
13 attribute is not determined by the rules above is **firstprivate**.

14 Additional restrictions on the variables whose data-sharing attributes cannot be  
15 implicitly determined in a **task** construct are described in the *Restrictions* section of  
16 the **firstprivate** clause (Section 2.9.3.4 on page 97).

## 17 2.9.1.2 Data-sharing Attribute Rules for Variables Referenced in a 18 Region but not in a Construct

19 The data-sharing attributes of variables that are referenced in a region, but not in a  
20 construct, are determined as follows:

▼ C/C++ ▲

- 21 • Static variables declared in called routines in the region are shared.
- 22 • Variables with const-qualified type having no mutable member, and that are declared  
23 in called routines, are shared.
- 24 • File-scope or namespace-scope variables referenced in called routines in the region  
25 are shared unless they appear in a **threadprivate** directive.
- 26 • Objects with dynamic storage duration are shared.
- 27 • Static data members are shared unless they appear in a **threadprivate** directive.
- 28 • Formal arguments of called routines in the region that are passed by reference inherit  
29 the data-sharing attributes of the associated actual argument.
- 30 • Other variables declared in called routines in the region are private.

▲ C/C++ ▼

## Fortran

- 1 • Local variables declared in called routines in the region and that have the **save**  
2 attribute, or that are data initialized, are shared unless they appear in a  
3 **threadprivate** directive.
- 4 • Variables belonging to common blocks, or declared in modules, and referenced in  
5 called routines in the region are shared unless they appear in a **threadprivate**  
6 directive.
- 7 • Dummy arguments of called routines in the region that are passed by reference inherit  
8 the data-sharing attributes of the associated actual argument.
- 9 • Cray pointees inherit the data-sharing attribute of the storage with which their Cray  
10 pointers are associated.
- 11 • **implied-do** indices, **forall** indices, and other local variables declared in called  
12 routines in the region are private.

## Fortran

### 13 2.9.2 threadprivate Directive

#### 14 Summary

15 The **threadprivate** directive specifies that variables are replicated, with each thread  
16 having its own copy.

#### 17 Syntax

##### 18 C/C++

The syntax of the **threadprivate** directive is as follows:

```
#pragma omp threadprivate(list) new-line
```

19 where *list* is a comma-separated list of file-scope, namespace-scope, or static  
20 block-scope variables that do not have incomplete types.

##### C/C++

##### Fortran

21 The syntax of the **threadprivate** directive is as follows:

```
!$omp threadprivate(list)
```

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

---

## Fortran

### Description

3 Each copy of a threadprivate variable is initialized once, in the manner specified by the  
4 program, but at an unspecified point in the program prior to the first reference to that  
5 copy. The storage of all copies of a threadprivate variable is freed according to how  
6 static variables are handled in the base language, but at an unspecified point in the  
7 program.  
8

9 A program in which a thread references another thread's copy of a threadprivate variable  
10 is non-conforming.

11 The content of a threadprivate variable can change across a task scheduling point if the  
12 executing thread switches to another task that modifies the variable. For more details on  
13 task scheduling, see Section 1.3 on page 12 and Section 2.7 on page 59.

14 In **parallel** regions, references by the master thread will be to the copy of the  
15 variable in the thread that encountered the **parallel** region.

16 During the sequential part references will be to the initial thread's copy of the variable.  
17 The values of data in the initial thread's copy of a threadprivate variable are guaranteed  
18 to persist between any two consecutive references to the variable in the program.

19 The values of data in the threadprivate variables of non-initial threads are guaranteed to  
20 persist between two consecutive active **parallel** regions only if all the following  
21 conditions hold:

- 22 • Neither **parallel** region is nested inside another explicit **parallel** region.
- 23 • The number of threads used to execute both **parallel** regions is the same.
- 24 • The value of the *dyn-var* internal control variable in the enclosing task region is false  
25 at entry to both **parallel** regions.

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

---

## C/C++

29 If the above conditions hold, the storage duration, lifetime, and value of a thread's copy  
30 of a threadprivate variable that does not appear in any **copyin** clause on the second  
31 region will be retained. Otherwise, the storage duration, lifetime, and value of a thread's  
32 copy of the variable in the second region is unspecified.

1 If the value of a variable referenced in an explicit initializer of a `threadprivate` variable  
2 is modified prior to the first reference to any instance of the `threadprivate` variable, then  
3 the behavior is unspecified.

4 **Note** – The order in which any constructors for different `threadprivate` variables of class  
5 type are called is unspecified. The order in which any destructors for different  
6 `threadprivate` variables of class type are called is unspecified.

### C/C++

### Fortran

7 A variable is affected by a `copyin` clause if the variable appears in the `copyin` clause  
8 or it is in a common block that appears in the `copyin` clause.

9 If the above conditions hold, the definition, association, or allocation status of a thread's  
10 copy of a `threadprivate` variable or a variable in a `threadprivate` common  
11 block, that is not affected by any `copyin` clause that appears on the second region, will  
12 be retained. Otherwise, the definition and association status of a thread's copy of the  
13 variable in the second region is undefined, and the allocation status of an allocatable  
14 array will be implementation defined.

15 If a `threadprivate` variable or a variable in a `threadprivate` common block is  
16 not affected by any `copyin` clause that appears on the first `parallel` region in which  
17 it is referenced, the variable or any subobject of the variable is initially defined or  
18 undefined according to the following rules:

- 19 • If it has the **ALLOCATABLE** attribute, each copy created will have an initial  
20 allocation status of not currently allocated.
- 21 • If it has the **POINTER** attribute:
  - 22 • if it has an initial association status of disassociated, either through explicit  
23 initialization or default initialization, each copy created will have an association  
24 status of disassociated;
  - 25 • otherwise, each copy created will have an association status of undefined.
- 26 • If it does not have either the **POINTER** or the **ALLOCATABLE** attribute:
  - 27 • if it is initially defined, either through explicit initialization or default  
28 initialization, each copy created is so defined;
  - 29 • otherwise, each copy created is undefined.

### Fortran

30 For examples of the `threadprivate` directive, see Section A.24 on page 230.



## Restrictions

The restrictions to the `threadprivate` directive are as follows:

- A `threadprivate` variable must not appear in any clause except the `copyin`, `copyprivate`, `schedule`, `num_threads`, and `if` clauses.
- A program in which an untied task accesses `threadprivate` storage is non-conforming.

### C/C++

- A variable that is part of another variable (as an array or structure element) cannot appear in a `threadprivate` clause unless it is a static data member of a C++ class.
- A `threadprivate` directive for file-scope variables must appear outside any definition or declaration, and must lexically precede all references to any of the variables in its list.
- A `threadprivate` directive for static class member variables must appear in the class definition, in the same scope in which the member variables are declared, and must lexically precede all references to any of the variables in its list.
- A `threadprivate` directive for namespace-scope variables must appear outside any definition or declaration other than the namespace definition itself, and must lexically precede all references to any of the variables in its list.
- Each variable in the list of a `threadprivate` directive at file, namespace, or class scope must refer to a variable declaration at file, namespace, or class scope that lexically precedes the directive.
- A `threadprivate` directive for static block-scope variables must appear in the scope of the variable and not in a nested scope. The directive must lexically precede all references to any of the variables in its list.
- Each variable in the list of a `threadprivate` directive in block scope must refer to a variable declaration in the same scope that lexically precedes the directive. The variable declaration must use the static storage-class specifier.
- If a variable is specified in a `threadprivate` directive in one translation unit, it must be specified in a `threadprivate` directive in every translation unit in which it is declared.
- The address of a `threadprivate` variable is not an address constant.
- A `threadprivate` variable must not have an incomplete type or a reference type.
- A `threadprivate` variable with class type must have:
  - an accessible, unambiguous default constructor in case of default initialization without a given initializer;
  - an accessible, unambiguous constructor accepting the given argument in case of direct initialization;

- 1           • an accessible, unambiguous copy constructor in case of copy initialization with an  
2           explicit initializer.



- 3           • A variable that is part of another variable (as an array or structure element) cannot  
4           appear in a **threadprivate** clause.
- 5           • The **threadprivate** directive must appear in the declaration section of a scoping  
6           unit in which the common block or variable is declared. Although variables in  
7           common blocks can be accessed by use association or host association, common  
8           block names cannot. This means that a common block name specified in a  
9           **threadprivate** directive must be declared to be a common block in the same  
10          scoping unit in which the **threadprivate** directive appears.
- 11          • If a **threadprivate** directive specifying a common block name appears in one  
12          program unit, then such a directive must also appear in every other program unit that  
13          contains a **COMMON** statement specifying the same name. It must appear after the last  
14          such **COMMON** statement in the program unit.
- 15          • A blank common block cannot appear in a **threadprivate** directive.
- 16          • A variable can only appear in a **threadprivate** directive in the scope in which it  
17          is declared. It must not be an element of a common block or appear in an  
18          **EQUIVALENCE** statement.
- 19          • A variable that appears in a **threadprivate** directive must have the **SAVE**  
20          attribute, either explicitly or implicitly.



21           **Cross References:**

- 22           • *dyn-var* ICV, see Section 2.3 on page 28.
- 23           • number of threads used to execute a **parallel** region, see Section 2.4.1 on page 36.
- 24           • **copyin** clause, see Section 2.9.4.1 on page 106.

25           **2.9.3      Data-Sharing Attribute Clauses**

26           Several constructs accept clauses that allow a user to control the data-sharing attributes  
27           of variables referenced in the construct. Data-sharing attribute clauses apply only to  
28           variables whose names are visible in the construct on which the clause appears.

29           Not all of the clauses listed in this section are valid on all directives. The set of clauses  
30           that is valid on a particular directive is described with the directive.

1 Most of the clauses accept a comma-separated list of list items (see Section 2.1 on page  
2 22). All list items appearing in a clause must be visible, according to the scoping rules  
3 of the base language. With the exception of the **default** clause, clauses may be  
4 repeated as needed. A list item that specifies a given variable may not appear in more  
5 than one clause on the same directive, except that a variable may be specified in both  
6 **firstprivate** and **lastprivate** clauses.

#### C/C++

7 If a variable referenced in a data-sharing attribute clause has a type derived from a  
8 template, and there are no other references to that variable in the program, then any  
9 behavior related to that variable is unspecified.

#### C/C++

#### Fortran

10 A named common block may be specified in a list by enclosing the name in slashes.  
11 When a named common block appears in a list, it has the same meaning as if every  
12 explicit member of the common block appeared in the list. An explicit member of a  
13 common block is a variable that is named in a **COMMON** statement that specifies the  
14 common block name and is declared in the same scoping unit in which the clause  
15 appears.

16 Although variables in common blocks can be accessed by use association or host  
17 association, common block names cannot. This means that a common block name  
18 specified in a data-sharing attribute clause must be declared to be a common block in the  
19 same scoping unit in which the data-sharing attribute clause appears.

20 When a named common block appears in a **private**, **firstprivate**,  
21 **lastprivate**, or **shared** clause of a directive, none of its members may be declared  
22 in another data-sharing attribute clause in that directive (see Section A.26 on page 237  
23 for examples). When individual members of a common block appear in a **private**,  
24 **firstprivate**, **lastprivate**, or **reduction** clause of a directive, the storage of  
25 the specified variables is no longer associated with the storage of the common block  
26 itself (see Section A.30 on page 247 for examples).



#### Fortran

### 27 2.9.3.1 **default** clause

#### 28 **Summary**



29 The default clause allows the user to control the data-sharing attributes of variables that  
30 are referenced in a **parallel** or **task** construct, and whose data-sharing attributes are  
31 implicitly determined (see Section 2.9.1.1 on page 83).

## 1       **Syntax**

2        **C/C++**   
The syntax of the **default** clause is as follows:

```
default(shared / none)
```

 **C/C++** 

3        **Fortran**   
The syntax of the **default** clause is as follows:

```
default(private / firstprivate / shared / none)
```

 **Fortran** 

## 4       **Description**

5       The **default(shared)** clause causes all variables referenced in the construct that  
6       have implicitly determined data-sharing attributes to be shared.

 **Fortran** 

7       The **default(firstprivate)** clause causes all variables in the construct that have  
8       implicitly determined data-sharing attributes to be firstprivate.

9       The **default(private)** clause causes all variables referenced in the construct that  
10       have implicitly determined data-sharing attributes to be private.

 **Fortran** 

11       The **default(none)** clause requires that each variable that is referenced in the  
12       construct, and that does not have a predetermined data-sharing attribute, must have its  
13       data-sharing attribute explicitly determined by being listed in a data-sharing attribute  
14       clause. See Section A.27 on page 239 for examples.

## 15       **Restrictions**

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

- 17
  - Only a single default clause may be specified on a **parallel** or **task** directive.

## 1 2.9.3.2 shared clause

### 2 Summary

3 The **shared** clause declares one or more list items to be shared by tasks generated by  
4 a **parallel** or **task** construct.

### 5 Syntax

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

```
shared (list)
```

### 7 Description

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

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

### Fortran

13 The association status of a shared pointer becomes undefined upon entry to and on exit  
14 from the **parallel** or **task** construct if it is associated with a target or a subobject of  
15 a target that is in a **private**, **firstprivate**, **lastprivate**, or **reduction**  
16 clause inside the construct.

17 Under certain conditions, passing a shared variable to a non-intrinsic procedure may  
18 result in the value of the shared variable being copied into temporary storage before the  
19 procedure reference, and back out of the temporary storage into the actual argument  
20 storage after the procedure reference. It is implementation defined when this situation  
21 occurs. See Section A.28 on page 241 for an example of this behavior.

22 **Note** – This situation may occur when the following three conditions hold regarding an  
23 actual argument in a reference to a non-intrinsic procedure:

- 24 a. The actual argument is one of the following:
- 25 • A shared variable.
  - 26 • A subobject of a shared variable.

- 1           • An object associated with a shared variable.
- 2           • An object associated with a subobject of a shared variable.
- 3           b. The actual argument is also one of the following:
- 4           • An array section.
- 5           • An array section with a vector subscript.
- 6           • An assumed-shape array.
- 7           • A pointer array.
- 8           c. The associated dummy argument for this actual argument is an explicit-shape array
- 9           or an assumed-size array.

10           This effectively results in references to, and definitions of, the temporary storage during  
11           the procedure reference. Any references to (or definitions of) the shared storage that is  
12           associated with the dummy argument by any other task must be synchronized with the  
13           procedure reference to avoid possible race conditions.



### 14 **2.9.3.3 private clause**

#### 15 **Summary**

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

#### 17 **Syntax**

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

```
private (list)
```

#### 19 **Description**

20           Each task that references a list item that appears in a **private** clause in any statement  
21           in the construct receives a new list item whose language-specific attributes are derived  
22           from the original list item. Inside the construct, all references to the original list item are  
23           replaced by references to the new list item. In the rest of the region, it is unspecified  
24           whether references are to the new list item or the original list item. Therefore, if an

1 attempt is made to reference the original item, its value after the region is also  
2 unspecified. If a task does not reference a list item that appears in a **private** clause, it  
3 is unspecified whether that task receives a new list item.

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

- 5 • if accessed and modified via pointer,
- 6 • if (possibly) accessed in the region but outside of the construct, or
- 7 • as a side effect of directives or clauses.

8 List items that appear in a **private**, **firstprivate**, or **reduction** clause in a  
9 **parallel** construct may also appear in a **private** clause in an enclosed **parallel**,  
10 **task**, or worksharing construct. List items that appear in a **private** or  
11 **firstprivate** clause in a **task** construct may also appear in a **private** clause in  
12 an enclosed **parallel** or **task** construct. List items that appear in a **private**,  
13 **firstprivate**, **lastprivate**, or **reduction** clause in a worksharing construct  
14 may also appear in a **private** clause in an enclosed **parallel** or **task** construct.  
15 See Section A.29 on page 242 for an example.

#### C/C++

16 A new list item of the same type, with automatic storage duration, is allocated for the  
17 construct. The storage and thus lifetime of these list items lasts until the block in which  
18 they are created exits. The size and alignment of the new list item are determined by the  
19 type of the variable. This allocation occurs once for each task generated by the  
20 construct, if the task references the list item in any statement.

21 The new list item is initialized, or has an undefined initial value, as if it had been locally  
22 declared without an initializer. The order in which any default constructors for different  
23 private variables of class type are called is unspecified. The order in which any  
24 destructors for different private variables of class type are called is unspecified.

#### C/C++

#### Fortran

25 A new list item of the same type is allocated once for each implicit task in the  
26 **parallel** region, or for each task generated by a **task** construct, if the construct  
27 references the list item in any statement. The initial value of the new list item is  
28 undefined. Within a **parallel**, **worksharing**, or **task** region, the initial status of a  
29 **private** pointer is undefined.

30 For a list item with the **ALLOCATABLE** attribute:

- 31 • if the list item is "not currently allocated", the new list item will have an initial state  
32 of "not currently allocated";
- 33 • if the list item is allocated, the new list item will have an initial state of allocated  
34 with the same array bounds.

1 A list item that appears in a **private** clause may be storage-associated with other  
2 variables when the **private** clause is encountered. Storage association may exist  
3 because of constructs such as **EQUIVALENCE** or **COMMON**. If *A* is a variable appearing  
4 in a **private** clause and *B* is a variable that is storage-associated with *A*, then:

- 5 • The contents, allocation, and association status of *B* are undefined on entry to the  
6 **parallel** or **task** region.
- 7 • Any definition of *A*, or of its allocation or association status, causes the contents,  
8 allocation, and association status of *B* to become undefined.
- 9 • Any definition of *B*, or of its allocation or association status, causes the contents,  
10 allocation, and association status of *A* to become undefined.

11 For examples, see Section A.30 on page 247.

Fortran

12 For examples of the **private** clause, see Section A.29 on page 242.

## 13 Restrictions

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

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

C/C++

- 17 • A variable of class type (or array thereof) that appears in a **private** clause requires  
18 an accessible, unambiguous default constructor for the class type.
- 19 • A variable that appears in a **private** clause must not have a **const**-qualified type  
20 unless it is of class type with a **mutable** member. This restriction does not apply to  
21 the **firstprivate** clause.
- 22 • A variable that appears in a **private** clause must not have an incomplete type or a  
23 reference type.

C/C++

Fortran

- 24 • A variable that appears in a **private** clause must either be definable, or an  
25 allocatable array. This restriction does not apply to the **firstprivate** clause.
- 26 • Assumed-size arrays may not appear in a **private** clause.
- 27 • Variables that appear in namelist statements, in variable format expressions, and in  
28 expressions for statement function definitions, may not appear in a **private** clause.

Fortran



## 1 2.9.3.4 `firstprivate` clause

### 2 **Summary**

3 The `firstprivate` clause declares one or more list items to be private to a task, and  
4 initializes each of them with the value that the corresponding original item has when the  
5 construct is encountered.

### 6 **Syntax**

7 The syntax of the `firstprivate` clause is as follows:

<code>firstprivate</code> ( <i>list</i> )
---

### 8 **Description**

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

11 A list item that appears in a `firstprivate` clause is subject to the `private` clause  
12 semantics described in Section 2.9.3.3 on page 94, except as noted. In addition, the new  
13 list item is initialized from the original list item existing before the construct. The  
14 initialization of the new list item is done once for each task that references the list item  
15 in any statement in the construct. The initialization is done prior to the execution of the  
16 construct.

17 For a `firstprivate` clause on a `parallel` or `task` construct, the initial value of  
18 the new list item is the value of the original list item that exists immediately prior to the  
19 construct in the task region where the construct is encountered. For a `firstprivate`  
20 clause on a worksharing construct, the initial value of the new list item for each implicit  
21 task of the threads that execute the worksharing construct is the value of the original list  
22 item that exists in the implicit task immediately prior to the point in time that the  
23 worksharing construct is encountered.

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

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

## C/C++

1 For variables of non-array type, the initialization occurs by copy assignment. For a  
2 (possibly multi-dimensional) array of elements of non-array type, each element is  
3 initialized as if by assignment from an element of the original array to the corresponding  
4 element of the new array. For variables of class type, a copy constructor is invoked to  
5 perform the initialization. The order in which copy constructors for different variables of  
6 class type are called is unspecified.

## C/C++

## Fortran

7 If the original list item does not have the **POINTER** attribute, initialization of the new  
8 list items occurs as if by intrinsic assignment, unless the original list item has the  
9 allocation status of not currently allocated, in which case the new list items will have the  
10 same status.

11 If the original list item has the **POINTER** attribute, the new list items receive the same  
12 association status of the original list item as if by pointer assignment.

## Fortran

## Restrictions

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

- 14 • A variable that is part of another variable (as an array or structure element) cannot  
15 appear in a **firstprivate** clause.
- 16 • A list item that is private within a **parallel** region must not appear in a  
17 **firstprivate** clause on a worksharing construct if any of the worksharing  
18 regions arising from the worksharing construct ever bind to any of the **parallel**  
19 regions arising from the **parallel** construct.
- 20 • A list item that appears in a **reduction** clause of a **parallel** construct must not  
21 appear in a **firstprivate** clause on a worksharing or **task** construct if any of  
22 the worksharing or **task** regions arising from the worksharing or **task** construct  
23 ever bind to any of the **parallel** regions arising from the **parallel** construct.
- 24 • A list item that appears in a **reduction** clause in a worksharing construct must not  
25 appear in a **firstprivate** clause in a task construct encountered during execution  
26 of any of the worksharing regions arising from the worksharing construct.
- 27

## C/C++

- 28 • A variable of class type (or array thereof) that appears in a **firstprivate** clause  
29 requires an accessible, unambiguous copy constructor for the class type.

- 1 • A variable that appears in a **firstprivate** clause must not have an incomplete  
2 type or a reference type.

▲ C/C++ ▲

- 3 • Assumed-size arrays may not appear in a **firstprivate** clause.  
4 • Variables that appear in namelist statements, in variable format expressions, and in  
5 expressions for statement function definitions, may not appear in a **firstprivate**  
6 clause.

▲ Fortran ▲

## 7 2.9.3.5 **lastprivate** clause

### 8 **Summary**

9 The **lastprivate** clause declares one or more list items to be private to an implicit  
10 task, and causes the corresponding original list item to be updated after the end of the  
11 region.

### 12 **Syntax**

13 The syntax of the **lastprivate** clause is as follows:

```
14 lastprivate (list)
```

### 14 **Description**

15 The **lastprivate** clause provides a superset of the functionality provided by the  
16 **private** clause.

17 A list item that appears in a **lastprivate** clause is subject to the **private** clause  
18 semantics described in Section 2.9.3.3 on page 94. In addition, when a **lastprivate**  
19 clause appears on the directive that identifies a worksharing construct, the value of each  
20 new list item from the sequentially last iteration of the associated loops, or the lexically  
21 last **section** construct, is assigned to the original list item.

## C/C++

1 For a (possibly multi-dimensional) array of elements of non-array type, each element is  
2 assigned to the corresponding element of the original array.

## C/C++

## Fortran

3 If the original list item does not have the **POINTER** attribute, its update occurs as if by  
4 intrinsic assignment.

5 If the original list item has the **POINTER** attribute, its update occurs as if by pointer  
6 assignment.

## Fortran

7 List items that are not assigned a value by the sequentially last iteration of the loops, or  
8 by the lexically last **section** construct, have unspecified values after the construct.  
9 Unassigned subcomponents also have unspecified values after the construct.

10 The original list item becomes defined at the end of the construct if there is an implicit  
11 barrier at that point. To avoid race conditions, concurrent reads or updates of the original  
12 list item must be synchronized with the update of the original list item that occurs as a  
13 result of the **lastprivate** clause.

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

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

20 For an example of the **lastprivate** clause, see Section A.32 on page 251.

## Restrictions

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

- 22 • A variable that is part of another variable (as an array or structure element) cannot  
23 appear in a **lastprivate** clause.
- 24 • A list item that is private within a **parallel** region, or that appears in the  
25 **reduction** clause of a **parallel** construct, must not appear in a **lastprivate**  
26 clause on a worksharing construct if any of the corresponding worksharing regions  
27 ever binds to any of the corresponding **parallel** regions.

### C/C++

- 1 • A variable of class type (or array thereof) that appears in a **lastprivate** clause  
2 requires an accessible, unambiguous default constructor for the class type, unless the  
3 list item is also specified in a **firstprivate** clause.
- 4 • A variable of class type (or array thereof) that appears in a **lastprivate** clause  
5 requires an accessible, unambiguous copy assignment operator for the class type. The  
6 order in which copy assignment operators for different variables of class type are  
7 called is unspecified.
- 8 • A variable that appears in a **lastprivate** clause must not have a **const**-qualified  
9 type unless it is of class type with a **mutable** member.
- 10 • A variable that appears in a **lastprivate** clause must not have an incomplete type  
11 or a reference type.

### C/C++

### Fortran

- 12 • A variable that appears in a **lastprivate** clause must be definable.
- 13 • An original list item with the **ALLOCATABLE** attribute must be in the allocated state  
14 at entry to the construct containing the **lastprivate** clause. The list item in the  
15 sequentially last iteration or lexically last section must be in the allocated state upon  
16 exit from that iteration or section with the same bounds as the corresponding original  
17 list item.
- 18 • Assumed-size arrays may not appear in a **lastprivate** clause.
- 19 • Variables that appear in namelist statements, in variable format expressions, and in  
20 expressions for statement function definitions, may not appear in a **lastprivate**  
21 clause.

### Fortran

## 22 2.9.3.6 reduction clause

### 23 Summary

24 The **reduction** clause specifies an operator and one or more list items. For each list  
25 item, a private copy is created in each implicit task, and is initialized appropriately for  
26 the operator. After the end of the region, the original list item is updated with the values  
27 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 list item.

Operator	Initialization value
+	0
*	1
-	0
&	~0
	0
^	0
&&	1
	0
<b>max</b>	Least representable value in the reduction list item type
<b>min</b>	Largest representable value in the reduction list item type

### C/C++

### Fortran

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

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

1  
2  
3

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 list item.

---

Operator/ Intrinsic	Initialization value
<code>+</code>	<code>0</code>
<code>*</code>	<code>1</code>
<code>-</code>	<code>0</code>
<code>.and.</code>	<code>.true.</code>
<code>.or.</code>	<code>.false.</code>
<code>.eqv.</code>	<code>.true.</code>
<code>.neqv.</code>	<code>.false.</code>
<code>max</code>	Least representable number in the reduction list item type
<code>min</code>	Largest representable number in the reduction list item type
<code>iand</code>	All bits on
<code>ior</code>	<code>0</code>
<code>ieor</code>	<code>0</code>

---



4  
5  
6  
7  
8  
9  
10  
11  
12

## 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 implicit task, 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.)

## C/C++

1 For max and min operators, the final values of the private copies are combined with the  
2 original list item value using the following expressions:

---

```
max  original_list_item =  
      original_list_item < private_copy ? private_copy : original_list_item;  
min  original_list_item =  
      original_list_item > private_copy ? private_copy : original_list_item;
```

---

## C/C++

3 If **nowait** is not used, the reduction computation will be complete at the end of the  
4 construct; however, if the reduction clause is used on a construct to which **nowait** is  
5 also applied, accesses to the original list item will create a race and, thus, have  
6 unspecified effect unless synchronization ensures that they occur after all threads have  
7 executed all of their iterations or **section** constructs, and the reduction computation  
8 has completed and stored the computed value of that list item. This can most simply be  
9 ensured through a barrier synchronization.

10 The order in which the values are combined is unspecified. Therefore, comparing  
11 sequential and parallel runs, or comparing one parallel run to another (even if the  
12 number of threads used is the same), there is no guarantee that bit-identical results will  
13 be obtained or that side effects (such as floating point exceptions) will be identical.

14 To avoid race conditions, concurrent reads or updates of the original list item must be  
15 synchronized with the update of the original list item that occurs as a result of the  
16 **reduction** computation.

## 17 Restrictions

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

- 19 • A list item that appears in a **reduction** clause of a worksharing construct must be  
20 shared in the **parallel** regions to which any of the worksharing regions arising  
21 from the worksharing construct bind.
- 22 • A list item that appears in a **reduction** clause of the innermost enclosing  
23 worksharing or **parallel** construct may not be accessed in an explicit task.
- 24 • Any number of **reduction** clauses can be specified on the directive, but a list item  
25 can appear only once in the **reduction** clause(s) for that directive.



## C/C++

- 1 • The type of a list item that appears in a **reduction** clause must be valid for the  
2 reduction operator. For a **max** or **min** reduction in C, the type of the list item must be  
3 an arithmetic data type; the allowed data types are **char**, **int**, **float**, **double**, or  
4 **\_Bool**, possibly modified with **long**, **short**, **signed**, or **unsigned**. For a **max**  
5 or **min** reduction in C++, the type of the list item must be an arithmetic data type; the  
6 allowed data types are **char**, **wchar\_t**, **int**, **float**, **double**, or **bool**, possibly  
7 modified with **long**, **short**, **signed**, or **unsigned**.
- 8 • Aggregate types (including arrays), pointer types and reference types may not appear  
9 in a **reduction** clause.
- 10 • A list item that appears in a **reduction** clause must not be **const**-qualified.

## C/C++

## Fortran

- 11 • The type of a list item that appears in a **reduction** clause must be valid for the  
12 reduction operator or intrinsic.
- 13 • A list item that appears in a **reduction** clause must be definable.
- 14 • A list item that appears in a **reduction** clause must be a named variable of  
15 intrinsic type.
- 16 • An original list item with the **ALLOCATABLE** attribute must be in the allocated state  
17 at entry to the construct containing the reduction clause. Additionally, the list item  
18 must not be deallocated and/or allocated within the region.
- 19 • Fortran pointers, Cray pointers and assumed-size arrays may not appear in a  
20 **reduction** clause.
- 21 • Operators specified must be intrinsic operators and any *intrinsic\_procedure\_name*  
22 must refer to one of the allowed intrinsic procedures. Assignment to the reduction list  
23 items must be via intrinsic assignment. See Section A.33 on page 252 for examples.

## Fortran

### 24 2.9.4 Data Copying Clauses

25 This section describes the **copyin** clause (valid on the **parallel** directive and  
26 combined parallel worksharing directives) and the **copyprivate** clause (valid on the  
27 **single** directive).

28 These clauses support the copying of data values from private or threadprivate variables  
29 on one implicit task or thread to the corresponding variables on other implicit tasks or  
30 threads in the team.

1 The clauses accept a comma-separated list of list items (see Section 2.1 on page 22). All  
2 list items appearing in a clause must be visible, according to the scoping rules of the  
3 base language. Clauses may be repeated as needed, but a list item that specifies a given  
4 variable may not appear in more than one clause on the same directive.

## 5 **2.9.4.1** **copyin clause**

### 6 **Summary**

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

### 10 **Syntax**

11 The syntax of the **copyin** clause is as follows:

```
copyin(list)
```

### 12 **Description**

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

20 **C/C++**

21 **Fortran**

22 The copy is done, as if by assignment, after the team is formed and prior to the start of  
23 execution of the associated structured block.

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

- If it has the **POINTER** attribute, each copy receives the same association status of the master thread's copy as if by pointer assignment.

- 1 • If it does not have the **POINTER** attribute, each copy becomes defined with the value  
2 of the master thread's copy as if by intrinsic assignment.

Fortran

3 For an example of the **copyin** clause, see Section A.34 on page 258.

## 4 Restrictions

5 The restrictions to the **copyin** clause are as follows:

C/C++

- 6 • A list item that appears in a **copyin** clause must be threadprivate.  
7 • A variable of class type (or array thereof) that appears in a **copyin** clause requires  
8 an accessible, unambiguous copy assignment operator for the class type.

C/C++

Fortran

- 9 • A list item that appears in a **copyin** clause must be threadprivate. Named variables  
10 appearing in a threadprivate common block may be specified: it is not necessary to  
11 specify the whole common block.  
12 • A common block name that appears in a **copyin** clause must be declared to be a  
13 common block in the same scoping unit in which the **copyin** clause appears.  
14 • If an array with the **ALLOCATABLE** attribute is allocated, then each thread's copy of  
15 that array must be allocated with the same bounds.

Fortran

### 16 2.9.4.2 **copyprivate** clause

## 17 Summary

18 The **copyprivate** clause provides a mechanism to use a private variable to broadcast  
19 a value from the data environment of one implicit task to the data environments of the  
20 other implicit tasks belonging to the **parallel** region.

21 To avoid race conditions, concurrent reads or updates of the list item must be  
22 synchronized with the update of the list item that occurs as a result of the  
23 **copyprivate** clause.

## Syntax

The syntax of the `copyprivate` clause is as follows:

```
copyprivate (list)
```

## Description

The effect of the `copyprivate` clause on the specified list items occurs after the execution of the structured block associated with the `single` construct (see Section 2.5.3 on page 49), and before any of the threads in the team have left the barrier at the end of the construct.

### C/C++

In all other implicit tasks belonging to the `parallel` region, each specified list item becomes defined with the value of the corresponding list item in the implicit task whose thread executed the structured block. For variables of non-array type, the definition occurs by copy assignment. For a (possibly multi-dimensional) array of elements of non-array type, each element is copied by copy assignment from an element of the array in the data environment of the implicit task whose thread executed the structured block to the corresponding element of the array in the data environment of the other implicit tasks. For class types, a copy assignment operator is invoked. The order in which copy assignment operators for different variables of class type are called is unspecified.

### C/C++

### Fortran

If a list item does not have the `POINTER` attribute, then in all other implicit tasks belonging to the `parallel` region, the list item becomes defined as if by intrinsic assignment with the value of the corresponding list item in the implicit task whose thread executed the structured block. If the list item has the `POINTER` attribute, then in all other implicit tasks belonging to the `parallel` region, the list item receives the same association status (as if by pointer assignment) of the corresponding list item in the implicit task whose thread executed the structured block.

### Fortran

For examples of the `copyprivate` clause, see Section A.35 on page 260.

**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`.
- Assumed-size arrays may not appear in a `copyprivate` clause.
- An array with the `ALLOCATABLE` attribute must be in the allocated state with the same bounds in all threads affected by the `copyprivate` clause.

### Fortran

---

## 2.10 Nesting of Regions

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

- A worksharing region may not be closely nested inside a worksharing, explicit `task`, `critical`, `ordered`, `atomic`, or `master` region.
- A `barrier` region may not be closely nested inside a worksharing, explicit `task`, `critical`, `ordered`, `atomic`, or `master` region.
- A `master` region may not be closely nested inside a worksharing, `atomic`, or explicit `task` region.
- An `ordered` region may not be closely nested inside a `critical`, `atomic`, or explicit `task` region.
- An `ordered` region must be closely nested inside a loop region (or parallel loop region) with an `ordered` clause.
- A `critical` region may not be nested (closely or otherwise) inside a `critical` region with the same name. Note that this restriction is not sufficient to prevent deadlock.

1           • **parallel**, **flush**, **critical**, **atomic**, **taskyield**, and explicit **task**  
2           regions may not be closely nested inside an **atomic** region.

3           For examples illustrating these rules, see Section A.17 on page 209, Section A.36 on  
4           page 265, Section A.37 on page 268, and Section A.13 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 112).
- Execution environment routines that can be used to control and query the parallel execution environment (Section 3.2 on page 113).
- Lock routines that can be used to synchronize access to data (Section 3.3 on page 139).
- Portable timer routines (Section 3.4 on page 146).

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 ▼

---

## 1 3.1 Runtime Library Definitions

2 For each base language, a compliant implementation must supply a set of definitions for  
3 the OpenMP API runtime library routines and the special data types of their parameters.  
4 The set of definitions must contain a declaration for each OpenMP API runtime library  
5 routine and a declaration for the *simple lock*, *nestable lock* and *schedule* data types. In  
6 addition, each set of definitions may specify other implementation specific values.

---

### C/C++

7 The library routines are external functions with “C” linkage.

8 Prototypes for the C/C++ runtime library routines described in this chapter shall be  
9 provided in a header file named **omp.h**. This file defines the following:

- 10 • The prototypes of all the routines in the chapter.
- 11 • The type **omp\_lock\_t**.
- 12 • The type **omp\_nest\_lock\_t**.
- 13 • The type **omp\_sched\_t**.

14 See Section D.1 on page 312 for an example of this file.

---

### C/C++

---

### Fortran

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

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

21 These files define the following:

- 22 • The interfaces of all of the routines in this chapter.
- 23 • The **integer parameter** **omp\_lock\_kind**.
- 24 • The **integer parameter** **omp\_nest\_lock\_kind**.
- 25 • The **integer parameter** **omp\_sched\_kind**.
- 26 • The **integer parameter** **openmp\_version** with a value *yyyymm* where *yyyy*  
27 and *mm* are the year and month designations of the version of the OpenMP Fortran  
28 API that the implementation supports. This value matches that of the C preprocessor  
29 macro **\_OPENMP**, when a macro preprocessor is supported (see Section 2.2 on page  
30 26).



1 See Section D.2 on page 314 and Section D.3 on page 316 for examples of these files.

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

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 • the `omp_set_schedule` routine.
- 19 • the `omp_get_schedule` routine.
- 20 • the `omp_get_thread_limit` routine.
- 21 • the `omp_set_max_active_levels` routine.
- 22 • the `omp_get_max_active_levels` routine.
- 23 • the `omp_get_level` routine.
- 24 • the `omp_get_ancestor_thread_num` routine.
- 25 • the `omp_get_team_size` routine.
- 26 • the `omp_get_active_level` routine.

## 1 3.2.1 `omp_set_num_threads`

### 2 Summary

3 The `omp_set_num_threads` routine affects the number of threads to be used for  
4 subsequent parallel regions that do not specify a `num_threads` clause, by setting the  
5 value of the first element of the `nthreads-var` ICV of the current task.

### 6 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 ▲

### 7 Constraints on Arguments

8 The value of the argument passed to this routine must evaluate to a positive integer, or  
9 else the behavior of this routine is implementation defined.

### 10 Binding

11 The binding task set for an `omp_set_num_threads` region is the generating task.

### 12 Effect

13 The effect of this routine is to set the value of the first element of the `nthreads-var` ICV  
14 of the current task to the value specified in the argument.

15 See Section 2.4.1 on page 36 for the rules governing the number of threads used to  
16 execute a `parallel` region.

1 For an example of the `omp_set_num_threads` routine, see Section A.38 on page  
2 275.

### 3 **Cross References**

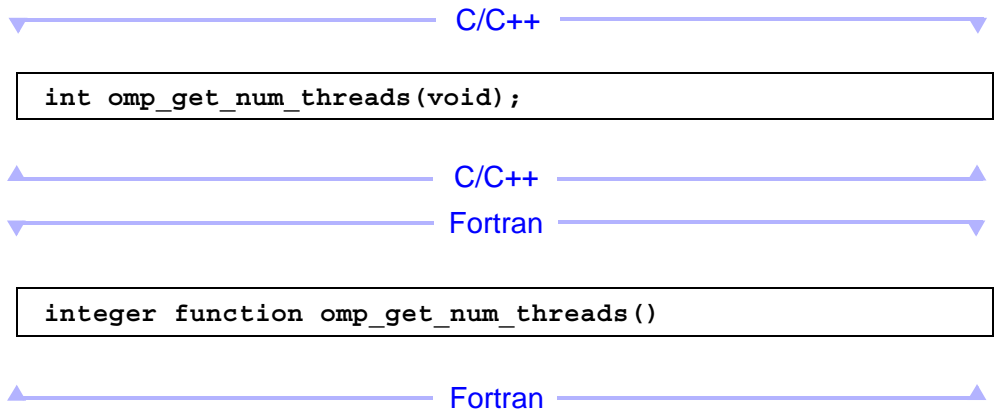
- 4 • *nthreads-var* ICV, see Section 2.3 on page 28.
- 5 • `OMP_NUM_THREADS` environment variable, see Section 4.2 on page 153.
- 6 • `omp_get_max_threads` routine, see Section 3.2.3 on page 116.
- 7 • `parallel` construct, see Section 2.4 on page 32.
- 8 • `num_threads` clause, see Section 2.4 on page 32.

## 9 **3.2.2 `omp_get_num_threads`**

### 10 **Summary**

11 The `omp_get_num_threads` routine returns the number of threads in the current  
12 team.

### 13 **Format**



### 14 **Binding**

15 The binding region for an `omp_get_num_threads` region is the innermost enclosing  
16 `parallel` region.

1  
2  
3  
4  
5  
6  
7  
8  
9  
10  
11  
12  
13  
14  
15  
16  
17

### 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.39 on page 276.

See Section 2.4.1 on page 36 for the rules governing the number of threads used to execute a `parallel` region.

### Cross References

- `parallel` construct, see Section 2.4 on page 32.
- `omp_set_num_threads` routine, see Section 3.2.1 on page 114.
- `OMP_NUM_THREADS` environment variable, see Section 4.2 on page 153.

## 3.2.3 `omp_get_max_threads`

### Summary

The `omp_get_max_threads` routine returns an upper bound on the number of threads that could be used to form a new team if a `parallel` region without a `num_threads` clause were encountered after execution returns from this routine.

### Format

C/C++

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

C/C++

Fortran

```
integer function omp_get_max_threads()
```

Fortran

1           **Binding**

2           The binding task set for an `omp_get_max_threads` region is the generating task.

3           **Effect**

4           The value returned by `omp_get_max_threads` is the value of the first element of  
5           the *nthreads-var* ICV of the current task. This value is also an upper bound on the  
6           number of threads that could be used to form a new team if a parallel region without a  
7           **num\_threads** clause were encountered after execution returns from this routine.

8           See Section 2.4.1 on page 36 for the rules governing the number of threads used to  
9           execute a `parallel` region.

10          ▼  
11          **Note** – The return value of the `omp_get_max_threads` routine can be used to  
12          dynamically allocate sufficient storage for all threads in the team formed at the  
            subsequent active `parallel` region.  
            ▲

13          **Cross References**

- 14          • *nthreads-var* ICV, see Section 2.3 on page 28.
- 15          • `parallel` construct, see Section 2.4 on page 32.
- 16          • `num_threads` clause, see Section 2.4 on page 32.
- 17          • `omp_set_num_threads` routine, see Section 3.2.1 on page 114.
- 18          • `OMP_NUM_THREADS` environment variable, see Section 4.2 on page 153.

19 **3.2.4**       **omp\_get\_thread\_num**

20          **Summary**

21          The `omp_get_thread_num` routine returns the thread number, within the current  
22          team, of the thread executing the implicit or explicit `task` region from which  
23          **omp\_get\_thread\_num** is called.

1

## Format

▼ C/C++ ▼

```
int omp_get_thread_num(void);
```

▲ C/C++ ▲

▼ Fortran ▼

```
integer function omp_get_thread_num()
```

▲ Fortran ▲

2

## Binding

3

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.

4

5

6

## Effect

7

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.

8

9

10

11

▼

12

**Note** – The thread number may change at any time during the execution of an untied task. The value returned by `omp_get_thread_num` is not generally useful during the execution of such a task region.

13

14

▲

15

## Cross References

16

- `omp_get_num_threads` routine, see Section 3.2.2 on page 115.

## 1 3.2.5 `omp_get_num_procs`

### 2 Summary

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

### 5 Format

C/C++

```
int omp_get_num_procs(void);
```

C/C++

Fortran

```
integer function omp_get_num_procs()
```

Fortran

### 6 Binding

7 The binding thread set for an `omp_get_num_procs` region is all threads. The effect  
8 of executing this routine is not related to any specific region corresponding to any  
9 construct or API routine.

### 10 Effect

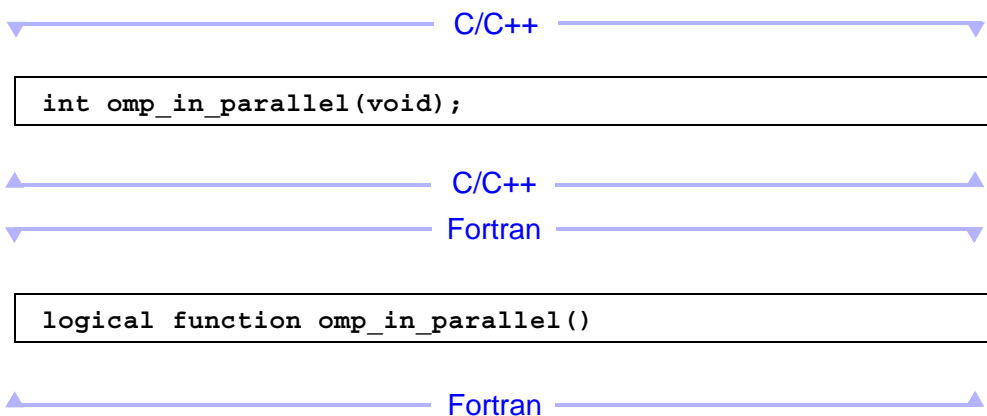
11 The `omp_get_num_procs` routine returns the number of processors that are available  
12 to the program at the time the routine is called. Note that this value may change between  
13 the time that it is determined by the `omp_get_num_procs` routine and the time that it  
14 is read in the calling context due to system actions outside the control of the OpenMP  
15 implementation.

## 1 3.2.6 `omp_in_parallel`

### 2 Summary

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

### 5 Format



### 6 Binding

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

### 10 Effect

11 `omp_in_parallel` returns *true* if any enclosing `parallel` region is active. If the  
12 routine call is enclosed by only inactive `parallel` regions (including the implicit  
13 `parallel` region), then it returns *false*.



## 1 3.2.7 `omp_set_dynamic`

### 2 Summary

3 The `omp_set_dynamic` routine enables or disables dynamic adjustment of the  
4 number of threads available for the execution of subsequent `parallel` regions by  
5 setting the value of the *dyn-var* ICV.

### 6 Format

C/C++

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

C/C++

Fortran

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

Fortran

### 7 Binding

8 The binding task set for an `omp_set_dynamic` region is the generating task.

### 9 Effect

10 For implementations that support dynamic adjustment of the number of threads, if the  
11 argument to `omp_set_dynamic` evaluates to *true*, dynamic adjustment is enabled for  
12 the current task; otherwise, dynamic adjustment is disabled for the current task. For  
13 implementations that do not support dynamic adjustment of the number of threads this  
14 routine has no effect: the value of *dyn-var* remains false.

15 For an example of the `omp_set_dynamic` routine, see Section A.38 on page 275.

16 See Section 2.4.1 on page 36 for the rules governing the number of threads used to  
17 execute a `parallel` region.

## Cross References:

- *dyn-var* ICV, see Section 2.3 on page 28.
- `omp_get_num_threads` routine, see Section 3.2.2 on page 115.
- `omp_get_dynamic` routine, see Section 3.2.8 on page 122.
- `OMP_DYNAMIC` environment variable, see Section 4.3 on page 154.

## 3.2.8 `omp_get_dynamic`

### Summary

The `omp_get_dynamic` routine returns the value of the *dyn-var* ICV, which determines whether dynamic adjustment of the number of threads is enabled or disabled.

### Format

C/C++

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

C/C++

Fortran

```
logical function omp_get_dynamic()
```

Fortran

### Binding

The binding task set for an `omp_get_dynamic` region is the generating task.

### Effect

This routine returns *true* if dynamic adjustment of the number of threads is enabled for the current task; it returns *false*, otherwise. If an implementation does not support dynamic adjustment of the number of threads, then this routine always returns *false*.

1 See Section 2.4.1 on page 36 for the rules governing the number of threads used to  
2 execute a **parallel** region.

### 3 **Cross References**

- 4 • *dyn-var* ICV, see Section 2.3 on page 28.
- 5 • **omp\_set\_dynamic** routine, see Section 3.2.7 on page 121.
- 6 • **OMP\_DYNAMIC** environment variable, see Section 4.3 on page 154.

## 7 **3.2.9 omp\_set\_nested**

### 8 **Summary**

9 The **omp\_set\_nested** routine enables or disables nested parallelism, by setting the  
10 *nest-var* ICV.

### 11 **Format**

▼ C/C++ ▼

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

▲ C/C++ ▲

▼ Fortran ▼

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

▲ Fortran ▲

### 12 **Binding**

13 The binding task set for an **omp\_set\_nested** region is the generating task.

## Effect

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

See Section 2.4.1 on page 36 for the rules governing the number of threads used to execute a `parallel` region.

## Cross References

- *nest-var* ICV, see Section 2.3 on page 28.
- `omp_set_max_active_levels` routine, see Section 3.2.14 on page 130.
- `omp_get_max_active_levels` routine, see Section 3.2.15 on page 131.
- `omp_get_nested` routine, see Section 3.2.10 on page 124.
- `OMP_NESTED` environment variable, see Section 4.5 on page 155.

## 3.2.10 `omp_get_nested`

### Summary

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

### Format

C/C++

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

C/C++

Fortran

```
logical function omp_get_nested()
```

## 1 Binding

2 The binding task set for an `omp_get_nested` region is the generating task.

## 3 Effect

4 This routine returns *true* if nested parallelism is enabled for the current task; it returns  
5 *false*, otherwise. If an implementation does not support nested parallelism, this routine  
6 always returns *false*.

7 See Section 2.4.1 on page 36 for the rules governing the number of threads used to  
8 execute a `parallel` region.

## 9 Cross References

- 10 • *nest-var* ICV, see Section 2.3 on page 28.
- 11 • `omp_set_nested` routine, see Section 3.2.9 on page 123.
- 12 • `OMP_NESTED` environment variable, see Section 4.5 on page 155.

## 13 3.2.11 `omp_set_schedule`

### 14 Summary

15 The `omp_set_schedule` routine affects the schedule that is applied when `runtime`  
16 is used as schedule kind, by setting the value of the *run-sched-var* ICV.

### 17 Format

C/C++

```
void omp_set_schedule(omp_sched_t kind, int modifier);
```

C/C++

## Fortran

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

## Fortran

1  
2  
3  
4  
5  
6

### Constraints on Arguments

The first argument passed to this routine can be one of the valid OpenMP schedule kinds (except for **runtime**) or any implementation specific schedule. The C/C++ header file (**omp.h**) and the Fortran include file (**omp\_lib.h**) and/or Fortran 90 module file (**omp\_lib**) define the valid constants. The valid constants must include the following, which can be extended with implementation specific values:

## C/C++

```
typedef enum omp_sched_t {
  omp_sched_static = 1,
  omp_sched_dynamic = 2,
  omp_sched_guided = 3,
  omp_sched_auto = 4
} omp_sched_t;
```

## C/C++

## Fortran

```
integer(kind=omp_sched_kind), parameter :: omp_sched_static = 1
integer(kind=omp_sched_kind), parameter :: omp_sched_dynamic = 2
integer(kind=omp_sched_kind), parameter :: omp_sched_guided = 3
integer(kind=omp_sched_kind), parameter :: omp_sched_auto = 4
```

## Fortran

1

## Binding

2

The binding task set for an `omp_set_schedule` region is the generating task.

3

## Effect

4

The effect of this routine is to set the value of the *run-sched-var* ICV of the current task to the values specified in the two arguments. The schedule is set to the schedule type

5

specified by the first argument **kind**. It can be any of the standard schedule types or

6

any other implementation specific one. For the schedule types **static**, **dynamic**, and

7

**guided** the *chunk\_size* is set to the value of the second argument, or to the default

8

*chunk\_size* if the value of the second argument is less than 1; for the schedule type

9

**auto** the second argument has no meaning; for implementation specific schedule types,

10

the values and associated meanings of the second argument are implementation defined.

11

12

## Cross References

13

- *run-sched-var* ICV, see Section 2.3 on page 28.

14

- `omp_get_schedule` routine, see Section 3.2.12 on page 127.

15

- `OMP_SCHEDULE` environment variable, see Section 4.1 on page 152.

16

- Determining the schedule of a worksharing loop, see Section 2.5.1.1 on page 46.

## 17 3.2.12

## `omp_get_schedule`

18

## Summary

19

The `omp_get_schedule` routine returns the schedule that is applied when the

20

`runtime` schedule is used.

1

## Format

C/C++

```
void omp_get_schedule(omp_sched_t * kind, int * modifier );
```

C/C++

Fortran

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

Fortran

2

## Binding

3

The binding task set for an `omp_get_schedule` region is the generating task.

4

## Effect

5

This routine returns the run-sched-var ICV in the task to which the routine binds. The first argument `kind` returns the schedule to be used. It can be any of the standard schedule types as defined in Section 3.2.11 on page 125, or any implementation specific schedule type. The second argument is interpreted as in the `omp_set_schedule` call, defined in Section 3.2.11 on page 125.

6

7

8

9

10

## Cross References

11

- *run-sched-var* ICV, see Section 2.3 on page 28.

12

- `omp_set_schedule` routine, see Section 3.2.11 on page 125.

13

- `OMP_SCHEDULE` environment variable, see Section 4.1 on page 152.

14

- Determining the schedule of a worksharing loop, see Section 2.5.1.1 on page 46.



## 1 3.2.13 `omp_get_thread_limit`

### 2 **Summary**

3 The `omp_get_thread_limit` routine returns the maximum number of OpenMP  
4 threads available to the program.

### 5 **Format**

▼ C/C++ ▼

```
int omp_get_thread_limit(void)
```

▲ C/C++ ▲

▼ Fortran ▼

```
integer function omp_get_thread_limit()
```

▲ Fortran ▲

### 6 **Binding**

7 The binding thread set for an `omp_get_thread_limit` region is all threads. The  
8 effect of executing this routine is not related to any specific region corresponding to any  
9 construct or API routine.

### 10 **Effect**

11 The `omp_get_thread_limit` routine returns the maximum number of OpenMP  
12 threads available to the program as stored in the ICV *thread-limit-var*.

1 **Cross References**

- 2 • *thread-limit-var* ICV, see Section 2.3 on page 28.  
3 • **OMP\_THREAD\_LIMIT** environment variable, see Section 4.9 on page 157.

4 **3.2.14 omp\_set\_max\_active\_levels**

5 **Summary**

6 The **omp\_set\_max\_active\_levels** routine limits the number of nested active  
7 parallel regions, by setting the *max-active-levels-var* ICV.

8 **Format**

▼ C/C++ ▼

```
void omp_set_max_active_levels (int max_levels)
```

▲ C/C++ ▲

▼ Fortran ▼

```
subroutine omp_set_max_active_levels (max_levels)  
integer max_levels
```

▲ Fortran ▲

9 **Constraints on Arguments**

10 The value of the argument passed to this routine must evaluate to a non-negative integer,  
11 or else the behavior of this routine is implementation defined.

1

## Binding

2

3

4

5

When called from the sequential part of the program, the binding thread set for an `omp_set_max_active_levels` region is the encountering thread. When called from within any explicit parallel region, the binding thread set (and binding region, if required) for the `omp_set_max_active_levels` region is implementation defined.

6

## Effect

7

8

The effect of this routine is to set the value of the *max-active-levels-var* ICV to the value specified in the argument.

9

10

11

If the number of parallel levels requested exceeds the number of levels of parallelism supported by the implementation, the value of the *max-active-levels-var* ICV will be set to the number of parallel levels support by the implementation.

12

13

14

This routine has the described effect only when called from the sequential part of the program. When called from within an explicit `parallel` region, the effect of this routine is implementation defined.

15

## Cross References

16

17

18

- *thread-limit-var* ICV, see Section 2.3 on page 28.
- `omp_get_max_active_levels` routine, see Section 3.2.15 on page 131.
- `OMP_MAX_ACTIVE_LEVELS` environment variable, see Section 4.8 on page 157.

19 **3.2.15**

## `omp_get_max_active_levels`

20

### Summary

21

22

The `omp_get_max_active_levels` routine returns the value of the *max-active-levels-var* ICV, which determines the maximum number of nested active parallel regions.

1

## Format

▼ C/C++ ▼

```
int omp_get_max_active_levels(void)
```

▲ C/C++ ▲

▼ Fortran ▼

```
integer function omp_get_max_active_levels()
```

▲ Fortran ▲

2

## Binding

3

When called from the sequential part of the program, the binding thread set for an `omp_get_max_active_levels` region is the encountering thread. When called from within any explicit parallel region, the binding thread set (and binding region, if required) for the `omp_get_max_active_levels` region is implementation defined.

4

5

6

7

## Effect

8

The `omp_get_max_active_levels` routine returns the value of the *max-active-levels-var ICV*, which determines the maximum number of nested active parallel regions.

9

10

## Cross References

11

- *thread-limit-var ICV*, see Section 2.3 on page 28.

12

- `omp_set_max_active_levels` routine, see Section 3.2.14 on page 130.

13

- `OMP_MAX_ACTIVE_LEVELS` environment variable, see Section 4.8 on page 157.

## 1 3.2.16 `omp_get_level`

### 2 Summary

3 The `omp_get_level` routine returns the number of nested **parallel** regions  
4 enclosing the task that contains the call.

### 5 Format

C/C++

```
int omp_get_level(void)
```

C/C++

Fortran

```
integer function omp_get_level()
```

Fortran

### 6 Binding

7 The binding task set for an `omp_get_level` region is the generating task. The  
8 binding region for an `omp_get_level` region is the innermost enclosing parallel  
9 region.

### 10 Effect

11 The `omp_get_level` routine returns the number of nested **parallel** regions  
12 (whether active or inactive) enclosing the task that contains the call, not including the  
13 implicit parallel region. The routine always returns a non-negative integer, and returns 0  
14 if it is called from the sequential part of the program.

1 **Cross References**

- 2 • `omp_get_active_level` routine, see Section 3.2.19 on page 137.  
3 • `OMP_MAX_ACTIVE_LEVELS` environment variable, see Section 4.8 on page 157.

4 **3.2.17 `omp_get_ancestor_thread_num`**

5 **Summary**

6 The `omp_get_ancestor_thread_num` routine returns, for a given nested level of  
7 the current thread, the thread number of the ancestor or the current thread.

8 **Format**

▼ C/C++ ▼

```
int omp_get_ancestor_thread_num(int level)
```

▲ C/C++ ▲

▼ Fortran ▼

```
integer function omp_get_ancestor_thread_num(level)  
integer level
```

▲ Fortran ▲

9 **Binding**

10 The binding thread set for an `omp_get_ancestor_thread_num` region is the  
11 encountering thread. The binding region for an `omp_get_ancestor_thread_num`  
12 region is the innermost enclosing `parallel` region.

1  
2  
3  
4  
5

## Effect

The `omp_get_ancestor_thread_num` routine returns the thread number of the ancestor at a given nest level of the current thread or the thread number of the current thread. If the requested nest level is outside the range of 0 and the nest level of the current thread, as returned by the `omp_get_level` routine, the routine returns -1.

6  
7  
8

▼  
**Note** – When the `omp_get_ancestor_thread_num` routine is called with a value of `level=0`, the routine always returns 0. If `level=omp_get_level()`, the routine has the same effect as the `omp_get_thread_num` routine.  
▲

9  
10  
11  
12

## Cross References

- `omp_get_level` routine, see Section 3.2.16 on page 133.
- `omp_get_thread_num` routine, see Section 3.2.4 on page 117.
- `omp_get_team_size` routine, see Section 3.2.18 on page 135.

## 13 **3.2.18** `omp_get_team_size`

14  
15  
16

## Summary

The `omp_get_team_size` routine returns, for a given nested level of the current thread, the size of the thread team to which the ancestor or the current thread belongs.

1

## Format

▼ C/C++ ▼

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

▲ C/C++ ▲

▼ Fortran ▼

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

▲ Fortran ▲

2

## Binding

3

The binding thread set for an `omp_get_team_size` region is the encountering thread. The binding region for an `omp_get_team_size` region is the innermost enclosing `parallel` region.

4

5

6

## Effect

7

The `omp_get_team_size` routine returns the size of the thread team to which the ancestor or the current thread belongs. If the requested nested level is outside the range of 0 and the nested level of the current thread, as returned by the `omp_get_level` routine, the routine returns -1. Inactive parallel regions are regarded like active parallel regions executed with one thread.

8

9

10

11

▼

**Note** – When the `omp_get_team_size` routine is called with a value of `level=0`, the routine always returns 1. If `level=omp_get_level()`, the routine has the same effect as the `omp_get_num_threads` routine.

▲

12

13

14



## 1 **Cross References**

- 2 • `omp_get_num_threads` routine, see Section 3.2.2 on page 115.
- 3 • `omp_get_level` routine, see Section 3.2.16 on page 133.
- 4 • `omp_get_ancestor_thread_num` routine, see Section 3.2.17 on page 134.

## 5 **3.2.19 `omp_get_active_level`**

### 6 **Summary**

7 The `omp_get_active_level` routine returns the number of nested, active  
8 `parallel` regions enclosing the task that contains the call.

### 9 **Format**

▼ `C/C++` ▼

```
int omp_get_active_level(void)
```

▲ `C/C++` ▲

▼ `Fortran` ▼

```
integer function omp_get_active_level()
```

▲ `Fortran` ▲

### 10 **Binding**

11 The binding task set for the an `omp_get_active_level` region is the generating  
12 task. The binding region for an `omp_get_active_level` region is the innermost  
13 enclosing `parallel` region.

1           **Effect**

2           The `omp_get_active_level` routine returns the number of nested, active parallel  
3           regions enclosing the task that contains the call. The routine always returns a non-  
4           negative integer, and always returns 0 if it is called from the sequential part of the  
5           program.

6           **Cross References**

- 7           • `omp_get_level` routine, see Section 3.2.16 on page 133.

8 **3.2.20**    **`omp_in_final`**

9           **Summary**

10          The `omp_in_final()` routine returns `true` if the routine is executed in a final or  
11          included task region; otherwise, it returns `false`.

12          **Format**

▼──────────────────────────────── C/C++ ─────────────────────────────────▼

```
int omp_in_final(void);
```

▲──────────────────────────────── C/C++ ─────────────────────────────────▲

▼──────────────────────────────── Fortran ─────────────────────────────────▼

```
logical function omp_in_final()
```

13          **Binding**

14          The binding task set for an `omp_in_final` region is the generating task.

1  
2  
3

## Effect

`omp_in_final` returns `true` if the enclosing task region is final or included. Otherwise, it returns `false`.

---

## 3.3 Lock Routines

5  
6  
7  
8  
9

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; programs that otherwise access OpenMP lock variables are non-conforming.

10  
11  
12  
13  
14

An OpenMP lock may be in one of the following states: *uninitialized*, *unlocked*, or *locked*. If a lock is in the unlocked state, a task may *set* the lock, which changes its state to locked. The task which sets the lock is then said to *own* the lock. A task which owns a lock may *unset* that lock, returning it to the unlocked state. A program in which a task unsets a lock that is owned by another task is non-conforming.

15  
16  
17  
18  
19  
20

Two types of locks are supported: *simple locks* and *nestable locks*. A nestable lock may be set multiple times by the same task before being unset; a simple lock may not be set if it is already owned by the task 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.

21  
22  
23

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.

24  
25  
26  
27

The OpenMP lock routines access a lock variable in such a way that they always read and update the most current value of the lock variable. It is not necessary for an OpenMP program to include explicit `flush` directives to ensure that the lock variable's value is consistent among different tasks.

28  
29



See Section A.42 on page 281 and Section A.43 on page 284, for examples of using the simple and the nestable lock routines, respectively.

30

## Binding

1 The binding thread set for all lock routine regions is all threads. As a consequence, for  
2 each OpenMP lock, the lock routine effects relate to all tasks that call the routines,  
3 without regard to which team(s) the threads executing the tasks belong.

## 4 Simple Lock Routines

5  C/C++   
6 The type `omp_lock_t` is an data type capable of representing a simple lock. For the  
7 following routines, a simple lock variable must be of `omp_lock_t` type. All simple  
lock routines require an argument that is a pointer to a variable of type `omp_lock_t`.

8  C/C++ 

9  Fortran 


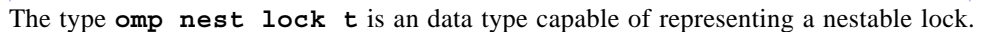
8 For the following routines, a simple lock variable must be an integer variable of  
9 `kind=omp_lock_kind`.

10  Fortran 

10 The simple lock routines are as follows:

- 11 • The `omp_init_lock` routine initializes a simple lock.
- 12 • The `omp_destroy_lock` routine uninitializes a simple lock.
- 13 • The `omp_set_lock` routine waits until a simple lock is available, and then sets it.
- 14 • The `omp_unset_lock` routine unsets a simple lock.
- 15 • The `omp_test_lock` routine tests a simple lock, and sets it if it is available.

## 16 Nestable Lock Routines:

17  C/C++   
18 The type `omp_nest_lock_t` is an data type capable of representing a nestable lock.  
19 For the following routines, a nested lock variable must be of `omp_nest_lock_t` type.  
20 All nestable lock routines require an argument that is a pointer to a variable of type  
`omp_nest_lock_t`.

21  C/C++ 

22  Fortran 

21 For the following routines, a nested lock variable must be an integer variable of  
22 `kind=omp_nest_lock_kind`.

23  Fortran 

- 1 The nestable lock routines are as follows:
- 2 • The `omp_init_nest_lock` routine initializes a nestable lock.
  - 3 • The `omp_destroy_nest_lock` routine uninitialized a nestable lock.
  - 4 • The `omp_set_nest_lock` routine waits until a nestable lock is available, and then
  - 5 sets it.
  - 6 • The `omp_unset_nest_lock` routine unsets a nestable lock.
  - 7 • The `omp_test_nest_lock` routine tests a nestable lock, and sets it if it is
  - 8 available.

### 9 3.3.1 `omp_init_lock` and `omp_init_nest_lock`

#### 10 Summary

11 These routines provide the only means of initializing an OpenMP lock.

#### 12 Format

▼ C/C++ ▼

```
void omp_init_lock(omp_lock_t *lock);  
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

▲ C/C++ ▲

▼ Fortran ▼

```
subroutine omp_init_lock(svar)  
integer (kind=omp_lock_kind) svar  
  
subroutine omp_init_nest_lock(nvar)  
integer (kind=omp_nest_lock_kind) nvar
```

▲ Fortran ▲

## Constraints on Arguments

A program that accesses a lock that is not in the uninitialized state through either routine is non-conforming.

## Effect

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

For an example of the `omp_init_lock` routine, see Section A.40 on page 279.

## 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

1           **Constraints on Arguments**

2           A program that accesses a lock that is not in the unlocked state through either routine is  
3           non-conforming.

4           **Effect**

5           The effect of these routines is to change the state of the lock to uninitialized.

6   **3.3.3   omp\_set\_lock and omp\_set\_nest\_lock**

7           **Summary**

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

10          **Format**

▼────────────────── C/C++ ───────────────────▶

```
void omp_set_lock(omp_lock_t *lock);  
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

▲────────────────── C/C++ ───────────────────▶

▼────────────────── Fortran ───────────────────▶

```
subroutine omp_set_lock(svar)  
integer (kind=omp_lock_kind) svar  
  
subroutine omp_set_nest_lock(nvar)  
integer (kind=omp_nest_lock_kind) nvar
```

▲────────────────── Fortran ───────────────────▶

## Constraints on Arguments

A program that accesses a lock that is in the uninitialized state through either routine is non-conforming. A simple lock accessed by `omp_set_lock` that is in the locked state must not be owned by the task that contains the call or deadlock will result.

## Effect

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

A simple lock is available if it is unlocked. Ownership of the lock is granted to the task executing the routine.

A nestable lock is available if it is unlocked or if it is already owned by the task executing the routine. The task executing the routine is granted, or retains, ownership of the lock, and the nesting count for the lock is incremented.

## 3.3.4 `omp_unset_lock` and `omp_unset_nest_lock`

### Summary

These routines provide the means of unsetting an OpenMP lock.

### Format

C/C++

```
void omp_unset_lock(omp_lock_t *lock);  
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

C/C++

Fortran

```
subroutine omp_unset_lock(svar)  
integer (kind=omp_lock_kind) svar  
  
subroutine omp_unset_nest_lock(nvar)  
integer (kind=omp_nest_lock_kind) nvar
```



## 1        **Constraints on Arguments**

2        A program that accesses a lock that is not in the locked state or that is not owned by the  
3        task that contains the call through either routine is non-conforming.

## 4        **Effect**

5        For a simple lock, the `omp_unset_lock` routine causes the lock to become unlocked.

6        For a nestable lock, the `omp_unset_nest_lock` routine decrements the nesting  
7        count, and causes the lock to become unlocked if the resulting nesting count is zero.

8        For either routine, if the lock becomes unlocked, and if one or more tasks regions were  
9        suspended because the lock was unavailable, the effect is that one task is chosen and  
10       given ownership of the lock.

## 11 **3.3.5        `omp_test_lock` and `omp_test_nest_lock`**

### 12        **Summary**

13        These routines attempt to set an OpenMP lock but do not suspend execution of the task  
14        executing the routine.

1

## 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

2

## Constraints on Arguments

3

A program that accesses a lock that is in the uninitialized state through either routine is non-conforming. The behavior is unspecified if a simple lock accessed by

4

**omp\_test\_lock** that is in the locked state is owned by the task that contains the call.

5

6

## Effect

7

These routines attempt to set a lock in the same manner as **omp\_set\_lock** and **omp\_set\_nest\_lock**, except that they do not suspend execution of the task executing the routine.

8

9

10

For a simple lock, the **omp\_test\_lock** routine returns *true* if the lock is successfully set; otherwise, it returns *false*.

11

12

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.

13

14

## 3.4 Timing Routines

15

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

- 1 • the `omp_get_wtime` routine.
- 2 • the `omp_get_wtick` routine.

### 3 3.4.1 `omp_get_wtime`

#### 4 **Summary**

5 The `omp_get_wtime` routine returns elapsed wall clock time in seconds.

#### 6 **Format**

C/C++

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

C/C++

Fortran

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

Fortran

#### 7 **Binding**

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

#### 10 **Effect**

11 The `omp_get_wtime` routine returns a value equal to the elapsed wall clock time in  
12 seconds since some “time in the past”. The actual “time in the past” is arbitrary, but it is  
13 guaranteed not to change during the execution of the application program. The times  
14 returned are “per-thread times”, so they are not required to be globally consistent across  
15 all the threads participating in an application.

1  
2

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

C/C++

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

C/C++

Fortran

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

Fortran

### 3.4.2 `omp_get_wtick`

#### Summary

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

1

## Format

C/C++

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

C/C++

Fortran

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

Fortran

2

## Binding

3

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

4

5

## Effect

6

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`.

7



# Environment Variables

This chapter describes the OpenMP environment variables that specify the settings of the ICVs that affect the execution of OpenMP programs (see Section 2.3 on page 28). The names of the environment variables must be upper case. The values assigned to the environment variables are case insensitive and may have leading and trailing white space. Modifications to the environment variables after the program has started, even if modified by the program itself, are ignored by the OpenMP implementation. However, the settings of some of the ICVs can be modified during the execution of the OpenMP program by the use of the appropriate directive clauses or OpenMP API routines.

The environment variables are as follows:

- **OMP\_SCHEDULE** sets the *run-sched-var* ICV for the runtime schedule type and chunk size. It can be set to any of the valid OpenMP schedule types (i.e., **static**, **dynamic**, **guided**, and **auto**).
- **OMP\_NUM\_THREADS** sets the *nthreads-var* ICV for the number of threads to use for **parallel** regions.
- **OMP\_DYNAMIC** sets the *dyn-var* ICV for the dynamic adjustment of threads to use for **parallel** regions.
- **OMP\_NESTED** sets the *nest-var* ICV to enable or to disable nested parallelism.
- **OMP\_STACKSIZE** sets the *stacksize-var* ICV that specifies the size of the stack for threads created by the OpenMP implementation.
- **OMP\_WAIT\_POLICY** sets the *wait-policy-var* ICV that controls the desired behavior of waiting threads.
- **OMP\_MAX\_ACTIVE\_LEVELS** sets the *max-active-levels-var* ICV that controls the maximum number of nested active parallel regions.
- **OMP\_THREAD\_LIMIT** sets the *thread-limit-var* ICV that controls the maximum number of threads participating in the OpenMP program.

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:

- 1 • csh:

```
setenv OMP_SCHEDULE "dynamic"
```

- 2 • ksh:

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

- 3 • DOS:

```
set OMP_SCHEDULE=dynamic
```

---

## 4.1 OMP\_SCHEDULE

5 The **OMP\_SCHEDULE** environment variable controls the schedule type and chunk size  
6 of all loop directives that have the schedule type **runtime**, by setting the value of the  
7 *run-sched-var* ICV.

8 The value of this environment variable takes the form:

9 *type[,chunk]*

10 where

- 11 • *type* is one of **static**, **dynamic**, **guided**, or **auto**  
12 • *chunk* is an optional positive integer that specifies the chunk size

13 If chunk is present, there may be white space on either side of the “,”. See Section 2.5.1  
14 on page 38 for a detailed description of the schedule types.

15 The behavior of the program is implementation defined if the value of **OMP\_SCHEDULE**  
16 does not conform to the above format.

17 Implementation specific schedules cannot be specified in **OMP\_SCHEDULE**. They can  
18 only be specified by calling **omp\_set\_schedule**, described in Section 3.2.11 on page  
19 125.

20 Example:

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



## Cross References

- *run-sched-var* ICV, see Section 2.3 on page 28.
- Loop construct, see Section 2.5.1 on page 38.
- Parallel loop construct, see Section 2.6.1 on page 54.
- `omp_set_schedule` routine, see Section 3.2.11 on page 125.
- `omp_get_schedule` routine, see Section 3.2.12 on page 127.

---

## 4.2 OMP\_NUM\_THREADS

The `OMP_NUM_THREADS` environment variable sets the number of threads to use for parallel regions by setting the initial value of the *nthreads-var* ICV. See Section 2.3 on page 28 for a comprehensive set of rules about the interaction between the `OMP_NUM_THREADS` environment variable, the `num_threads` clause, the `omp_set_num_threads` library routine and dynamic adjustment of threads, and Section 2.4.1 on page 36 for a complete algorithm that describes how the number of threads for a parallel region is determined.

The value of this environment variable must be a list of positive integer values. The values of the list set the number of threads to use for `parallel` regions at the corresponding nested level.

The behavior of the program is implementation defined if any value of the list specified in the `OMP_NUM_THREADS` environment variable leads to a number of threads which is greater than an implementation can support, or if any value is not a positive integer.

Example:

```
setenv OMP_NUM_THREADS 4,3,2
```

### Cross References:

- *nthreads-var* ICV, see Section 2.3 on page 28.
- `num_threads` clause, Section 2.4 on page 32.
- `omp_set_num_threads` routine, see Section 3.2.1 on page 114.
- `omp_get_num_threads` routine, see Section 3.2.2 on page 115.
- `omp_get_max_threads` routine, see Section 3.2.3 on page 116.
- `omp_get_team_size` routine, see Section 3.2.18 on page 135.

---

## 1 4.3 OMP\_DYNAMIC

2 The `OMP_DYNAMIC` environment variable controls dynamic adjustment of the number  
3 of threads to use for executing `parallel` regions by setting the initial value of the  
4 *dyn-var* ICV. The value of this environment variable must be `true` or `false`. If the  
5 environment variable is set to `true`, the OpenMP implementation may adjust the  
6 number of threads to use for executing `parallel` regions in order to optimize the use  
7 of system resources. If the environment variable is set to `false`, the dynamic  
8 adjustment of the number of threads is disabled. The behavior of the program is  
9 implementation defined if the value of `OMP_DYNAMIC` is neither `true` nor `false`.

10 Example:

```
setenv OMP_DYNAMIC true
```

### 11 Cross References:

- 12 • *dyn-var* ICV, see Section 2.3 on page 28.
- 13 • `omp_set_dynamic` routine, see Section 3.2.7 on page 121.
- 14 • `omp_get_dynamic` routine, see Section 3.2.8 on page 122.

---

## 15 4.4 OMP\_PROC\_BIND

16 The `OMP_PROC_BIND` environment variable sets the value of the global *bind-var* ICV.  
17 The value of this environment variable must be `true` or `false`. If the environment  
18 variable is set to `true`, the execution environment should not move OpenMP threads  
19 between processors. If the environment variable is set to `false`, the execution  
20 environment may move OpenMP threads between processors. The behavior of the  
21 program is implementation defined if the value of `OMP_PROC_BIND` is neither `true`  
22 nor `false`.

23 Example:

```
setenv OMP_PROC_BIND true
```

### 24 Cross References:

- 25 • *bind-var*, see Section 2.3 on page 28.

1

\*

---

## 2 4.5 OMP\_NESTED

3 The `OMP_NESTED` environment variable controls nested parallelism by setting the  
4 initial value of the *nest-var* ICV. The value of this environment variable must be `true`  
5 or `false`. If the environment variable is set to `true`, nested parallelism is enabled; if  
6 set to `false`, nested parallelism is disabled. The behavior of the program is  
7 implementation defined if the value of `OMP_NESTED` is neither `true` nor `false`.

8 Example:

```
setenv OMP_NESTED false
```

### 9 Cross References

- 10 • *nest-var* ICV, see Section 2.3 on page 28.
- 11 • `omp_set_nested` routine, see Section 3.2.9 on page 123.
- 12 • `omp_get_nested` routine, see Section 3.2.18 on page 135.

---

## 13 4.6 OMP\_STACKSIZE

14 The `OMP_STACKSIZE` environment variable controls the size of the stack for threads  
15 created by the OpenMP implementation, by setting the value of the *stacksize-var* ICV.  
16 The environment variable does not control the size of the stack for the initial thread.

17 The value of this environment variable takes the form:

18 *size* | *size***B** | *size***K** | *size***M** | *size***G**

19 where:

- 20 • *size* is a positive integer that specifies the size of the stack for threads that are created  
21 by the OpenMP implementation.
- 22 • **B**, **K**, **M**, and **G** are letters that specify whether the given size is in Bytes, Kilobytes,  
23 Megabytes, or Gigabytes, respectively. If one of these letters is present, there may be  
24 white space between *size* and the letter.

1 If only *size* is specified and none of **B**, **K**, **M**, or **G** is specified, then *size* is assumed to be  
2 in Kilobytes.

3 The behavior of the program is implementation defined if **OMP\_STACKSIZE** does not  
4 conform to the above format, or if the implementation cannot provide a stack with the  
5 requested size.

6 Examples:

```
setenv OMP_STACKSIZE 2000500B
setenv OMP_STACKSIZE "3000 k "
setenv OMP_STACKSIZE 10M
setenv OMP_STACKSIZE " 10 M "
setenv OMP_STACKSIZE "20 m "
setenv OMP_STACKSIZE " 1G"
setenv OMP_STACKSIZE 20000
```

## 7 **Cross References**

- 8 • *stacksize-var* ICV, see Section 2.3 on page 28.

---

## 9 **4.7 OMP\_WAIT\_POLICY**

10 The **OMP\_WAIT\_POLICY** environment variable provides a hint to an OpenMP  
11 implementation about the desired behavior of waiting threads by setting the *wait-policy-*  
12 *var* ICV. A compliant OpenMP implementation may or may not abide by the setting of  
13 the environment variable.

14 The value of this environment variable takes the form:

15 **ACTIVE** | **PASSIVE**

16 The **ACTIVE** value specifies that waiting threads should mostly be active, i.e., consume  
17 processor cycles, while waiting. An OpenMP implementation may, for example, make  
18 waiting threads spin.

19 The **PASSIVE** value specifies that waiting threads should mostly be passive, i.e., not  
20 consume processor cycles, while waiting. An OpenMP implementation, may for  
21 example, make waiting threads yield the processor to other threads or go to sleep.

22 The details of the **ACTIVE** and **PASSIVE** behaviors are implementation defined.

1 Examples:

```
setenv OMP_WAIT_POLICY ACTIVE
setenv OMP_WAIT_POLICY active
setenv OMP_WAIT_POLICY PASSIVE
setenv OMP_WAIT_POLICY passive
```

## 2 Cross References

- 3
- *wait-policy-var* ICV, see Section 2.3 on page 24.

---

## 4 4.8 OMP\_MAX\_ACTIVE\_LEVELS

5 The `OMP_MAX_ACTIVE_LEVELS` environment variable controls the maximum number  
6 of nested active parallel regions by setting the initial value of the *max-active-levels-var*  
7 ICV.

8 The value of this environment variable must be a non-negative integer. The behavior of  
9 the program is implementation defined if the requested value of  
10 `OMP_MAX_ACTIVE_LEVELS` is greater than the maximum number of nested active  
11 parallel levels an implementation can support, or if the value is not a non-negative  
12 integer.

## 13 Cross References

- 14
- *max-active-levels-var* ICV, see Section 2.3 on page 28.
  - `omp_set_max_active_levels` routine, see Section 3.2.14 on page 130.
  - `omp_get_max_active_levels` routine, see Section 3.2.15 on page 131.

---

## 17 4.9 OMP\_THREAD\_LIMIT

18 The `OMP_THREAD_LIMIT` environment variable sets the number of OpenMP threads  
19 to use for the whole OpenMP program by setting the *thread-limit-var* ICV.

20 The value of this environment variable must be a positive integer. The behavior of the  
21 program is implementation defined if the requested value of `OMP_THREAD_LIMIT` is  
22 greater than the number of threads an implementation can support, or if the value is not  
23 a positive integer.

- 1           **Cross References**
- 2           • *thread-limit-var* ICV, see Section 2.3 on page 28.
- 3           • `omp_get_thread_limit` routine

1

# A

2

## Examples

3

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

4

5

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

7

8

9

The following example demonstrates how to parallelize a simple loop using the parallel loop construct (Section 2.6.1 on page 54). The loop iteration variable is private by default, so it is not necessary to specify it explicitly in a **private** clause.

10

11

12

13

14

15

16

C/C++

*Example A.1.1c*

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

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

C/C++

*Example A.1.1f*

```
1          SUBROUTINE SIMPLE(N, A, B)
2              INTEGER I, N
3              REAL B(N), A(N)
4
5              !$OMP PARALLEL DO !I is private by default
6                  DO I=2,N
7                      B(I) = (A(I) + A(I-1)) / 2.0
8                  ENDDO
9              !$OMP END PARALLEL DO
10
11             END SUBROUTINE SIMPLE
```

## 10 A.2 The OpenMP Memory Model

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



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

C/C++

### Example A.2.1c

```
4 #include <stdio.h>
5 #include <omp.h>
6
7 int main(){
8     int x;
9
10    x = 2;
11    #pragma omp parallel num_threads(2) shared(x)
12    {
13        if (omp_get_thread_num() == 0) {
14            x = 5;
15        } else {
16            /* Print 1: the following read of x has a race */
17            printf("1: Thread# %d: x = %d\n", omp_get_thread_num(), x );
18        }
19
20        #pragma omp barrier
21
22        if (omp_get_thread_num() == 0) {
23            /* Print 2 */
24            printf("2: Thread# %d: x = %d\n", omp_get_thread_num(), x );
25        } else {
26            /* Print 3 */
27            printf("3: Thread# %d: x = %d\n", omp_get_thread_num(), x );
28        }
29    }
30    return 0;
31 }
```

C/C++

*Example A.2.1f*

```

1      PROGRAM MEMMODEL
2          INCLUDE "omp_lib.h"      ! or USE OMP_LIB
3          INTEGER X
4
5          X = 2
6      !$OMP PARALLEL NUM_THREADS(2) SHARED(X)
7
8          IF (OMP_GET_THREAD_NUM() .EQ. 0) THEN
9              X = 5
10             ELSE
11                 ! PRINT 1: The following read of x has a race
12                 PRINT *, "1: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
13             ENDIF
14
15         !$OMP BARRIER
16
17         IF (OMP_GET_THREAD_NUM() .EQ. 0) THEN
18             ! PRINT 2
19             PRINT *, "2: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
20         ELSE
21             ! PRINT 3
22             PRINT *, "3: THREAD# ", OMP_GET_THREAD_NUM(), "X = ", X
23         ENDIF
24
25     !$OMP END PARALLEL
26
27     END PROGRAM MEMMODEL

```

27 The following example demonstrates why synchronization is difficult to perform  
 28 correctly through variables. The value of flag is undefined in both prints on thread 1 and  
 29 the value of data is only well-defined in the second print.

*Example A.2.2c*

```

1      #include <omp.h>
2      #include <stdio.h>
3      int main()
4      {
5          int data;
6          int flag=0;
7          #pragma omp parallel
8          {
9              if (omp_get_thread_num()==0)
10             {
11                 /* Write to the data buffer that will be
12                  read by thread */
13                 data = 42;
14                 /* Flush data to thread 1 and strictly order
15                  the write to data
16                  relative to the write to the flag */
17                 #pragma omp flush(flag, data)
18                 /* Set flag to release thread 1 */
19                 flag = 1;
20                 /* Flush flag to ensure that thread 1 sees
21                  the change */
22                 #pragma omp flush(flag)
23             }
24             else if(omp_get_thread_num()==1)
25             {
26                 /* Loop until we see the update to the flag */
27                 #pragma omp flush(flag, data)
28                 while (flag < 1)
29                 {
30                     #pragma omp flush(flag, data)
31                 }
32                 /* Values of flag and data are undefined */
33                 printf("flag=%d data=%d\n", flag, data);
34                 #pragma omp flush(flag, data)
35                 /* Values data will be 42, value of flag
36                  still undefined */
37                 printf("flag=%d data=%d\n", flag, data);
38             }
39         }
40     }

```

*Example A.2.2f*

```

1          PROGRAM EXAMPLE
2          INCLUDE "omp_lib.h" ! or USE OMP_LIB
3          INTEGER DATA
4          INTEGER FLAG

5          FLAG = 0
6          !$OMP PARALLEL
7              IF(OMP_GET_THREAD_NUM() .EQ. 0) THEN
8                  ! Write to the data buffer that will be read by thread 1
9                  DATA = 42
10                 ! Flush DATA to thread 1 and strictly order the write to DATA
11                 ! relative to the write to the FLAG
12                 !$OMP FLUSH(FLAG, DATA)
13                 ! Set FLAG to release thread 1
14                 FLAG = 1;
15                 ! Flush FLAG to ensure that thread 1 sees the change */
16                 !$OMP FLUSH(FLAG)
17             ELSE IF(OMP_GET_THREAD_NUM() .EQ. 1) THEN
18                 ! Loop until we see the update to the FLAG
19                 !$OMP FLUSH(FLAG, DATA)
20                 DO WHILE(FLAG .LT. 1)
21                     !$OMP FLUSH(FLAG, DATA)
22                 ENDDO

23                 ! Values of FLAG and DATA are undefined
24                 PRINT *, 'FLAG=', FLAG, ' DATA=', DATA
25                 !$OMP FLUSH(FLAG, DATA)

26                 !Values DATA will be 42, value of FLAG still undefined */
27                 PRINT *, 'FLAG=', FLAG, ' DATA=', DATA
28             ENDIF
29         !$OMP END PARALLEL
30     END

```

31 This example demonstrates why synchronization is difficult to perform correctly through  
32 variables. Because the write(1)-flush(1)-flush(2)-read(2) sequence cannot be guaranteed  
33 in the example, the statements on thread 1 and thread 2 may execute in either order.

*Example A.2.3c*

```

1      #include <omp.h>
2      #include <stdio.h>
3      int main()
4      {
5          int flag=0;

6          #pragma omp parallel
7          {
8              if(omp_get_thread_num()==0)
9              {
10                 /* Set flag to release thread 1 */
11                 #pragma omp atomic update
12                 flag++;
13                 /* Flush of flag is implied by the atomic directive */
14             }
15             else if(omp_get_thread_num()==1)
16             {
17                 /* Loop until we see that flag reaches 1*/
18                 #pragma omp flush(flag)
19                 while(flag < 1)
20                 {
21                     #pragma omp flush(flag)
22                 }
23                 printf("Thread 1 awoken\n");

24                 /* Set flag to release thread 2 */
25                 #pragma omp atomic update
26                 flag++;
27                 /* Flush of flag is implied by the atomic directive */
28             }
29             else if(omp_get_thread_num()==2)
30             {
31                 /* Loop until we see that flag reaches 2 */
32                 #pragma omp flush(flag)
33                 while(flag < 2)
34                 {
35                     #pragma omp flush(flag)
36                 }
37                 printf("Thread 2 awoken\n");
38             }
39         }
40     }

```

*Example A.2.3f*

```

1          PROGRAM EXAMPLE
2          INCLUDE "omp_lib.h" ! or USE OMP_LIB
3          INTEGER FLAG

4          FLAG = 0
5          !$OMP PARALLEL
6              IF(OMP_GET_THREAD_NUM() .EQ. 0) THEN
7                  ! Set flag to release thread 1
8                  !$OMP ATOMIC UPDATE
9                      FLAG = FLAG + 1
10                 !Flush of FLAG is implied by the atomic directive
11             ELSE IF(OMP_GET_THREAD_NUM() .EQ. 1) THEN
12                 ! Loop until we see that FLAG reaches 1
13                 !$OMP FLUSH(FLAG, DATA)
14                 DO WHILE(FLAG .LT. 1)
15                     !$OMP FLUSH(FLAG, DATA)
16                 ENDDO

17                 PRINT *, 'Thread 1 awoken'

18                 ! Set FLAG to release thread 2
19                 !$OMP ATOMIC UPDATE
20                     FLAG = FLAG + 1
21                 !Flush of FLAG is implied by the atomic directive
22             ELSE IF(OMP_GET_THREAD_NUM() .EQ. 2) THEN
23                 ! Loop until we see that FLAG reaches 2
24                 !$OMP FLUSH(FLAG, DATA)
25                 DO WHILE(FLAG .LT. 2)
26                     !$OMP FLUSH(FLAG, DATA)
27                 ENDDO

28                 PRINT *, 'Thread 2 awoken'
29             ENDIF
30         !$OMP END PARALLEL
31     END

```

---

## 1 A.3 Conditional Compilation

2 The following example illustrates the use of conditional compilation using the OpenMP  
3 macro `_OPENMP` (Section 2.2 on page 26). With OpenMP compilation, the `_OPENMP`  
4 macro becomes defined.

Example A.3.1c

```
5 #include <stdio.h>
6
7 int main()
8 {
9     # ifdef _OPENMP
10        printf("Compiled by an OpenMP-compliant implementation.\n");
11        # endif
12
13        return 0;
14    }
```

C/C++

13 The following example illustrates the use of the conditional compilation sentinel (see  
14 Section 2.2 on page 26). With OpenMP compilation, the conditional compilation  
15 sentinel `!$` is recognized and treated as two spaces. In fixed form source, statements  
16 guarded by the sentinel must start after column 6.

Example A.3.1f

```
17          PROGRAM EXAMPLE
18
19          C234567890
20          !$   PRINT *, "Compiled by an OpenMP-compliant implementation."
```

Fortran

---

## 1 A.4 Internal Control Variables (ICVs)

2 According to Section 2.3 on page 28, an OpenMP implementation must act as if there  
3 are ICVs that control the behavior of the program. This example illustrates two ICVs,  
4 *nthreads-var* and *max-active-levels-var*. The *nthreads-var* ICV controls the number of  
5 threads requested for encountered parallel regions; there is one copy of this ICV per  
6 task. The *max-active-levels-var* ICV controls the maximum number of nested active  
7 parallel regions; there is one copy of this ICV for the whole program.

8 In the following example, the *nest-var*, *max-active-levels-var*, *dyn-var*, and *nthreads-var*  
9 ICVs are modified through calls to the runtime library routines `omp_set_nested`,  
10 `omp_set_max_active_levels`, `omp_set_dynamic`, and  
11 `omp_set_num_threads` respectively. This example assumes that nested parallelism  
12 is supported.

13 These ICVs affect the operation of parallel regions. Each implicit task generated by a  
14 parallel region has its own copy of the *nest-var*, *dyn-var*, and *nthreads-var*. In the  
15 example, the new value of *nthreads-var* applies only to the implicit tasks that execute  
16 the call to `omp_set_num_threads`. There is one copy of the *max-active-levels-var*  
17 for the whole program; its value is the same for all tasks.

18 The outer parallel region creates a team of two threads; each of the threads will execute  
19 one of the two implicit tasks generated by the outer parallel region.

20 Each implicit task generated by the outer parallel region calls  
21 `omp_set_num_threads(3)`, assigning the value 3 to its respective copy of  
22 *nthreads-var*. Then each implicit task encounters an inner parallel region that creates a  
23 team of three threads; each of the threads will execute one of the three implicit tasks  
24 generated by that inner parallel region.

25 Since the outer parallel region is executed by 2 threads, and the inner by 3, there will be  
26 a total of 6 implicit tasks generated by the two inner parallel regions.

27 Each implicit task generated by an inner parallel region will execute the call to  
28 `omp_set_num_threads(4)`, assigning the value 4 to its respective copy of  
29 *nthreads-var*.

30 The print statement in the outer parallel region is executed by only one of the threads in  
31 the team. So it will be executed only once.

32 The print statement in an inner parallel region is also executed by only one of the  
33 threads in the team. Since we have a total of two inner parallel regions, the print  
34 statement will be executed twice (once per inner parallel region).



*Example A.4.1c*

```

1      #include <stdio.h>
2      #include <omp.h>

3      int main (void)
4      {
5          omp_set_nested(1);
6          omp_set_max_active_levels(8);
7          omp_set_dynamic(0);
8          omp_set_num_threads(2);
9          #pragma omp parallel
10         {
11             omp_set_num_threads(3);

12             #pragma omp parallel
13             {
14                 omp_set_num_threads(4);
15                 #pragma omp single
16                 {
17                     /*
18                      * The following should print:
19                      * Inner: max_act_lev=8, num_thds=3, max_thds=4
20                      * Inner: max_act_lev=8, num_thds=3, max_thds=4
21                      */
22                     printf ("Inner: max_act_lev=%d, num_thds=%d, max_thds=%d\n",
23                            omp_get_max_active_levels(), omp_get_num_threads(),
24                            omp_get_max_threads());
25                 }
26             }

27             #pragma omp barrier
28             #pragma omp single
29             {
30                 /*
31                  * The following should print:
32                  * Outer: max_act_lev=8, num_thds=2, max_thds=3
33                  */
34                 printf ("Outer: max_act_lev=%d, num_thds=%d, max_thds=%d\n",
35                        omp_get_max_active_levels(), omp_get_num_threads(),
36                        omp_get_max_threads());
37             }
38         }
39     }

```

*Example A.4.1f*

```

1          program icv
2          use omp_lib

3          call omp_set_nested(.true.)
4          call omp_set_max_active_levels(8)
5          call omp_set_dynamic(.false.)
6          call omp_set_num_threads(2)

7          !$omp parallel
8              call omp_set_num_threads(3)

9          !$omp parallel
10             call omp_set_num_threads(4)
11         !$omp single
12         !     The following should print:
13         !     Inner: max_act_lev= 8 , num_thds= 3 , max_thds= 4
14         !     Inner: max_act_lev= 8 , num_thds= 3 , max_thds= 4
15         print *, "Inner: max_act_lev=", omp_get_max_active_levels(),
16             &      " , num_thds=", omp_get_num_threads(),
17             &      " , max_thds=", omp_get_max_threads()
18         !$omp end single
19         !$omp end parallel

20         !$omp barrier
21         !$omp single
22         !     The following should print:
23         !     Outer: max_act_lev= 8 , num_thds= 2 , max_thds= 3
24         print *, "Outer: max_act_lev=", omp_get_max_active_levels(),
25             &      " , num_thds=", omp_get_num_threads(),
26             &      " , max_thds=", omp_get_max_threads()
27         !$omp end single
28         !$omp end parallel
29         end

```

## 30 A.5 The parallel Construct

31 The **parallel** construct (Section 2.4 on page 32) can be used in coarse-grain parallel  
32 programs. In the following example, each thread in the **parallel** region decides what  
33 part of the global array *x* to work on, based on the thread number:

*Example A.5.1c*

```
1      #include <omp.h>
2
3      void subdomain(float *x, int istart, int ipoints)
4      {
5          int i;
6
7          for (i = 0; i < ipoints; i++)
8              x[istart+i] = 123.456;
9      }
10
11     void sub(float *x, int npoints)
12     {
13         int iam, nt, ipoints, istart;
14
15         #pragma omp parallel default(shared) private(iam,nt,ipoints,istart)
16         {
17             iam = omp_get_thread_num();
18             nt = omp_get_num_threads();
19             ipoints = npoints / nt; /* size of partition */
20             istart = iam * ipoints; /* starting array index */
21             if (iam == nt-1) /* last thread may do more */
22                 ipoints = npoints - istart;
23             subdomain(x, istart, ipoints);
24         }
25     }
26
27     int main()
28     {
29         float array[10000];
30
31         sub(array, 10000);
32
33         return 0;
34     }
```

*Example A.5.1f*

```

1          SUBROUTINE SUBDOMAIN(X, ISTART, IPOINITS)
2              INTEGER ISTART, IPOINITS
3              REAL X(*)

4              INTEGER I
5
6              DO 100 I=1,IPOINITS
7                  X(ISTART+I) = 123.456
8          100      CONTINUE

9          END SUBROUTINE SUBDOMAIN

10         SUBROUTINE SUB(X, NPOINITS)
11             INCLUDE "omp_lib.h"      ! or USE OMP_LIB
12
13             REAL X(*)
14             INTEGER NPOINITS
15             INTEGER IAM, NT, IPOINITS, ISTART

16         !$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,NPOINITS)
17
18             IAM = OMP_GET_THREAD_NUM()
19             NT = OMP_GET_NUM_THREADS()
20             IPOINITS = NPOINITS/NT
21             ISTART = IAM * IPOINITS
22             IF (IAM .EQ. NT-1) THEN
23                 IPOINITS = NPOINITS - ISTART
24             ENDIF
25             CALL SUBDOMAIN(X, ISTART, IPOINITS)

26         !$OMP END PARALLEL
27         END SUBROUTINE SUB

28         PROGRAM PAREXAMPLE
29             REAL ARRAY(10000)
30             CALL SUB(ARRAY, 10000)
31         END PROGRAM PAREXAMPLE

```

---

## 1 A.6 Interaction Between the `num_threads` Clause and `omp_set_dynamic`

2  
3 The following example demonstrates the `num_threads` clause (Section 2.4 on page  
4 32) and the effect of the `omp_set_dynamic` routine (Section 3.2.7 on page 121)  
5 on it.

6 The call to the `omp_set_dynamic` routine with argument 0 in C/C++, or `.FALSE.`  
7 in Fortran, disables the dynamic adjustment of the number of threads in OpenMP  
8 implementations that support it. In this case, 10 threads are provided. Note that in case  
9 of an error the OpenMP implementation is free to abort the program or to supply any  
10 number of threads available.

C/C++

*Example A.6.1c*

```
11 #include <omp.h>
12 int main()
13 {
14     omp_set_dynamic(0);
15     #pragma omp parallel num_threads(10)
16     {
17         /* do work here */
18     }
19     return 0;
20 }
```

C/C++

Fortran

*Example A.6.1f*

```
21 PROGRAM EXAMPLE
22     INCLUDE "omp_lib.h" ! or USE OMP_LIB
23     CALL OMP_SET_DYNAMIC(.FALSE.)
24 !$OMP PARALLEL NUM_THREADS(10)
25     ! do work here
26 !$OMP END PARALLEL
27 END PROGRAM EXAMPLE
```

Fortran

1 The call to the `omp_set_dynamic` routine with a non-zero argument in C/C++,  
2 or `.TRUE.` in Fortran, allows the OpenMP implementation to choose any number of  
3 threads between 1 and 10 (see also Algorithm 2.1 in Section 2.4.1 on page 36).

C/C++

*Example A.6.2c*

```
4 #include <omp.h>
5 int main()
6 {
7     omp_set_dynamic(1);
8     #pragma omp parallel num_threads(10)
9     {
10        /* do work here */
11    }
12    return 0;
13 }
```

C/C++

Fortran

*Example A.6.2f*

```
14 PROGRAM EXAMPLE
15     INCLUDE "omp_lib.h" ! or USE OMP_LIB
16     CALL OMP_SET_DYNAMIC(.TRUE.)
17 !$OMP PARALLEL NUM_THREADS(10)
18     ! do work here
19 !$OMP END PARALLEL
20 END PROGRAM EXAMPLE
```

Fortran

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

## 1 A.7 Fortran Restrictions on the `do` Construct

2 If an `end do` directive follows a *do-construct* in which several `DO` statements share a  
3 `DO` termination statement, then a `do` directive can only be specified for the first (i.e.  
4 outermost) of these `DO` statements. For more information, see Section 2.5.1 on page 38.  
5 The following example contains correct usages of loop constructs:

### Example A.7.1f

```
1          SUBROUTINE WORK(I, J)
2          INTEGER I,J
3          END SUBROUTINE WORK

4          SUBROUTINE DO_GOOD()
5          INTEGER I, J
6          REAL A(1000)

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

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

16         !$OMP DO
17             DO 300 I = 1,10
18                 DO 300 J = 1,10
19                     CALL WORK(I,J)
20             300 CONTINUE
21         !$OMP ENDDO
22         END SUBROUTINE DO_GOOD
```

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

### Example A.7.2f

```
25          SUBROUTINE WORK(I, J)
26          INTEGER I,J
27          END SUBROUTINE WORK

28          SUBROUTINE DO_WRONG
29          INTEGER I, J

30          DO 100 I = 1,10
31          !$OMP DO
32              DO 100 J = 1,10
33                  CALL WORK(I,J)
34             100 CONTINUE
35         !$OMP ENDDO
36         END SUBROUTINE DO_WRONG
```

Fortran



## 1 A.8 Fortran Private Loop Iteration Variables

2 In general loop iteration variables will be private, when used in the *do-loop* of a **do** and  
3 **parallel do** construct or in sequential loops in a **parallel** construct (see  
4 Section 2.5.1 on page 38 and Section 2.9.1 on page 83). In the following example of a  
5 sequential loop in a **parallel** construct the loop iteration variable *I* will be private.

### *Example A.8.1f*

```
6      SUBROUTINE PLOOP_1(A,N)
7      INCLUDE "omp_lib.h"      ! or USE OMP_LIB

8      REAL A(*)
9      INTEGER I, MYOFFSET, N

10     !$OMP PARALLEL PRIVATE(MYOFFSET)
11         MYOFFSET = OMP_GET_THREAD_NUM()*N
12         DO I = 1, N
13             A(MYOFFSET+I) = FLOAT(I)
14         ENDDO
15     !$OMP END PARALLEL

16     END SUBROUTINE PLOOP_1
```

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

### *Example A.8.2f*

```
3      SUBROUTINE PLOOP_2(A,B,N,I1,I2)
4      REAL A(*), B(*)
5      INTEGER I1, I2, N
6
7      !$OMP PARALLEL SHARED(A,B,I1,I2)
8      !$OMP SECTIONS
9      !$OMP SECTION
10         DO I1 = I1, N
11            IF (A(I1).NE.0.0) EXIT
12         ENDDO
13      !$OMP SECTION
14         DO I2 = I2, N
15            IF (B(I2).NE.0.0) EXIT
16         ENDDO
17      !$OMP END SECTIONS
18      !$OMP SINGLE
19         IF (I1.LE.N) PRINT *, 'ITEMS IN A UP TO ', I1, 'ARE ALL ZERO.'
20         IF (I2.LE.N) PRINT *, 'ITEMS IN B UP TO ', I2, 'ARE ALL ZERO.'
21      !$OMP END SINGLE
22      !$OMP END PARALLEL
23
24      END SUBROUTINE PLOOP_2
```

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



Fortran

---

## 25 **A.9** The `nowait` clause

26 If there are multiple independent loops within a **parallel** region, you can use the  
27 **nowait** clause (see Section 2.5.1 on page 38) to avoid the implied barrier at the end of  
28 the loop construct, as follows:

*Example A.9.1c*

```

1      #include <math.h>
2
3      void nowait_example(int n, int m, float *a, float *b, float *y, float *z)
4      {
5          int i;
6          #pragma omp parallel
7          {
8              #pragma omp for nowait
9              for (i=1; i<n; i++)
10                 b[i] = (a[i] + a[i-1]) / 2.0;
11
12             #pragma omp for nowait
13             for (i=0; i<m; i++)
14                 y[i] = sqrt(z[i]);
15         }
16     }

```

*Example A.9.1f*

```

16      SUBROUTINE NOWAIT_EXAMPLE(N, M, A, B, Y, Z)
17
18      INTEGER N, M
19      REAL A(*), B(*), Y(*), Z(*)
20
21      INTEGER I
22
23      !$OMP PARALLEL
24
25      !$OMP DO
26      DO I=2,N
27          B(I) = (A(I) + A(I-1)) / 2.0
28      ENDDO
29      !$OMP END DO NOWAIT
30
31      !$OMP DO
32      DO I=1,M
33          Y(I) = SQRT(Z(I))
34      ENDDO
35      !$OMP END DO NOWAIT
36
37      !$OMP END PARALLEL
38
39      END SUBROUTINE NOWAIT_EXAMPLE

```

1 In the following example, static scheduling distributes the same logical iteration  
2 numbers to the threads that execute the three loop regions. This allows the `nowait`  
3 clause to be used, even though there is a data dependence between the loops. The  
4 dependence is satisfied as long the same thread executes the same logical iteration  
5 numbers in each loop.

6 Note that the iteration count of the loops must be the same. The example satisfies this  
7 requirement, since the iteration space of the first two loops is from 0 to `n-1` (from 1 to  
8 `N` in the Fortran version), while the iteration space of the last loop is from 1 to `n` (2 to  
9 `N+1` in the Fortran version).

C/C++

*Example A.9.2c*

```
10 #include <math.h>
11 void nowait_example2(int n, float *a, float *b, float *c, float *y, float *z)
12 {
13     int i;
14     #pragma omp parallel
15     {
16         #pragma omp for schedule(static) nowait
17         for (i=0; i<n; i++)
18             c[i] = (a[i] + b[i]) / 2.0f;
19         #pragma omp for schedule(static) nowait
20         for (i=0; i<n; i++)
21             z[i] = sqrtf(c[i]);
22         #pragma omp for schedule(static) nowait
23         for (i=1; i<=n; i++)
24             y[i] = z[i-1] + a[i];
25     }
26 }
```

C/C++

Fortran

*Example A.9.2f*

```
27
28     SUBROUTINE NOWAIT_EXAMPLE2(N, A, B, C, Y, Z)
29     INTEGER N
30     REAL A(*), B(*), C(*), Y(*), Z(*)
31     INTEGER I
32     !$OMP PARALLEL
33     !$OMP DO SCHEDULE(STATIC)
34     DO I=1,N
35         C(I) = (A(I) + B(I)) / 2.0
36     ENDDO
37     !$OMP END DO NOWAIT
38     !$OMP DO SCHEDULE(STATIC)
39     DO I=1,N
40         Z(I) = SQRT(C(I))
41     ENDDO
42     !$OMP END DO NOWAIT
```

```

1      !$OMP DO SCHEDULE(STATIC)
2          DO I=2,N+1
3              Y(I) = Z(I-1) + A(I)
4          ENDDO
5      !$OMP END DO NOWAIT
6      !$OMP END PARALLEL
7      END SUBROUTINE NOWAIT_EXAMPLE2

```

Fortran

## 8 A.10 The collapse clause

9 For the following three examples, see Section 2.5.1 on page 38 for a description of the  
10 **collapse** clause, Section 2.8.7 on page 81 for a description of the **ordered**  
11 construct, and Section 2.9.3.5 on page 99 for a description of the **lastprivate**  
12 clause.

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

C/C++

*Example A.10.1c*

```

18 #include <omp.h>
19 int a, kl, ku, ks, jl, ju, js, il, iu, is;
20 void sub(int ku, int ju, int iu, float *a)
21 {
22     int i, j, k;
23     #pragma omp for collapse(2) private(i, k, j)
24     for (k=kl; k<=ku; k+=ks)
25         for (j=jl; j<=ju; j+=js)
26             for (i=il; i<=iu; i+=is)
27                 bar(a,i,j,k);
28 }

```

C/C++

Fortran

*Example A.10.1f*

```

29     subroutine sub()
30     !$omp do collapse(2) private(i,j,k)

```

```

1         do k = kl, ku, ks
2             do j = jl, ju, js
3                 do i = il, iu, is
4                     call bar(a,i,j,k)
5                 enddo
6             enddo
7         enddo
8 !$omp end do
9         end subroutine

```

## Fortran

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

13 The sequential execution of the iterations in the **k** and **j** loops determines the order of  
14 the iterations in the collapsed iteration space. This implies that in the sequentially last  
15 iteration of the collapsed iteration space, **k** will have the value 2 and **j** will have the  
16 value 3. Since **klast** and **jlast** are **lastprivate**, their values are assigned by the  
17 sequentially last iteration of the collapsed **k** and **j** loop. This example prints: 2 3.

## C/C++

### *Example A.10.2c*

```

18 #include <omp.h>
19 #include <stdio.h>
20 void test()
21 {
22     int j, k, jlast, klast;
23     #pragma omp parallel
24     {
25         #pragma omp for collapse(2) lastprivate(jlast, klast)
26         for (k=1; k<=2; k++)
27             for (j=1; j<=3; j++)
28                 {
29                     jlast=j;
30                     klast=k;
31                 }
32     #pragma omp single
33     printf("%d %d\n", klast, jlast);
34 }
35 }

```

*Example A.10.2f*

```

1      program test
2      !$omp parallel
3      !$omp do private(j,k) collapse(2) lastprivate(jlast, klast)
4          do k = 1,2
5              do j = 1,3
6                  jlast=j
7                  klast=k
8              enddo
9          enddo
10     !$omp end do
11     !$omp single
12         print *, klast, jlast
13     !$omp end single
14     !$omp end parallel
15     end program test

```

16 The next example illustrates the interaction of the **collapse** and **ordered** clauses.

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

22 According to section Section 2.8.7 on page 81, a thread must not execute more than one  
23 ordered region that binds to the same loop region. So the **collapse** clause is required  
24 for the example to be conforming. With the **collapse** clause, the iterations of the **k**  
25 and **j** loops are collapsed into one loop, and therefore only one ordered region will bind  
26 to the collapsed **k** and **j** loop. Without the **collapse** clause, there would be two  
27 ordered regions that bind to each iteration of the **k** loop (one arising from the first  
28 iteration of the **j** loop, and the other arising from the second iteration of the **j** loop).

29 The code prints

```

30 0 1 1
31 0 1 2
32 0 2 1
33 1 2 2
34 1 3 1
35 1 3 2

```

*Example A.10.3c*

```

1      #include <omp.h>
2      #include <stdio.h>
3      void sub()
4      {
5          int j, k, a;
6          #pragma omp parallel num_threads(2)
7          {
8              #pragma omp for collapse(2) ordered private(j,k) schedule(static,3)
9              for (k=1; k<=3; k++)
10                 for (j=1; j<=2; j++)
11                     {
12                         #pragma omp ordered
13                         printf("%d %d %d\n", omp_get_thread_num(), k, j);
14                         /* end ordered */
15                         work(a,j,k);
16                     }
17             }
18     }

```

*Example A.10.3f*

```

19      program test
20      include 'omp_lib.h'
21      !$omp parallel num_threads(2)
22      !$omp do collapse(2) ordered private(j,k) schedule(static,3)
23      do k = 1,3
24      do j = 1,2
25      !$omp ordered
26      print *, omp_get_thread_num(), k, j
27      !$omp end ordered
28      call work(a,j,k)
29      enddo
30      enddo
31      !$omp end do
32      !$omp end parallel
33      end program test

```



---

## 1 A.11 The parallel sections Construct

2 In the following example (for Section 2.6.2 on page 56) routines *xaxis*, *yaxis*, and *zaxis*  
3 can be executed concurrently. The first **section** directive is optional. Note that all  
4 **section** directives need to appear in the **parallel sections** construct.

C/C++

*Example A.11.1c*

```
5 void XAXIS();
6 void YAXIS();
7 void ZAXIS();

8 void sect_example()
9 {
10     #pragma omp parallel sections
11     {
12         #pragma omp section
13             XAXIS();

14         #pragma omp section
15             YAXIS();

16         #pragma omp section
17             ZAXIS();
18     }
19 }
20
```

C/C++

Fortran

*Example A.11.1f*

```
21 SUBROUTINE SECT_EXAMPLE()
22     !$OMP PARALLEL SECTIONS
23     !$OMP SECTION
24         CALL XAXIS()
25     !$OMP SECTION
26         CALL YAXIS()
27     !$OMP SECTION
28         CALL ZAXIS()
29     !$OMP END PARALLEL SECTIONS
```

1

END SUBROUTINE SECT\_EXAMPLE

Fortran

## 2 A.12 The `single` Construct

3 The following example demonstrates the `single` construct (Section 2.5.3 on page 49).  
 4 In the example, only one thread prints each of the progress messages. All other threads  
 5 will skip the `single` region and stop at the barrier at the end of the `single` construct  
 6 until all threads in the team have reached the barrier. If other threads can proceed  
 7 without waiting for the thread executing the `single` region, a `nowait` clause can be  
 8 specified, as is done in the third `single` construct in this example. The user must not  
 9 make any assumptions as to which thread will execute a `single` region.

C/C++

*Example A.12.1c*

```

10     #include <stdio.h>
11
12     void work1() {}
13     void work2() {}
14
15     void single_example()
16     {
17         #pragma omp parallel
18         {
19             #pragma omp single
20             printf("Beginning work1.\n");
21
22             work1();
23
24             #pragma omp single
25             printf("Finishing work1.\n");
26
27             #pragma omp single nowait
28             printf("Finished work1 and beginning work2.\n");
29
30             work2();
31         }
32     }

```

C/C++

*Example A.12.1f*

```

1          SUBROUTINE WORK1 ()
2          END SUBROUTINE WORK1
3
4          SUBROUTINE WORK2 ()
5          END SUBROUTINE WORK2
6
7          PROGRAM SINGLE_EXAMPLE
8  !$OMP PARALLEL
9
9  !$OMP SINGLE
10         print *, "Beginning work1."
11  !$OMP END SINGLE
12
12         CALL WORK1 ()
13
13  !$OMP SINGLE
14         print *, "Finishing work1."
15  !$OMP END SINGLE
16
16  !$OMP SINGLE
17         print *, "Finished work1 and beginning work2."
18  !$OMP END SINGLE NOWAIT
19
19         CALL WORK2 ()
20
20  !$OMP END PARALLEL
21
21         END PROGRAM SINGLE_EXAMPLE

```

## 22 A.13 Tasking Constructs

23 The following example shows how to traverse a tree-like structure using explicit tasks  
24 (see Section 2.7 on page 59). Note that the *traverse* function should be called from  
25 within a parallel region for the different specified tasks to be executed in parallel. Also,  
26 note that the tasks will be executed in no specified order because there are no  
27 synchronization directives. Thus, assuming that the traversal will be done in post order,  
28 as in the sequential code, is wrong.

C/C++

*Example A.13.1c*

```
1      struct node {
2          struct node *left;
3          struct node *right;
4      };
5      extern void process(struct node *);
6      void traverse( struct node *p ) {
7          if (p->left)
8              #pragma omp task // p is firstprivate by default
9                  traverse(p->left);
10         if (p->right)
11             #pragma omp task // p is firstprivate by default
12                 traverse(p->right);
13         process(p);
14     }
```

C/C++

Fortran

*Example A.13.1f*

```
15      RECURSIVE SUBROUTINE traverse ( P )
16          TYPE Node
17              TYPE(Node), POINTER :: left, right
18          END TYPE Node
19          TYPE(Node) :: P
20          IF (associated(P%left)) THEN
21              !$OMP TASK ! P is firstprivate by default
22                  call traverse(P%left)
23              !$OMP END TASK
24          ENDIF
25          IF (associated(P%right)) THEN
26              !$OMP TASK ! P is firstprivate by default
27                  call traverse(P%right)
28              !$OMP END TASK
29          ENDIF
30          CALL process ( P )
31      END SUBROUTINE
```

Fortran

1 In the next example, we force a postorder traversal of the tree by adding a `taskwait`  
 2 directive (See Section 2.8.4 on page 70). Now, we can safely assume that the left and  
 3 right sons have been executed before we process the current node.

C/C++

*Example A.13.2c*

```

4 struct node {
5     struct node *left;
6     struct node *right;
7 };
8 extern void process(struct node *);
9 void postorder_traverse( struct node *p ) {
10     if (p->left)
11         #pragma omp task // p is firstprivate by default
12             postorder_traverse(p->left);
13     if (p->right)
14         #pragma omp task // p is firstprivate by default
15             postorder_traverse(p->right);
16     #pragma omp taskwait
17     process(p);
18 }

```

C/C++

Fortran

*Example A.13.2f*

```

19 RECURSIVE SUBROUTINE traverse ( P )
20     TYPE Node
21         TYPE(Node), POINTER :: left, right
22     END TYPE Node
23     TYPE(Node) :: P
24     IF (associated(P%left)) THEN
25         !$OMP TASK ! P is firstprivate by default
26         call traverse(P%left)
27         !$OMP END TASK
28     ENDIF
29     IF (associated(P%right)) THEN
30         !$OMP TASK ! P is firstprivate by default
31         call traverse(P%right)
32         !$OMP END TASK
33     ENDIF
34     !$OMP TASKWAIT
35     CALL process ( P )
36 END SUBROUTINE

```

Fortran

1 The following example demonstrates how to use the task construct to process elements  
2 of a linked list in parallel. The thread executing the single region generates all of the  
3 explicit tasks, which are then executed by the threads in the current team. The pointer *p*  
4 is **firstprivate** by default on the **task** construct so it is not necessary to specify it  
5 in a **firstprivate** clause (see page 85).

C/C++

*Example A.13.3c*

```
6 typedef struct node node;
7 struct node {
8     int data;
9     node * next;
10 };
11
12 void process(node * p)
13 {
14     /* do work here */
15 }
16 void increment_list_items(node * head)
17 {
18     #pragma omp parallel
19     {
20         #pragma omp single
21         {
22             node * p = head;
23             while (p) {
24                 #pragma omp task
25                 // p is firstprivate by default
26                 process(p);
27                 p = p->next;
28             }
29         }
30     }
31 }
```

C/C++

*Example A.13.3f*

```

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

```

1 The `fib()` function should be called from within a parallel region for the different  
2 specified tasks to be executed in parallel. Also, only one thread of the parallel region  
3 should call `fib()` unless multiple concurrent Fibonacci computations are desired.

C/C++

*Example A.13.4c*

```
4     int fib(int n) {  
5         int i, j;  
6         if (n<2)  
7             return n;  
8         else {  
9             #pragma omp task shared(i)  
10            i=fib(n-1);  
11            #pragma omp task shared(j)  
12            j=fib(n-2);  
13            #pragma omp taskwait  
14            return i+j;  
15        }  
16    }
```

C/C++

Fortran

*Example A.13.4f*

```
17     RECURSIVE INTEGER FUNCTION fib(n) RESULT(res)  
18     INTEGER n, i, j  
19     IF ( n .LT. 2) THEN  
20         res = n  
21     ELSE  
22     !$OMP TASK SHARED(i)  
23         i = fib( n-1 )  
24     !$OMP END TASK  
25     !$OMP TASK SHARED(j)  
26         j = fib( n-2 )  
27     !$OMP END TASK  
28     !$OMP TASKWAIT  
29         res = i+j  
30     END IF  
31     END FUNCTION
```

Fortran

32 Note: There are more efficient algorithms for computing Fibonacci numbers. This  
33 classic recursion algorithm is for illustrative purposes.



1 The following example demonstrates a way to generate a large number of tasks with one  
2 thread and execute them with the threads in the parallel team (see Section 2.7.3 on page  
3 63). While generating these tasks, the implementation may reach its limit on unassigned  
4 tasks. If it does, the implementation is allowed to cause the thread executing the task  
5 generating loop to suspend its task at the task scheduling point in the **task** directive,  
6 and start executing unassigned tasks. Once the number of unassigned tasks is  
7 sufficiently low, the thread may resume execution of the task generating loop.

C/C++

*Example A.13.5c*

```
8 #define LARGE_NUMBER 10000000
9 double item[LARGE_NUMBER];
10 extern void process(double);
11
12 int main() {
13 #pragma omp parallel
14 {
15     #pragma omp single
16     {
17         int i;
18         for (i=0; i<LARGE_NUMBER; i++)
19             #pragma omp task // i is firstprivate, item is shared
20                 process(item[i]);
21     }
22 }
23 }
```

C/C++

Fortran

*Example A.13.5f*

```
24
25     real*8 item(10000000)
26     integer i
27
28     !$omp parallel
29     !$omp single ! loop iteration variable i is private
30     do i=1,10000000
31     !$omp task
32         ! i is firstprivate, item is shared
33         call process(item(i))
34     !$omp end task
35     end do
36     !$omp end single
37     !$omp end parallel
38     end
```

Fortran

1 The following example is the same as the previous one, except that the tasks are  
2 generated in an untied task (see Section 2.7 on page 59). While generating the tasks, the  
3 implementation may reach its limit on unassigned tasks. If it does, the implementation is  
4 allowed to cause the thread executing the task generating loop to suspend its task at the  
5 task scheduling point in the `task` directive, and start executing unassigned tasks. If  
6 that thread begins execution of a task that takes a long time to complete, the other  
7 threads may complete all the other tasks before it is finished.

8 In this case, since the loop is in an untied task, any other thread is eligible to resume the  
9 task generating loop. In the previous examples, the other threads would be forced to  
10 wait idly until the generating thread finishes its long task, since the task generating loop  
11 was in a tied task.

C/C++

*Example A.13.6c*

```
12 #define LARGE_NUMBER 10000000
13 double item[LARGE_NUMBER];
14 extern void process(double);
15 int main() {
16 #pragma omp parallel
17 {
18     #pragma omp single
19     {
20         int i;
21         #pragma omp task untied
22         // i is firstprivate, item is shared
23         {
24             for (i=0; i<LARGE_NUMBER; i++)
25                 #pragma omp task
26                 process(item[i]);
27         }
28     }
29 }
30 }
```

C/C++

*Example A.13.6f*

```

1           real*8 item(10000000)
2       !$omp parallel
3       !$omp single
4       !$omp task untied
5           ! loop iteration variable i is private
6           do i=1,10000000
7       !$omp task ! i is firstprivate, item is shared
8           call process(item(i))
9       !$omp end task
10          end do
11       !$omp end task
12       !$omp end single
13       !$omp end parallel
14       end

```

15 The following two examples demonstrate how the scheduling rules illustrated in  
16 Section 2.7.3 on page 63 affect the usage of **threadprivate** variables in tasks. A  
17 **threadprivate** variable can be modified by another task that is executed by the  
18 same thread. Thus, the value of a **threadprivate** variable cannot be assumed to be  
19 unchanged across a task scheduling point. In untied tasks, task scheduling points may be  
20 added in any place by the implementation.

21 A task switch may occur at a task scheduling point. A single thread may execute both of  
22 the task regions that modify *tp*. The parts of these task regions in which *tp* is modified  
23 may be executed in any order so the resulting value of *var* can be either 1 or 2.

*Example A.13.7c*

```

1      int tp;
2      #pragma omp threadprivate(tp)
3      int var;
4      void work()
5      {
6          #pragma omp task
7              {
8                  /* do work here */
9          #pragma omp task
10             {
11                 tp = 1;
12                 /* do work here */
13          #pragma omp task
14              {
15                  /* no modification of tp */
16              }
17              var = tp; //value of tp can be 1 or 2
18          }
19          tp = 2;
20      }
21  }
```

*Example A.13.7f*

```

22      module example
23      integer tp
24      !$omp threadprivate(tp)
25      integer var
26      contains
27      subroutine work
28      use globals
29      !$omp task
30          ! do work here
31      !$omp task
32          tp = 1
33          ! do work here
34      !$omp task
35          ! no modification of tp
36      !$omp end task
37          var = tp      ! value of var can be 1 or 2
38      !$omp end task
39          tp = 2
40      !$omp end task
41      end subroutine
42      end module
```

1  
2  
3  
4

In this example, scheduling constraints (see Section 2.7.3 on page 63) prohibit a thread in the team from executing a new task that modifies `tp` while another such task region tied to the same thread is suspended. Therefore, the value written will persist across the task scheduling point.

C/C++

*Example A.13.8c*

```
5 #include <omp.h>
6 int tp;
7 #pragma omp threadprivate(tp)
8 int var;
9 void work()
10 {
11 #pragma omp parallel
12     {
13         /* do work here */
14 #pragma omp task
15         {
16             tp++;
17             /* do work here */
18 #pragma omp task
19             {
20                 /* do work here but don't modify tp */
21             }
22             var = tp; //Value does not change after write above
23         }
24     }
25 }
```

C/C++

Fortran

*Example A.13.8f*

```
26 module example
27     integer tp
28 !$omp threadprivate(tp)
29     integer var
30     contains
31     subroutine work
32 !$omp parallel
33     ! do work here
34 !$omp task
35     tp = tp + 1
36     ! do work here
37 !$omp task
38     ! do work here but don't modify tp
39 !$omp end task
40     var = tp ! value does not change after write above
41 !$omp end task
42 !$omp end parallel
43 end subroutine
```

1

end module

Fortran

2

The following two examples demonstrate how the scheduling rules illustrated in Section 2.7.3 on page 63 affect the usage of locks and critical sections in tasks. If a lock is held across a task scheduling point, no attempt should be made to acquire the same lock in any code that may be interleaved. Otherwise, a deadlock is possible.

3

4

5

6

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

7

8

C/C++

*Example A.13.9c*

9

```

void work()
{
  #pragma omp task
  { //Task 1
    #pragma omp task
    { //Task 2
      #pragma omp critical //Critical region 1
      { /*do work here */ }
    }
    #pragma omp critical //Critical Region 2
    {
      //Capture data for the following task
      #pragma omp task
      { /* do work here */ } //Task 3
    }
  }
}

```

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

C/C++

1

*Example A.13.9f*

```
2           module example
3           contains
4           subroutine work
5           !$omp task
6             ! Task 1
7           !$omp task
8             ! Task 2
9           !$omp critical
10            ! Critical region 1
11            ! do work here
12           !$omp end critical
13           !$omp end task
14           !$omp critical
15            ! Critical region 2
16            ! Capture data for the following task
17           !$omp task
18             !Task 3
19             ! do work here
20           !$omp end task
21           !$omp end critical
22           !$omp end task
23           end subroutine
24           end module
```

1 In the following example, *lock* is held across a task scheduling point. However,  
2 according to the scheduling restrictions outlined in Section 2.7.3 on page 63, the  
3 executing thread can't begin executing one of the non-descendant tasks that also acquires  
4 *lock* before the task region is complete. Therefore, no deadlock is possible.

C/C++

*Example A.13.10c*

```
5 #include <omp.h>
6 void work() {
7     omp_lock_t lock;
8     omp_init_lock(&lock);
9 #pragma omp parallel
10 {
11     int i;
12 #pragma omp for
13     for (i = 0; i < 100; i++) {
14 #pragma omp task
15     {
16         // lock is shared by default in the task
17         omp_set_lock(&lock);
18         // Capture data for the following task
19 #pragma omp task
20         // Task Scheduling Point 1
21         { /* do work here */ }
22         omp_unset_lock(&lock);
23     }
24 }
25 }
26     omp_destroy_lock(&lock);
27 }
```

C/C++



*Example A.13.10f*

```

1      module example
2      include 'omp_lib.h'
3      integer (kind=omp_lock_kind) lock
4      integer i
5      contains
6      subroutine work
7      call omp_init_lock(lock)
8      !$omp parallel
9      !$omp do
10     do i=1,100
11         !$omp task
12             ! Outer task
13             call omp_set_lock(lock)    ! lock is shared by
14                                         ! default in the task
15                                         ! Capture data for the following task
16             !$omp task    ! Task Scheduling Point 1
17                 ! do work here
18             !$omp end task
19             call omp_unset_lock(lock)
20         !$omp end task
21     end do
22 !$omp end parallel
23 call omp_destroy_lock(lock)
24 end subroutine
25 end module

```

## 26 A.14 The workshare Construct

27 The following are examples of the **workshare** construct (see Section 2.5.4 on page  
28 51).

▼ ----- Fortran (cont.) ----- ▼

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

*Example A.14.1f*

```
4          SUBROUTINE WSHARE1(AA, BB, CC, DD, EE, FF, N)
5          INTEGER N
6          REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N), EE(N,N), FF(N,N)

7          !$OMP PARALLEL
8          !$OMP WORKSHARE
9              AA = BB
10             CC = DD
11             EE = FF
12          !$OMP END WORKSHARE
13          !$OMP END PARALLEL

14          END SUBROUTINE WSHARE1
```

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

*Example A.14.2f*

```
18          SUBROUTINE WSHARE2(AA, BB, CC, DD, EE, FF, N)
19          INTEGER N
20          REAL AA(N,N), BB(N,N), CC(N,N)
21          REAL DD(N,N), EE(N,N), FF(N,N)

22          !$OMP PARALLEL
23          !$OMP WORKSHARE
24              AA = BB
25              CC = DD
26          !$OMP END WORKSHARE NOWAIT
27          !$OMP WORKSHARE
28              EE = FF
29          !$OMP END WORKSHARE
30          !$OMP END PARALLEL
31          END SUBROUTINE WSHARE2
```

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

*Example A.14.3f*

```

3           SUBROUTINE WSHARE3 (AA, BB, CC, DD, N)
4           INTEGER N
5           REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
6           REAL R

7           R=0
8           !$OMP PARALLEL
9           !$OMP WORKSHARE
10           AA = BB
11           !$OMP ATOMIC UPDATE
12           R = R + SUM(AA)
13           CC = DD
14           !$OMP END WORKSHARE
15           !$OMP END PARALLEL

16           END SUBROUTINE WSHARE3
  
```

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

21           Each task gets worked on in order by the threads:

```

22           AA = BB then
23           CC = DD then
24           EE .ne. 0 then
25           FF = 1 / EE then
26           GG = HH
  
```

*Example A.14.4f*

```

1          SUBROUTINE WSHARE4(AA, BB, CC, DD, EE, FF, GG, HH, 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          REAL GG(N,N), HH(N,N)

6          !$OMP PARALLEL
7          !$OMP WORKSHARE
8              AA = BB
9              CC = DD
10             WHERE (EE .ne. 0) FF = 1 / EE
11             GG = HH
12          !$OMP END WORKSHARE
13          !$OMP END PARALLEL
14
15          END SUBROUTINE WSHARE4

```

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

*Example A.14.5f*

```

18          SUBROUTINE WSHARE5(AA, BB, CC, DD, N)
19          INTEGER N
20          REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)

21          INTEGER SHR

22          !$OMP PARALLEL SHARED(SHR)
23          !$OMP WORKSHARE
24              AA = BB
25              SHR = 1
26              CC = DD * SHR
27          !$OMP END WORKSHARE
28          !$OMP END PARALLEL
29
30          END SUBROUTINE WSHARE5

```

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

### Example A.14.6f

```
1      SUBROUTINE WSHARE6_WRONG(AA, BB, CC, DD, N)
2      INTEGER N
3      REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
4
5      INTEGER PRI
6
7      !$OMP PARALLEL PRIVATE(PRI)
8      WORKSHARE
9      AA = BB
10     PRI = 1
11     CC = DD * PRI
12     !$OMP END WORKSHARE
13     !$OMP END PARALLEL
14
15     END SUBROUTINE WSHARE6_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.14.7f

```
17     SUBROUTINE WSHARE7(AA, BB, CC, N)
18     INTEGER N
19     REAL AA(N), BB(N), CC(N)
20
21     !$OMP PARALLEL
22     WORKSHARE
23     AA(1:50) = BB(11:60)
24     CC(11:20) = AA(1:10)
25     !$OMP END WORKSHARE
26     !$OMP END PARALLEL
27
28     END SUBROUTINE WSHARE7
```

Fortran

## 27 A.15 The master Construct

28 The following example demonstrates the master construct (Section 2.8.1 on page 65). In  
29 the example, the master keeps track of how many iterations have been executed and  
30 prints out a progress report. The other threads skip the master region without waiting.

*Example A.15.1c*

```

1      #include <stdio.h>
2
3      extern float average(float,float,float);
4
5      void master_example( float* x, float* xold, int n, float tol )
6      {
7          int c, i, toobig;
8          float error, y;
9          c = 0;
10         #pragma omp parallel
11         {
12             do{
13                 #pragma omp for private(i)
14                 for( i = 1; i < n-1; ++i ){
15                     xold[i] = x[i];
16                 }
17                 #pragma omp single
18                 {
19                     toobig = 0;
20                 }
21                 #pragma omp for private(i,y,error) reduction(+:toobig)
22                 for( i = 1; i < n-1; ++i ){
23                     y = x[i];
24                     x[i] = average( xold[i-1], x[i], xold[i+1] );
25                     error = y - x[i];
26                     if( error > tol || error < -tol ) ++toobig;
27                 }
28                 #pragma omp master
29                 {
30                     ++c;
31                     printf( "iteration %d, toobig=%d\n", c, toobig );
32                 }
33             }while( toobig > 0 );
34         }
35     }

```

*Example A.15.1f*

```

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

```

## 32 A.16 The critical Construct

33 The following example includes several **critical** constructs (Section 2.8.2 on page  
34 67). The example illustrates a queuing model in which a task is dequeued and worked  
35 on. To guard against multiple threads dequeuing the same task, the dequeuing operation  
36 must be in a **critical** region. Because the two queues in this example are  
37 independent, they are protected by **critical** constructs with different names, *xaxis*  
38 and *yaxis*.

## C/C++

*Example A.16.1c*

```

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

3      void critical_example(float *x, float *y)
4      {
5          int ix_next, iy_next;

6          #pragma omp parallel shared(x, y) private(ix_next, iy_next)
7          {
8              #pragma omp critical (xaxis)
9                  ix_next = dequeue(x);
10             work(ix_next, x);

11             #pragma omp critical (yaxis)
12                 iy_next = dequeue(y);
13             work(iy_next, y);
14         }
15     }

```

## C/C++

## Fortran

*Example A.16.1f*

```

16      SUBROUTINE CRITICAL_EXAMPLE(X, Y)
17          REAL X(*), Y(*)
18          INTEGER IX_NEXT, IY_NEXT

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

20      !$OMP CRITICAL(XAXIS)
21          CALL DEQUEUE(IX_NEXT, X)
22      !$OMP END CRITICAL(XAXIS)
23          CALL WORK(IX_NEXT, X)

24      !$OMP CRITICAL(YAXIS)
25          CALL DEQUEUE(IY_NEXT, Y)
26      !$OMP END CRITICAL(YAXIS)
27          CALL WORK(IY_NEXT, Y)

28      !$OMP END PARALLEL

29      END SUBROUTINE CRITICAL_EXAMPLE

```

## Fortran



---

## 1 A.17 worksharing Constructs Inside a 2 critical Construct

3 The following example demonstrates using a worksharing construct inside a **critical**  
4 construct (see Section 2.8.2 on page 67). This example is conforming because the  
5 worksharing **single** region is not closely nested inside the critical region (see  
6 Section 2.10 on page 109). A single thread executes the one and only section in the  
7 sections region, and executes the critical region. The same thread encounters the nested  
8 parallel region, creates a new team of threads, and becomes the master of the new team.  
9 One of the threads in the new team enters the single region and increments **i** by 1. At  
10 the end of this example **i** is equal to 2.

C/C++

*Example A.17.1c*

```
11 void critical_work()  
12 {  
13     int i = 1;  
14     #pragma omp parallel sections  
15     {  
16         #pragma omp section  
17         {  
18             #pragma omp critical (name)  
19             {  
20                 #pragma omp parallel  
21                 {  
22                     #pragma omp single  
23                     {  
24                         i++;  
25                     }  
26                 }  
27             }  
28         }  
29     }  
30 }
```

C/C++

*Example A.17.1f*

```

1          SUBROUTINE CRITICAL_WORK()
2
3          INTEGER I
4          I = 1
5
6          !$OMP PARALLEL SECTIONS
7          !$OMP SECTION
8          !$OMP CRITICAL (NAME)
9          !$OMP PARALLEL
10         !$OMP SINGLE
11         I = I + 1
12         !$OMP END SINGLE
13         !$OMP END PARALLEL
14         !$OMP END CRITICAL (NAME)
15         !$OMP END PARALLEL SECTIONS
16         END SUBROUTINE CRITICAL_WORK

```

## 15 A.18 Binding of barrier Regions

16 The binding rules call for a **barrier** region to bind to the closest enclosing  
 17 **parallel** region (see Section 2.8.3 on page 68).

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

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

*Example A.18.1c*

```
1      void work(int n) {}
2
3      void sub3(int n)
4      {
5          work(n);
6          #pragma omp barrier
7          work(n);
8      }
9
10     void sub2(int k)
11     {
12         #pragma omp parallel shared(k)
13         sub3(k);
14     }
15
16     void sub1(int n)
17     {
18         int i;
19         #pragma omp parallel private(i) shared(n)
20         {
21             #pragma omp for
22             for (i=0; i<n; i++)
23                 sub2(i);
24         }
25     }
26
27     int main()
28     {
29         sub1(2);
30         sub2(2);
31         sub3(2);
32         return 0;
33     }
```

*Example A.18.1f*

```

1          SUBROUTINE WORK (N)
2              INTEGER N
3          END SUBROUTINE WORK

4          SUBROUTINE SUB3 (N)
5              INTEGER N
6              CALL WORK (N)
7          !$OMP BARRIER
8              CALL WORK (N)
9          END SUBROUTINE SUB3

10         SUBROUTINE SUB2 (K)
11             INTEGER K
12         !$OMP PARALLEL SHARED (K)
13             CALL SUB3 (K)
14         !$OMP END PARALLEL
15         END SUBROUTINE SUB2

16         SUBROUTINE SUB1 (N)
17             INTEGER N
18             INTEGER I
19         !$OMP PARALLEL PRIVATE (I) SHARED (N)
20         !$OMP DO
21             DO I = 1, N
22                 CALL SUB2 (I)
23             END DO
24         !$OMP END PARALLEL
25         END SUBROUTINE SUB1

26         PROGRAM EXAMPLE
27             CALL SUB1 (2)
28             CALL SUB2 (2)
29             CALL SUB3 (2)
30         END PROGRAM EXAMPLE

```

## 31 A.19 The atomic Construct

32 The following example avoids race conditions (simultaneous updates of an element of  $x$   
33 by multiple threads) by using the **atomic** construct (Section 2.8.5 on page 71).

1 The advantage of using the **atomic** construct in this example is that it allows updates  
2 of two different elements of  $x$  to occur in parallel. If a **critical** construct (see  
3 Section 2.8.2 on page 67) were used instead, then all updates to elements of  $x$  would be  
4 executed serially (though not in any guaranteed order).

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

C/C++

### *Example A.19.1c*

```
7 float work1(int i)
8 {
9     return 1.0 * i;
10 }

11 float work2(int i)
12 {
13     return 2.0 * i;
14 }

15 void atomic_example(float *x, float *y, int *index, int n)
16 {
17     int i;

18     #pragma omp parallel for shared(x, y, index, n)
19     for (i=0; i<n; i++) {
20         #pragma omp atomic update
21         x[index[i]] += work1(i);
22         y[i] += work2(i);
23     }
24 }

25 int main()
26 {
27     float x[1000];
28     float y[10000];
29     int index[10000];
30     int i;

31     for (i = 0; i < 10000; i++) {
32         index[i] = i % 1000;
33         y[i]=0.0;
34     }
35     for (i = 0; i < 1000; i++)
36         x[i] = 0.0;
37     a19(x, y, index, 10000);
38     return 0;
39 }
```

C/C++

*Example A.19.1f*

```

1          REAL FUNCTION WORK1(I)
2            INTEGER I
3            WORK1 = 1.0 * I
4            RETURN
5          END FUNCTION WORK1

6          REAL FUNCTION WORK2(I)
7            INTEGER I
8            WORK2 = 2.0 * I
9            RETURN
10         END FUNCTION WORK2

11         SUBROUTINE SUB(X, Y, INDEX, N)
12           REAL X(*), Y(*)
13           INTEGER INDEX(*), N

14           INTEGER I

15         !$OMP PARALLEL DO SHARED(X, Y, INDEX, N)
16           DO I=1,N
17         !$OMP   ATOMIC UPDATE
18             X(INDEX(I)) = X(INDEX(I)) + WORK1(I)
19             Y(I) = Y(I) + WORK2(I)
20           ENDDO

21         END SUBROUTINE SUB

22         PROGRAM ATOMIC_EXAMPLE
23           REAL X(1000), Y(10000)
24           INTEGER INDEX(10000)
25           INTEGER I

26           DO I=1,10000
27             INDEX(I) = MOD(I, 1000) + 1
28             Y(I) = 0.0
29           ENDDO

30           DO I = 1,1000
31             X(I) = 0.0
32           ENDDO

33           CALL SUB(X, Y, INDEX, 10000)
34
35         END PROGRAM ATOMIC_EXAMPLE
36
37

```

## 1 A.20 Restrictions on the `atomic` Construct

2 The following non-conforming examples illustrate the restrictions on the `atomic`  
3 construct given in Section 2.8.5 on page 71.

C/C++

*Example A.20.1c*

```
4 void atomic_wrong ()
5 {
6     union {int n; float x;} u;
7
8     #pragma omp parallel
9     {
10    #pragma omp atomic update
11        u.n++;
12
13    #pragma omp atomic update
14        u.x += 1.0;
15
16    /* Incorrect because the atomic constructs reference the same location
17       through incompatible types */
18    }
19 }
```

C/C++

Fortran

*Example A.20.1f*

```
17 SUBROUTINE ATOMIC_WRONG()
18     INTEGER:: I
19     REAL:: R
20     EQUIVALENCE(I,R)
21
22     !$OMP PARALLEL
23     !$OMP ATOMIC UPDATE
24         I = I + 1
25     !$OMP ATOMIC UPDATE
26         R = R + 1.0
27     ! incorrect because I and R reference the same location
28     ! but have different types
29     !$OMP END PARALLEL
30     END SUBROUTINE ATOMIC_WRONG
```

Fortran

*Example A.20.2c*

```
1      void atomic_wrong2 ()
2      {
3          int x;
4          int *i;
5          float *r;

6          i = &x;
7          r = (float *)&x;

8          #pragma omp parallel
9              {
10             #pragma omp atomic update
11                 *i += 1;

12             #pragma omp atomic update
13                 *r += 1.0;

14             /* Incorrect because the atomic constructs reference the same location
15                through incompatible types */

16         }
17     }
```



1 The following example is non-conforming because *I* and *R* reference the same location  
2 but have different types.

*Example A.20.2f*

```
3           SUBROUTINE SUB()
4             COMMON /BLK/ R
5             REAL R
6
7             !$OMP   ATOMIC UPDATE
8               R = R + 1.0
9             END SUBROUTINE SUB
10
11           SUBROUTINE ATOMIC_WRONG2()
12             COMMON /BLK/ I
13             INTEGER I
14
15             !$OMP   PARALLEL
16               !$OMP   ATOMIC UPDATE
17                 I = I + 1
18                 CALL SUB()
19             END PARALLEL
20           END SUBROUTINE ATOMIC_WRONG2
```

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

*Example A.20.3f*

```
3          SUBROUTINE ATOMIC_WRONG3
4              INTEGER:: I
5              REAL:: R
6              EQUIVALENCE(I,R)

7          !$OMP PARALLEL
8          !$OMP ATOMIC UPDATE
9              I = I + 1
10         ! incorrect because I and R reference the same location
11         ! but have different types
12         !$OMP END PARALLEL

13         !$OMP PARALLEL
14         !$OMP ATOMIC UPDATE
15             R = R + 1.0
16         ! incorrect because I and R reference the same location
17         ! but have different types
18         !$OMP END PARALLEL

19     END SUBROUTINE ATOMIC_WRONG3
```

Fortran

---

## 20 A.21 The `flush` Construct without a List

21 The following example (for Section 2.8.6 on page 76) distinguishes the shared variables  
22 affected by a `flush` construct with no list from the shared objects that are not affected:

C/C++

*Example A.21.1c*

```
23     int x, *p = &x;

24     void f1(int *q)
25     {
26         *q = 1;
27         #pragma omp flush
28         /* x, p, and *q are flushed */
29         /* because they are shared and accessible */
30         /* q is not flushed because it is not shared. */
31     }
```

```

1      void f2(int *q)
2      {
3          #pragma omp barrier
4          *q = 2;
5          #pragma omp barrier

6          /* a barrier implies a flush */
7          /* x, p, and *q are flushed */
8          /* because they are shared and accessible */
9          /* q is not flushed because it is not shared. */
10     }

11     int g(int n)
12     {
13         int i = 1, j, sum = 0;
14         *p = 1;
15         #pragma omp parallel reduction(+: sum) num_threads(10)
16         {
17             f1(&j);

18             /* i, n and sum were not flushed */
19             /* because they were not accessible in f1 */
20             /* j was flushed because it was accessible */
21             sum += j;

22             f2(&j);

23             /* i, n, and sum were not flushed */
24             /* because they were not accessible in f2 */
25             /* j was flushed because it was accessible */
26             sum += i + j + *p + n;
27         }
28         return sum;
29     }

30     int main()
31     {
32         int result = g(7);
33         return result;
34     }

```

▲ C/C++ ▲

▼ Fortran ▼

### *Example A.21.1f*

```

35     SUBROUTINE F1(Q)
36         COMMON /DATA/ X, P
37         INTEGER, TARGET :: X
38         INTEGER, POINTER :: P
39         INTEGER Q

```

```

1           Q = 1
2   !$OMP   FLUSH
3           ! X, P and Q are flushed
4           ! because they are shared and accessible
5           END SUBROUTINE F1

6           SUBROUTINE F2(Q)
7           COMMON /DATA/ X, P
8           INTEGER, TARGET :: X
9           INTEGER, POINTER :: P
10          INTEGER Q

11          !$OMP   BARRIER
12              Q = 2
13          !$OMP   BARRIER
14              ! a barrier implies a flush
15              ! X, P and Q are flushed
16              ! because they are shared and accessible
17          END SUBROUTINE F2

18          INTEGER FUNCTION G(N)
19          COMMON /DATA/ X, P
20          INTEGER, TARGET :: X
21          INTEGER, POINTER :: P
22          INTEGER N
23          INTEGER I, J, SUM

24          I = 1
25          SUM = 0
26          P = 1
27          !$OMP   PARALLEL REDUCTION(+: SUM) NUM_THREADS(10)
28              CALL F1(J)
29              ! I, N and SUM were not flushed
30              ! because they were not accessible in F1
31              ! J was flushed because it was accessible
32              SUM = SUM + J

33              CALL F2(J)
34              ! I, N, and SUM were not flushed
35              ! because they were not accessible in f2
36              ! J was flushed because it was accessible
37              SUM = SUM + I + J + P + N
38          !$OMP   END PARALLEL

39          G = SUM
40          END FUNCTION G

41          PROGRAM FLUSH_NOLIST
42          COMMON /DATA/ X, P
43          INTEGER, TARGET :: X
44          INTEGER, POINTER :: P
45          INTEGER RESULT, G

```

```
1      P => X
2      RESULT = G(7)
3      PRINT *, RESULT
4      END PROGRAM FLUSH_NOLIST
```

Fortran

---

## 1 A.22 Placement of `flush`, `barrier`, and 2 `taskwait` Directives

3 The following example is non-conforming, because the `flush`, `barrier`, `taskwait`,  
4 and `taskyield` directives are stand-alone directives and cannot be the immediate  
5 substatement of an `if` statement. See Section 2.8.3 on page 68, Section 2.8.6 on page  
6 76, Section 2.8.4 on page 70, and Section 2.7.2 on page 62.

C/C++

*Example A.22.1c*

```
7     void standalone_wrong()  
8     {  
9         int a = 1;  
  
10        if (a != 0)  
11            #pragma omp flush(a)  
12        /* incorrect as flush cannot be immediate substatement  
13           of if statement */  
  
14        if (a != 0)  
15            #pragma omp barrier  
16        /* incorrect as barrier cannot be immediate substatement  
17           of if statement */  
  
18        if (a!=0)  
19            #pragma omp taskyield  
20        /* incorrect as taskyield cannot be immediate substatement of if statement */  
  
21        if (a != 0)  
22            #pragma omp taskwait  
23        /* incorrect as taskwait cannot be immediate substatement  
24           of if statement */  
  
25    }
```

C/C++

26 The following example is non-conforming, because the `flush`, `barrier`, `taskwait`,  
27 and `taskyield` directives are stand-alone directives and cannot be the action  
28 statement of an `if` statement or a labeled branch target.

*Example A.22.1f*

```
1      SUBROUTINE STANDALONE_WRONG()
2          INTEGER A
3          A = 1
4          ! the FLUSH directive must not be the action statement
5          ! in an IF statement
6          IF (A .NE. 0) !$OMP FLUSH(A)

7          ! the BARRIER directive must not be the action statement
8          ! in an IF statement
9          IF (A .NE. 0) !$OMP BARRIER

10         ! the TASKWAIT directive must not be the action statement
11         ! in an IF statement
12         IF (A .NE. 0) !$OMP TASKWAIT

13         ! the TASKYIELD directive must not be the action statement
14         ! in an IF statement
15         IF (A .NE. 0) !$OMP TASKYIELD

16         GOTO 100

17         ! the FLUSH directive must not be a labeled branch target
18         ! statement
19         100 !$OMP FLUSH(A)
20         GOTO 200

21         ! the BARRIER directive must not be a labeled branch target
22         ! statement
23         200 !$OMP BARRIER
24         GOTO 300

25         ! the TASKWAIT directive must not be a labeled branch target
26         ! statement
27         300 !$OMP TASKWAIT
28         GOTO 400

29         ! the TASKYIELD directive must not be a labeled branch target
30         ! statement
31         400 !$OMP TASKYIELD

32     END SUBROUTINE
```

1 The following version of the above example is conforming because the **flush**,  
2 **barrier**, **taskwait**, and **taskyield** directives are enclosed in a compound  
3 statement.

C/C++

*Example A.22.2c*

```
4 void standalone_ok()
5 {
6     int a = 1;
7
8     #pragma omp parallel
9     {
10        if (a != 0) {
11            #pragma omp flush(a)
12        }
13        if (a != 0) {
14            #pragma omp barrier
15        }
16        if (a != 0) {
17            #pragma omp taskwait
18        }
19        if (a != 0) {
20            #pragma omp taskyield
21        }
22    }
```

C/C++

23 The following example is conforming because the **flush**, **barrier**, **taskwait**, and  
24 **taskyield** directives are enclosed in an **if** construct or follow the labeled branch  
25 target.

Fortran

*Example A.22.2f*

```
26 SUBROUTINE STANDALONE_OK ()
27     INTEGER A
28     A = 1
29     IF (A .NE. 0) THEN
30         !$OMP FLUSH(A)
31     ENDIF
32     IF (A .NE. 0) THEN
33         !$OMP BARRIER
34     ENDIF
35     IF (A .NE. 0) THEN
36         !$OMP TASKWAIT
37     ENDIF
```



```
1      IF (A .NE. 0) THEN
2          !$OMP TASKYIELD
3      ENDIF
4      GOTO 100
5      100 CONTINUE
6          !$OMP FLUSH(A)
7      GOTO 200
8      200 CONTINUE
9          !$OMP BARRIER
10     GOTO 300
11     300 CONTINUE
12     !$OMP TASKWAIT
13     GOTO 400
14     400 CONTINUE
15     !$OMP TASKYIELD
16 END SUBROUTINE
```

---

Fortran

---

## 17 **A.23**      **The ordered Clause and the ordered** 18 **Construct**

19            Ordered constructs (Section 2.8.7 on page 81) are useful for sequentially ordering the  
20            output from work that is done in parallel. The following program prints out the indices  
21            in sequential order:

*Example A.23.1c*

```
1      #include <stdio.h>
2
3      void work(int k)
4      {
5          #pragma omp ordered
6              printf(" %d\n", k);
7      }
8
9      void ordered_example(int lb, int ub, int stride)
10     {
11         #pragma omp parallel for ordered schedule(dynamic)
12             for (i=lb; i<ub; i+=stride)
13                 work(i);
14     }
15
16     int main()
17     {
18         ordered_example(0, 100, 5);
19         return 0;
20     }
```

*Example A.23.1f*

```

1          SUBROUTINE WORK(K)
2              INTEGER k

3          !$OMP ORDERED
4              WRITE(*,*) K
5          !$OMP END ORDERED

6          END SUBROUTINE WORK

7          SUBROUTINE SUB(LB, UB, STRIDE)
8              INTEGER LB, UB, STRIDE
9              INTEGER I

10         !$OMP PARALLEL DO ORDERED SCHEDULE(DYNAMIC)
11             DO I=LB,UB,STRIDE
12                 CALL WORK(I)
13             END DO
14         !$OMP END PARALLEL DO

15         END SUBROUTINE SUB

16         PROGRAM ORDERED_EXAMPLE
17             CALL SUB(1,100,5)
18         END PROGRAM ORDERED_EXAMPLE

```

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

*Example A.23.2c*

```

1      void work(int i) {}
2
3      void ordered_wrong(int n)
4      {
5          int i;
6          #pragma omp for ordered
7          for (i=0; i<n; i++) {
8              /* incorrect because an iteration may not execute more than one
9                 ordered region */
10             #pragma omp ordered
11             work(i);
12             #pragma omp ordered
13             work(i+1);
14         }
15     }

```

*Example A.23.2f*

```

15      SUBROUTINE WORK(I)
16      INTEGER I
17      END SUBROUTINE WORK
18
19      SUBROUTINE ORDERED_WRONG(N)
20      INTEGER N
21
22      INTEGER I
23      !$OMP DO ORDERED
24      DO I = 1, N
25          ! incorrect because an iteration may not execute more than one
26          ! ordered region
27          !$OMP ORDERED
28              CALL WORK(I)
29          !$OMP END ORDERED
30
31      !$OMP ORDERED
32          CALL WORK(I+1)
33      !$OMP END ORDERED
34
35      END DO
36      END SUBROUTINE ORDERED_WRONG

```

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

C/C++

---

*Example A.23.3c*

```
3 void work(int i) {}
4 void ordered_good(int n)
5 {
6     int i;
7
8     #pragma omp for ordered
9     for (i=0; i<n; i++) {
10         if (i <= 10) {
11             #pragma omp ordered
12                 work(i);
13         }
14         if (i > 10) {
15             #pragma omp ordered
16                 work(i+1);
17         }
18     }
```

---

C/C++

---

*Example A.23.3f*

```
19 SUBROUTINE ORDERED_GOOD(N)
20 INTEGER N
21 !$OMP DO ORDERED
22 DO I = 1,N
23     IF (I <= 10) THEN
24         !$OMP ORDERED
25             CALL WORK(I)
26         !$OMP END ORDERED
27     ENDIF
28
29     IF (I > 10) THEN
30         !$OMP ORDERED
31             CALL WORK(I+1)
32         !$OMP END ORDERED
33     ENDIF
34 ENDDO
END SUBROUTINE ORDERED_GOOD
```

---

Fortran

## 1 A.24 The threadprivate Directive

2 The following examples demonstrate how to use the `threadprivate` directive  
3 (Section 2.9.2 on page 86) to give each thread a separate counter.

Example A.24.1c C/C++

```
4 int counter = 0;  
5 #pragma omp threadprivate(counter)  
  
6 int increment_counter()  
7 {  
8     counter++;  
9     return(counter);  
10 }
```

C/C++

Example A.24.1f Fortran

```
11 INTEGER FUNCTION INCREMENT_COUNTER()  
12 COMMON/INC_COMMON/COUNTER  
13 !$OMP THREADPRIVATE (/INC_COMMON/)  
  
14 COUNTER = COUNTER +1  
15 INCREMENT_COUNTER = COUNTER  
16 RETURN  
17 END FUNCTION INCREMENT_COUNTER
```

Fortran

Example A.24.2c C/C++

18 The following example uses `threadprivate` on a static variable:

Example A.24.2c

```
19 int increment_counter_2()  
20 {  
21     static int counter = 0;  
22     #pragma omp threadprivate(counter)  
23     counter++;  
24     return(counter);  
25 }
```

1 The following example demonstrates unspecified behavior for the initialization of a  
2 **threadprivate** variable. A **threadprivate** variable is initialized once at an  
3 unspecified point before its first reference. Because *a* is constructed using the value of *x*  
4 (which is modified by the statement **x++**), the value of **a.val** at the start of the  
5 parallel region could be either 1 or 2. This problem is avoided for *b*, which uses an  
6 auxiliary **const** variable and a copy-constructor.

### *Example A.24.3c*

```
7     class T {
8         public:
9             int val;
10            T (int);
11            T (const T&);
12        };
13
14    T :: T (int v){
15        val = v;
16    }
17
18    T :: T (const T& t) {
19        val = t.val;
20    }
21
22    void g(T a, T b){
23        a.val += b.val;
24    }
25
26    int x = 1;
27    T a(x);
28    const T b_aux(x); /* Capture value of x = 1 */
29    T b(b_aux);
30    #pragma omp threadprivate(a, b)
31
32    void f(int n) {
33        x++;
34        #pragma omp parallel for
35        /* In each thread:
36         * a is constructed from x (with value 1 or 2?)
37         * b is copy-constructed from b_aux
38         */
39
40        for (int i=0; i<n; i++) {
41            g(a, b); /* Value of a is unspecified. */
42        }
43    }
44
```

1 The following examples show non-conforming uses and correct uses of the  
 2 **threadprivate** directive. For more information, see Section 2.9.2 on page 86 and  
 3 Section 2.9.4.1 on page 106.

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

*Example A.24.2f*

```

6           MODULE INC_MODULE
7             COMMON /T/ A
8           END MODULE INC_MODULE
9
10          SUBROUTINE INC_MODULE_WRONG()
11            USE INC_MODULE
12          !$OMP THREADPRIVATE(/T/)
13            !non-conforming because /T/ not declared in INC_MODULE_WRONG
14          END SUBROUTINE INC_MODULE_WRONG

```

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

*Example A.24.3f*

```

17          SUBROUTINE INC_WRONG()
18            COMMON /T/ A
19          !$OMP THREADPRIVATE(/T/)
20
21            CONTAINS
22              SUBROUTINE INC_WRONG_SUB()
23                !$OMP PARALLEL COPYIN(/T/)
24                !non-conforming because /T/ not declared in INC_WRONG_SUB
25                END PARALLEL
26              END SUBROUTINE INC_WRONG_SUB
27            END SUBROUTINE INC_WRONG

```



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

*Example A.24.4f*

```

2           SUBROUTINE INC_GOOD()
3           COMMON /T/ A
4       !$OMP    THREADPRIVATE(/T/)

5           CONTAINS
6           SUBROUTINE INC_GOOD_SUB()
7           COMMON /T/ A
8       !$OMP    THREADPRIVATE(/T/)

9       !$OMP    PARALLEL COPYIN(/T/)
10      !$OMP    END PARALLEL
11           END SUBROUTINE INC_GOOD_SUB
12      END SUBROUTINE INC_GOOD

```

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

*Example A.24.5f*

```

2          PROGRAM INC_GOOD2
3            INTEGER, ALLOCATABLE, SAVE :: A(:)
4            INTEGER, POINTER, SAVE :: PTR
5            INTEGER, SAVE :: I
6            INTEGER, TARGET :: TARG
7            LOGICAL :: FIRSTIN = .TRUE.
8    !$OMP  THREADPRIVATE(A, I, PTR)
9
10           ALLOCATE (A(3))
11           A = (/1,2,3/)
12           PTR => TARG
13           I = 5
14
15    !$OMP  PARALLEL COPYIN(I, PTR)
16    !$OMP  CRITICAL
17           IF (FIRSTIN) THEN
18             TARG = 4           ! Update target of ptr
19             I = I + 10
20             IF (ALLOCATED(A)) A = A + 10
21             FIRSTIN = .FALSE.
22           END IF
23
24           IF (ALLOCATED(A)) THEN
25             PRINT *, 'a = ', A
26           ELSE
27             PRINT *, 'A is not allocated'
28           END IF
29
30           PRINT *, 'ptr = ', PTR
31           PRINT *, 'i = ', I
32           PRINT *
33
34    !$OMP  END CRITICAL
35    !$OMP  END PARALLEL
36          END PROGRAM INC_GOOD2

```

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

```

37    a = 11 12 13
38    ptr = 4
39    i = 15
40
41    A is not allocated

```

```

1      ptr = 4
2      i = 5

3      or

4      A is not allocated
5      ptr = 4
6      i = 15

7      a = 1 2 3
8      ptr = 4
9      i = 5

```

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

*Example A.24.6f*

```

11      MODULE INC_MODULE_GOOD3
12          REAL, POINTER :: WORK(:)
13          SAVE WORK
14      !$OMP THREADPRIVATE(WORK)
15      END MODULE INC_MODULE_GOOD3
16
17      SUBROUTINE SUB1(N)
18          USE INC_MODULE_GOOD3
19      !$OMP PARALLEL PRIVATE(THE_SUM)
20          ALLOCATE(WORK(N))
21          CALL SUB2(THE_SUM)
22          WRITE(*,*) THE_SUM
23      !$OMP END PARALLEL
24      END SUBROUTINE SUB1
25
26      SUBROUTINE SUB2(THE_SUM)
27          USE INC_MODULE_GOOD3
28          WORK(:) = 10
29          THE_SUM=SUM(WORK)
30      END SUBROUTINE SUB2
31
32      PROGRAM INC_GOOD3
33          N = 10
34          CALL SUB1(N)
35      END PROGRAM INC_GOOD3

```

Fortran

C/C++

36 The following example illustrates initialization of threadprivate variables for class-type  
37 *T*. *t1* is default constructed, *t2* is constructed taking a constructor accepting one  
38 argument of integer type, *t3* is copy constructed with argument *f()*:



## 1 A.26 Fortran Restrictions on `shared` and 2 `private` Clauses with Common Blocks

3 When a named common block is specified in a `private`, `firstprivate`, or  
4 `lastprivate` clause of a construct, none of its members may be declared in another  
5 data-sharing attribute clause on that construct. The following examples illustrate this  
6 point. For more information, see Section 2.9.3 on page 90.

7 The following example is conforming:

*Example A.26.1f*

```
8           SUBROUTINE COMMON_GOOD()
9             COMMON /C/ X,Y
10            REAL X, Y
11
12            !$OMP PARALLEL PRIVATE (/C/)
13              ! do work here
14            !$OMP END PARALLEL
15
16            !$OMP PARALLEL SHARED (X,Y)
17              ! do work here
18            !$OMP END PARALLEL
19          END SUBROUTINE COMMON_GOOD
```

1 The following example is also conforming:

*Example A.26.2f*

```

2          SUBROUTINE COMMON_GOOD2 ()
3              COMMON /C/ X,Y
4              REAL X, Y
5
6              INTEGER I
7
6   !$OMP  PARALLEL
7   !$OMP  DO PRIVATE (/C/)
8           DO I=1,1000
9               ! do work here
10          ENDDO
11   !$OMP  END DO
12          !
13   !$OMP  DO PRIVATE(X)
14          DO I=1,1000
15              ! do work here
16          ENDDO
17   !$OMP  END DO
18   !$OMP  END PARALLEL
19          END SUBROUTINE COMMON_GOOD2

```

20 The following example is conforming:

*Example A.26.3f*

```

21          SUBROUTINE COMMON_GOOD3 ()
22              COMMON /C/ X,Y
23
23   !$OMP  PARALLEL PRIVATE (/C/)
24           ! do work here
25   !$OMP  END PARALLEL
26
26   !$OMP  PARALLEL SHARED (/C/)
27           ! do work here
28   !$OMP  END PARALLEL
29          END SUBROUTINE COMMON_GOOD3

```

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

*Example A.26.4f*

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

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

*Example A.26.5f*

```
11          SUBROUTINE COMMON_WRONG2()
12          COMMON /C/ X,Y
13          ! Incorrect: common block C cannot be declared both
14          ! shared and private
15          !$OMP PARALLEL PRIVATE (/C/), SHARED(/C/)
16              ! do work here
17          !$OMP END PARALLEL
18          END SUBROUTINE COMMON_WRONG2
```

Fortran

---

## 19 A.27 The default (none) Clause

20 The following example distinguishes the variables that are affected by the  
21 **default (none)** clause from those that are not. For more information on the  
22 **default** clause, see Section 2.9.3.1 on page 91.

*Example A.27.1c*

```
1      #include <omp.h>
2      int x, y, z[1000];
3      #pragma omp threadprivate(x)

4      void default_none(int a) {
5          const int c = 1;
6          int i = 0;

7          #pragma omp parallel default(none) private(a) shared(z)
8          {
9              int j = omp_get_num_threads();
10             /* O.K. - j is declared within parallel region */
11             a = z[j]; /* O.K. - a is listed in private clause */
12             /* - z is listed in shared clause */
13             x = c; /* O.K. - x is threadprivate */
14             /* - c has const-qualified type */
15             z[i] = y; /* Error - cannot reference i or y here */

16             #pragma omp for firstprivate(y)
17             /* Error - Cannot reference y in the firstprivate clause */
18             for (i=0; i<10 ; i++) {
19                 z[i] = i; /* O.K. - i is the loop iteration variable */
20             }

21             z[i] = y; /* Error - cannot reference i or y here */
22         }
23     }
```



*Example A.27.1f*

```

1          SUBROUTINE DEFAULT_NONE(A)
2          INCLUDE "omp_lib.h"      ! or USE OMP_LIB
3
4          INTEGER A

5          INTEGER X, Y, Z(1000)
6          COMMON/BLOCKX/X
7          COMMON/BLOCKY/Y
8          COMMON/BLOCKZ/Z
9  !$OMP THREADPRIVATE (/BLOCKX/)

10         INTEGER I, J
11         i = 1

12  !$OMP PARALLEL DEFAULT(NONE) PRIVATE(A) SHARED(Z) PRIVATE(J)
13         J = OMP_GET_NUM_THREADS();
14         ! O.K. - J is listed in PRIVATE clause
15         A = Z(J) ! O.K. - A is listed in PRIVATE clause
16         ! - Z is listed in SHARED clause
17         X = 1    ! O.K. - X is THREADPRIVATE
18         Z(I) = Y ! Error - cannot reference I or Y here

19  !$OMP DO firstprivate(y)
20         ! Error - Cannot reference y in the firstprivate clause
21         DO I = 1,10
22             Z(I) = I ! O.K. - I is the loop iteration variable
23         END DO

24         Z(I) = Y    ! Error - cannot reference I or Y here
25  !$OMP END PARALLEL
26  END SUBROUTINE DEFAULT_NONE

```

**A.28****Race Conditions Caused by Implied Copies of Shared Variables in Fortran**

28

29

30

31

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 2.9.3.2 on page 93). The subroutine call

1 passing an array section argument may cause the compiler to copy the argument into a  
2 temporary location prior to the call and copy from the temporary location into the  
3 original variable when the subroutine returns. This copying would cause races in the  
4 **parallel** region.

*Example A.28.If*

```
5 SUBROUTINE SHARED_RACE
6
7     INCLUDE "omp_lib.h"      ! or USE OMP_LIB
8
9     REAL A(20)
10    INTEGER MYTHREAD
11
12    !$OMP PARALLEL SHARED(A) PRIVATE(MYTHREAD)
13
14    MYTHREAD = OMP_GET_THREAD_NUM()
15    IF (MYTHREAD .EQ. 0) THEN
16        CALL SUB(A(1:10)) ! compiler may introduce writes to A(6:10)
17    ELSE
18        A(6:10) = 12
19    ENDIF
20
21    !$OMP END PARALLEL
22
23    END SUBROUTINE SHARED_RACE
24
25    SUBROUTINE SUB(X)
26        REAL X(*)
27        X(1:5) = 4
28    END SUBROUTINE SUB
```



Fortran

---

## 24 A.29 The private Clause

25 In the following example, the values of original list items *i* and *j* are retained on exit  
26 from the **parallel** region, while the private list items *i* and *j* are modified within the  
27 **parallel** construct. For more information on the **private** clause, see  
28 Section 2.9.3.3 on page 94.

*Example A.29.1c*

```

1      #include <stdio.h>
2      #include <assert.h>

3      int main()
4      {
5          int i, j;
6          int *ptr_i, *ptr_j;

7          i = 1;
8          j = 2;

9          ptr_i = &i;
10         ptr_j = &j;

11         #pragma omp parallel private(i) firstprivate(j)
12         {
13             i = 3;
14             j = j + 2;
15             assert (*ptr_i == 1 && *ptr_j == 2);
16         }

17         assert(i == 1 && j == 2);

18         return 0;
19     }

```

*Example A.29.1f*

```

20         PROGRAM PRIV_EXAMPLE
21             INTEGER I, J

22             I = 1
23             J = 2

24             !$OMP PARALLEL PRIVATE(I) FIRSTPRIVATE(J)
25                 I = 3
26                 J = J + 2
27             !$OMP END PARALLEL

28             PRINT *, I, J ! I .eq. 1 .and. J .eq. 2
29         END PROGRAM PRIV_EXAMPLE

```

1 In the following example, all uses of the variable *a* within the loop construct in the  
2 routine *f* refer to a private list item *a*, while it is unspecified whether references to *a* in  
3 the routine *g* are to a private list item or the original list item.

C/C++

*Example A.29.2c*

```
4     int a;
5
6     void g(int k) {
7         a = k; /* Accessed in the region but outside of the construct;
8                * therefore unspecified whether original or private list
9                * item is modified. */
10    }
11
12    void f(int n) {
13        int a = 0;
14
15        #pragma omp parallel for private(a)
16        for (int i=1; i<n; i++) {
17            a = i;
18            g(a*2); /* Private copy of "a" */
19        }
20    }
```

C/C++

*Example A.29.2f*

```

1      MODULE PRIV_EXAMPLE2
2          REAL A

3      CONTAINS

4          SUBROUTINE G(K)
5              REAL K
6              A = K ! Accessed in the region but outside of the
7                  ! construct; therefore unspecified whether
8                  ! original or private list item is modified.
9          END SUBROUTINE G

10         SUBROUTINE F(N)
11             INTEGER N
12             REAL A

13             INTEGER I
14             !$OMP PARALLEL DO PRIVATE(A)
15                 DO I = 1,N
16                     A = I
17                     CALL G(A*2)
18                 ENDDO
19             !$OMP END PARALLEL DO
20         END SUBROUTINE F

21     END MODULE PRIV_EXAMPLE2
22

```

23 The following example demonstrates that a list item that appears in a **private** clause  
24 in a **parallel** construct may also appear in a **private** clause in an enclosed  
25 worksharing construct, which results in an additional private copy.

C/C++

*Example A.29.3c*

```
1      #include <assert.h>
2      void priv_example3()
3      {
4          int i, a;

5          #pragma omp parallel private(a)
6          {
7              a = 1;
8              #pragma omp parallel for private(a)
9              for (i=0; i<10; i++)
10             {
11                 a = 2;
12             }
13             assert(a == 1);
14         }
15     }
```

C/C++

Fortran

*Example A.29.3f*

```
16          SUBROUTINE PRIV_EXAMPLE3()
17              INTEGER I, A

18          !$OMP PARALLEL PRIVATE(A)
19              A = 1
20          !$OMP PARALLEL DO PRIVATE(A)
21              DO I = 1, 10
22                  A = 2
23              END DO
24          !$OMP END PARALLEL DO
25              PRINT *, A ! Outer A still has value 1
26          !$OMP END PARALLEL
27          END SUBROUTINE PRIV_EXAMPLE3
```

Fortran

## 1 A.30 Fortran Restrictions on Storage Association with the `private` Clause

2

3

4

The following non-conforming examples illustrate the implications of the `private` clause rules with regard to storage association (see Section 2.9.3.3 on page 94).

### *Example A.30.1f*

```

5          SUBROUTINE SUB()
6          COMMON /BLOCK/ X
7          PRINT *,X           ! X is undefined
8          END SUBROUTINE SUB

9          PROGRAM PRIV_RESTRICT
10         COMMON /BLOCK/ X
11         X = 1.0
12         !$OMP PARALLEL PRIVATE (X)
13         X = 2.0
14         CALL SUB()
15         !$OMP END PARALLEL
16         END PROGRAM PRIV_RESTRICT

```

### *Example A.30.2f*

```

17         PROGRAM PRIV_RESTRICT2
18         COMMON /BLOCK2/ X
19         X = 1.0

20         !$OMP PARALLEL PRIVATE (X)
21         X = 2.0
22         CALL SUB()
23         !$OMP END PARALLEL

24         CONTAINS

25         SUBROUTINE SUB()
26         COMMON /BLOCK2/ Y

27         PRINT *,X           ! X is undefined
28         PRINT *,Y           ! Y is undefined
29         END SUBROUTINE SUB

30         END PROGRAM PRIV_RESTRICT2

```

*Example A.30.3f*

```

2          PROGRAM PRIV_RESTRICT3
3          EQUIVALENCE (X,Y)
4          X = 1.0

5          !$OMP PARALLEL PRIVATE(X)
6              PRINT *,Y                ! Y is undefined
7              Y = 10
8              PRINT *,X                ! X is undefined
9          !$OMP END PARALLEL
10         END PROGRAM PRIV_RESTRICT3

```

*Example A.30.4f*

```

11         PROGRAM PRIV_RESTRICT4
12         INTEGER I, J
13         INTEGER A(100), B(100)
14         EQUIVALENCE (A(51), B(1))

15         !$OMP PARALLEL DO DEFAULT(PRIVATE) PRIVATE(I,J) LASTPRIVATE(A)
16             DO I=1,100
17                 DO J=1,100
18                     B(J) = J - 1
19                 ENDDO

20                 DO J=1,100
21                     A(J) = J    ! B becomes undefined at this point
22                 ENDDO

23                 DO J=1,50
24                     B(J) = B(J) + 1 ! B is undefined
25                                     ! A becomes undefined at this point
26                 ENDDO
27             ENDDO
28         !$OMP END PARALLEL DO      ! The LASTPRIVATE write for A has
29                                     ! undefined results

30         PRINT *, B    ! B is undefined since the LASTPRIVATE
31                       ! write of A was not defined
32         END PROGRAM PRIV_RESTRICT4

```



### Example A.30.5f

```
1          SUBROUTINE SUB1(X)
2              DIMENSION X(10)
3
4              ! This use of X does not conform to the
5              ! specification. It would be legal Fortran 90,
6              ! but the OpenMP private directive allows the
7              ! compiler to break the sequence association that
8              ! A had with the rest of the common block.
9
10             FORALL (I = 1:10) X(I) = I
11         END SUBROUTINE SUB1
12
13         PROGRAM PRIV_RESTRICT5
14             COMMON /BLOCK5/ A
15
16             DIMENSION B(10)
17             EQUIVALENCE (A,B(1))
18
19             ! the common block has to be at least 10 words
20             A = 0
21
22             !$OMP PARALLEL PRIVATE(/BLOCK5/)
23                 ! Without the private clause,
24                 ! we would be passing a member of a sequence
25                 ! that is at least ten elements long.
26                 ! With the private clause, A may no longer be
27                 ! sequence-associated.
28
29                 CALL SUB1(A)
30             !$OMP MASTER
31                 PRINT *, A
32             !$OMP END MASTER
33
34             !$OMP END PARALLEL
35         END PROGRAM PRIV_RESTRICT5
```

Fortran

## 1 A.31 C/C++ Arrays in a `firstprivate` Clause

2 The following example illustrates the size and value of list items of array or pointer type  
3 in a `firstprivate` clause (Section 2.9.3.4 on page 97). The size of new list items is  
4 based on the type of the corresponding original list item, as determined by the base  
5 language.

6 In this example:

- 7 • The type of **A** is array of two arrays of two ints.
- 8 • The type of **B** is adjusted to pointer to array of **n** ints, because it is a function parameter.
- 9 • The type of **C** is adjusted to pointer to int, because it is a function parameter.
- 10 • The type of **D** is array of two arrays of two ints.
- 11 • The type of **E** is array of **n** arrays of **n** ints.

12 Note that **B** and **E** involve variable length array types.

13 The new items of array type are initialized as if each integer element of the original  
14 array is assigned to the corresponding element of the new array. Those of pointer type  
15 are initialized as if by assignment from the original item to the new item.

### Example A.31.1c

```
1      #include <assert.h>
2
3      int A[2][2] = {1, 2, 3, 4};
4
5      void f(int n, int B[n][n], int C[])
6      {
7          int D[2][2] = {1, 2, 3, 4};
8          int E[n][n];
9
10         assert(n >= 2);
11         E[1][1] = 4;
12
13         #pragma omp parallel firstprivate(B, C, D, E)
14         {
15             assert(sizeof(B) == sizeof(int (*)[n]));
16             assert(sizeof(C) == sizeof(int*));
17             assert(sizeof(D) == 4 * sizeof(int));
18             assert(sizeof(E) == n * n * sizeof(int));
19
20             /* Private B and C have values of original B and C. */
21             assert(&B[1][1] == &A[1][1]);
22             assert(&C[3] == &A[1][1]);
23             assert(D[1][1] == 4);
24             assert(E[1][1] == 4);
25         }
26     }
27
28     int main() {
29         f(2, A, A[0]);
30         return 0;
31     }
```

▲ C/C++ ▲

---

## 26 A.32 The lastprivate Clause

27 Correct execution sometimes depends on the value that the last iteration of a loop  
28 assigns to a variable. Such programs must list all such variables in a **lastprivate**  
29 clause (Section 2.9.3.5 on page 99) so that the values of the variables are the same as  
30 when the loop is executed sequentially.

C/C++

*Example A.32.1c*

```
1 void lastpriv (int n, float *a, float *b)
2 {
3     int i;
4
5     #pragma omp parallel
6     {
7         #pragma omp for lastprivate(i)
8         for (i=0; i<n-1; i++)
9             a[i] = b[i] + b[i+1];
10
11     a[i]=b[i];      /* i == n-1 here */
12 }
```

C/C++

Fortran

*Example A.32.1f*

```
12         SUBROUTINE LASTPRIV(N, A, B)
13
14             INTEGER N
15             REAL A(*), B(*)
16             INTEGER I
17
18             !$OMP PARALLEL
19             !$OMP DO LASTPRIVATE(I)
20
21                 DO I=1,N-1
22                     A(I) = B(I) + B(I+1)
23                 ENDDO
24
25             !$OMP END PARALLEL
26
27             A(I) = B(I)      ! I has the value of N here
28
29             END SUBROUTINE LASTPRIV
```

Fortran

## 24 A.33 The reduction Clause

25 The following example demonstrates the **reduction** clause (Section 2.9.3.6 on page  
26 101):

*Example A.33.1c*

```

1 void reduction1(float *x, int *y, int n)
2 {
3     int i, b;
4     float a;

5     a = 0.0;
6     b = 0;

7     #pragma omp parallel for private(i) shared(x, y, n) \
8         reduction(+:a) reduction(^:b)
9         for (i=0; i<n; i++) {
10         a += x[i];
11         b ^= y[i];
12     }
13 }

```

*Example A.33.1f*

```

14 SUBROUTINE REDUCTION1(A, B, X, Y, N)
15     INTEGER N
16     REAL X(*), Y(*), A, B

17     !$OMP PARALLEL DO PRIVATE(I) SHARED(X, N) REDUCTION(+:A)
18     !$OMP& REDUCTION(MIN:B)

19     DO I=1,N
20         A = A + X(I)
21         B = MIN(B, Y(I))

22     ! Note that some reductions can be expressed in
23     ! other forms. For example, the MIN could be expressed as
24     ! IF (B > Y(I)) B = Y(I)

25     END DO

26     END SUBROUTINE REDUCTION1

```

1 A common implementation of the preceding example is to treat it as if it had been  
2 written as follows:

C/C++

*Example A.33.2c*

```
3 void reduction2(float *x, int *y, int n)
4 {
5     int i, b, b_p;
6     float a, a_p;
7
8     a = 0.0;
9     b = 0;
10
11 #pragma omp parallel shared(a, b, x, y, n) \
12                        private(a_p, b_p)
13 {
14     a_p = 0.0;
15     b_p = 0;
16
17 #pragma omp for private(i)
18     for (i=0; i<n; i++) {
19
20         a_p += x[i];
21         b_p ^= y[i];
22
23     }
24
25 #pragma omp critical
26 {
27     a += a_p;
28     b ^= b_p;
29 }
30 }
```

C/C++

*Example A.33.2f*

```

1          SUBROUTINE REDUCTION2 (A, B, X, Y, N)
2
3          INTEGER N
4          REAL X(*), Y(*), A, B, A_P, B_P
5
6          !$OMP PARALLEL SHARED(X, Y, N, A, B) PRIVATE(A_P, B_P)
7
8              A_P = 0.0
9              B_P = HUGE(B_P)
10
11             !$OMP DO PRIVATE(I)
12             DO I=1,N
13                 A_P = A_P + X(I)
14                 B_P = MIN(B_P, Y(I))
15             ENDDO
16             !$OMP END DO
17
18             !$OMP CRITICAL
19                 A = A + A_P
20                 B = MIN(B, B_P)
21             !$OMP END CRITICAL
22
23             !$OMP END PARALLEL
24
25             END SUBROUTINE REDUCTION2

```

20 The following program is non-conforming because the reduction is on the *intrinsic*  
21 *procedure name* **MAX** but that name has been redefined to be the variable named **MAX**.

*Example A.33.3f*

```

22         PROGRAM REDUCTION_WRONG
23         MAX = HUGE(0)
24         M = 0
25
26         !$OMP PARALLEL DO REDUCTION(MAX: M)
27         ! MAX is no longer the intrinsic so this is non-conforming
28         DO I = 1, 100
29             CALL SUB(M,I)
30         END DO
31
32         END PROGRAM REDUCTION_WRONG
33
34         SUBROUTINE SUB(M,I)
35             M = MAX(M,I)
36         END SUBROUTINE SUB

```

1 The following conforming program performs the reduction using the *intrinsic procedure*  
2 *name* **MAX** even though the intrinsic **MAX** has been renamed to **REN**.

*Example A.33.4f*

```
3     MODULE M
4         INTRINSIC MAX
5     END MODULE M

6     PROGRAM REDUCTION3
7         USE M, REN => MAX
8         N = 0
9         !$OMP PARALLEL DO REDUCTION(REN: N)      ! still does MAX
10        DO I = 1, 100
11            N = MAX(N,I)
12        END DO
13    END PROGRAM REDUCTION3
```

14 The following conforming program performs the reduction using *intrinsic procedure*  
15 *name* **MAX** even though the intrinsic **MAX** has been renamed to **MIN**.

*Example A.33.5f*

```
16     MODULE MOD
17         INTRINSIC MAX, MIN
18     END MODULE MOD

19     PROGRAM REDUCTION4
20         USE MOD, MIN=>MAX, MAX=>MIN
21         REAL :: R
22         R = -HUGE(0.0)

23         !$OMP PARALLEL DO REDUCTION(MIN: R)      ! still does MAX
24         DO I = 1, 1000
25             R = MIN(R, SIN(REAL(I)))
26         END DO
27         PRINT *, R
28     END PROGRAM REDUCTION4
```

Fortran

29 The following example is non-conforming because the initialization ( $a = 0$ ) of the  
30 original list item "a" is not synchronized with the update of "a" as a result of the  
31 reduction computation in the for loop. Therefore, the example may print an incorrect  
32 value for "a".



1 To avoid this problem, the initialization of the original list item "a" should complete  
2 before any update of "a" as a result of the reduction clause. This can be achieved by  
3 adding an explicit barrier after the assignment a = 0, or by enclosing the assignment a =  
4 0 in a single directive (which has an implied barrier), or by initializing "a" before the  
5 start of the parallel region.

C/C++

*Example A.33.3c*

```
6      #include <stdio.h>
7
8      int main (void)
9      {
10         int a, i;
11
12         #pragma omp parallel shared(a) private(i)
13         {
14             #pragma omp master
15             a = 0;
16
17             // To avoid race conditions, add a barrier here.
18
19             #pragma omp for reduction(+:a)
20             for (i = 0; i < 10; i++) {
21                 a += i;
22             }
23
24             #pragma omp single
25             printf ("Sum is %d\n", a);
26         }
27     }
```

C/C++

*Example A.33.6f*

```

1           INTEGER A, I
2
3           !$OMP PARALLEL SHARED(A) PRIVATE(I)
4
5           !$OMP MASTER
6           A = 0
7           !$OMP END MASTER
8
9           ! To avoid race conditions, add a barrier here.
10
11          !$OMP DO REDUCTION(+:A)
12          DO I= 0, 9
13             A = A + I
14          END DO
15
16          !$OMP SINGLE
17          PRINT *, "Sum is ", A
18          !$OMP END SINGLE
19
20          !$OMP END PARALLEL
21          END

```

## 16 A.34 The `copyin` Clause

17 The `copyin` clause (see Section 2.9.4.1 on page 106) is used to initialize threadprivate  
 18 data upon entry to a `parallel` region. The value of the threadprivate variable in the  
 19 master thread is copied to the threadprivate variable of each other team member.

*Example A.34.1c*

```
1      #include <stdlib.h>
2
3      float* work;
4      int size;
5      float tol;
6
7      #pragma omp threadprivate(work,size,tol)
8
9      void build()
10     {
11         int i;
12         work = (float*)malloc( sizeof(float)*size );
13         for( i = 0; i < size; ++i ) work[i] = tol;
14     }
15
16     void copyin_example( float t, int n )
17     {
18         tol = t;
19         size = n;
20         #pragma omp parallel copyin(tol,size)
21         {
22             build();
23         }
24     }
```

*Example A.34.1f*

```

1      MODULE M
2          REAL, POINTER, SAVE :: WORK(:)
3          INTEGER :: SIZE
4          REAL :: TOL
5      !$OMP THREADPRIVATE(WORK,SIZE,TOL)
6      END MODULE M

7      SUBROUTINE COPYIN_EXAMPLE( T, N )
8          USE M
9          REAL :: T
10         INTEGER :: N
11         TOL = T
12         SIZE = N
13     !$OMP PARALLEL COPYIN(TOL,SIZE)
14         CALL BUILD
15     !$OMP END PARALLEL
16     END SUBROUTINE COPYIN_EXAMPLE

17     SUBROUTINE BUILD
18         USE M
19         ALLOCATE(WORK(SIZE))
20         WORK = TOL
21     END SUBROUTINE BUILD

```

## 22 A.35 The copyprivate Clause

23 The **copyprivate** clause (see Section 2.9.4.2 on page 107) can be used to broadcast  
 24 values acquired by a single thread directly to all instances of the private variables in the  
 25 other threads. In this example, if the routine is called from the sequential part, its  
 26 behavior is not affected by the presence of the directives. If it is called from a  
 27 **parallel** region, then the actual arguments with which *a* and *b* are associated must be  
 28 private.

29 The thread that executes the structured block associated with the **single** construct  
 30 broadcasts the values of the private variables *a*, *b*, *x*, and *y* from its implicit task's data  
 31 environment to the data environments of the other implicit tasks in the thread team. The  
 32 broadcast completes before any of the threads have left the barrier at the end of the  
 33 construct.

C/C++

*Example A.35.1c*

```
1      #include <stdio.h>
2      float x, y;
3      #pragma omp threadprivate(x, y)
4
5      void init(float a, float b ) {
6          #pragma omp single copyprivate(a,b,x,y)
7          {
8              scanf("%f %f %f %f", &a, &b, &x, &y);
9          }
10     }
```

C/C++

Fortran

*Example A.35.1f*

```
10     SUBROUTINE INIT(A,B)
11     REAL A, B
12     COMMON /XY/ X,Y
13     !$OMP THREADPRIVATE (/XY/)
14
15     !$OMP SINGLE
16     READ (11) A,B,X,Y
17     !$OMP END SINGLE COPYPRIVATE (A,B,/XY/)
18
19     END SUBROUTINE INIT
```

Fortran

1 In this example, assume that the input must be performed by the master thread. Since the  
2 **master** construct does not support the **copyprivate** clause, it cannot broadcast the  
3 input value that is read. However, **copyprivate** is used to broadcast an address where  
4 the input value is stored.

C/C++

*Example A.35.2c*

```
5      #include <stdio.h>
6      #include <stdlib.h>

7      float read_next( ) {
8          float * tmp;
9          float return_val;

10         #pragma omp single copyprivate(tmp)
11         {
12             tmp = (float *) malloc(sizeof(float));
13         } /* copies the pointer only */

14         #pragma omp master
15         {
16             scanf("%f", tmp);
17         }

18         #pragma omp barrier
19         return_val = *tmp;
20         #pragma omp barrier

21         #pragma omp single nowait
22         {
23             free(tmp);
24         }

25         return return_val;
26     }
```

C/C++

*Example A.35.2f*

```

1          REAL FUNCTION READ_NEXT()
2          REAL, POINTER :: TMP

3          !$OMP SINGLE
4              ALLOCATE (TMP)
5          !$OMP END SINGLE COPYPRIVATE (TMP) ! copies the pointer only

6          !$OMP MASTER
7              READ (11) TMP
8          !$OMP END MASTER

9          !$OMP BARRIER
10             READ_NEXT = TMP
11          !$OMP BARRIER

12         !$OMP SINGLE
13             DEALLOCATE (TMP)
14         !$OMP END SINGLE NOWAIT
15     END FUNCTION READ_NEXT

```

16 Suppose that the number of lock variables required within a **parallel** region cannot  
 17 easily be determined prior to entering it. The **copyprivate** clause can be used to  
 18 provide access to shared lock variables that are allocated within that **parallel** region.

*Example A.35.3c*

```

19     #include <stdio.h>
20     #include <stdlib.h>
21     #include <omp.h>

22     omp_lock_t *new_lock()
23     {
24         omp_lock_t *lock_ptr;

25         #pragma omp single copyprivate(lock_ptr)
26         {
27             lock_ptr = (omp_lock_t *) malloc(sizeof(omp_lock_t));
28             omp_init_lock( lock_ptr );
29         }

30         return lock_ptr;
31     }

```

*Example A.35.3f*

```

1      FUNCTION NEW_LOCK()
2      USE OMP_LIB      ! or INCLUDE "omp_lib.h"
3      INTEGER(OMP_LOCK_KIND), POINTER :: NEW_LOCK

4      !$OMP  SINGLE
5          ALLOCATE(NEW_LOCK)
6          CALL OMP_INIT_LOCK(NEW_LOCK)
7      !$OMP  END SINGLE COPYPRIVATE(NEW_LOCK)
8      END FUNCTION NEW_LOCK

```

9 Note that the effect of the **copyprivate** clause on a variable with the **allocatable**  
10 attribute is different than on a variable with the **pointer** attribute. The value of **A** is  
11 copied (as if by intrinsic assignment) and the pointer **B** is copied (as if by pointer  
12 assignment) to the corresponding list items in the other implicit tasks belonging to the  
13 parallel region.

*Example A.35.4f*

```

14      SUBROUTINE S(N)
15      INTEGER N

16      REAL, DIMENSION(:), ALLOCATABLE :: A
17      REAL, DIMENSION(:), POINTER :: B

18
19      ALLOCATE (A(N))
20      !$OMP  SINGLE
21          ALLOCATE (B(N))
22          READ (11) A,B
23      !$OMP  END SINGLE COPYPRIVATE(A,B)
24          ! Variable A is private and is
25          ! assigned the same value in each thread
26          ! Variable B is shared

27      !$OMP  BARRIER
28      !$OMP  SINGLE
29          DEALLOCATE (B)
30      !$OMP  END SINGLE NOWAIT
31      END SUBROUTINE S

```



---

## 1 A.36 Nested Loop Constructs

2 The following example of loop construct nesting (see Section 2.10 on page 109) is  
3 conforming because the inner and outer loop regions bind to different **parallel**  
4 regions:

▼ C/C++ ▼  
*Example A.36.1c*

```
5 void work(int i, int j) {}  
6  
7 void good_nesting(int n)  
8 {  
9     int i, j;  
10    #pragma omp parallel default(shared)  
11    {  
12        #pragma omp for  
13        for (i=0; i<n; i++) {  
14            #pragma omp parallel shared(i, n)  
15            {  
16                #pragma omp for  
17                for (j=0; j < n; j++)  
18                    work(i, j);  
19            }  
20        }  
21    }
```

▲ C/C++ ▲

*Example A.36.1f*

```
1          SUBROUTINE WORK(I, J)
2          INTEGER I, J
3          END SUBROUTINE WORK

4          SUBROUTINE GOOD_NESTING(N)
5          INTEGER N

6              INTEGER I
7          !$OMP PARALLEL DEFAULT(SHARED)
8          !$OMP DO
9              DO I = 1, N
10             !$OMP PARALLEL SHARED(I,N)
11             !$OMP DO
12                 DO J = 1, N
13                     CALL WORK(I,J)
14                 END DO
15             !$OMP END PARALLEL
16             END DO
17         !$OMP END PARALLEL
18     END SUBROUTINE GOOD_NESTING
```

1

The following variation of the preceding example is also conforming:

C/C++

*Example A.36.2c*

2

```
void work(int i, int j) {}
```

3

```
void work1(int i, int n)
```

4

```
{
```

5

```
    int j;
```

6

```
    #pragma omp parallel default(shared)
```

7

```
    {
```

8

```
        #pragma omp for
```

9

```
        for (j=0; j<n; j++)
```

10

```
            work(i, j);
```

11

```
    }
```

12

```
}
```

13

```
void good_nesting2(int n)
```

14

```
{
```

15

```
    int i;
```

16

```
    #pragma omp parallel default(shared)
```

17

```
    {
```

18

```
        #pragma omp for
```

19

```
        for (i=0; i<n; i++)
```

20

```
            work1(i, n);
```

21

```
    }
```

22

```
}
```

C/C++

*Example A.36.2f*

```

1          SUBROUTINE WORK(I, J)
2          INTEGER I, J
3          END SUBROUTINE WORK

4          SUBROUTINE WORK1(I, N)
5          INTEGER J
6          !$OMP PARALLEL DEFAULT(SHARED)
7          !$OMP DO
8              DO J = 1, N
9                  CALL WORK(I, J)
10             END DO
11         !$OMP END PARALLEL
12     END SUBROUTINE WORK1

13         SUBROUTINE GOOD_NESTING2(N)
14         INTEGER N
15         !$OMP PARALLEL DEFAULT(SHARED)
16         !$OMP DO
17             DO I = 1, N
18                 CALL WORK1(I, N)
19             END DO
20         !$OMP END PARALLEL
21     END SUBROUTINE GOOD_NESTING2

```

## 22 **A.37** Restrictions on Nesting of Regions

23 The examples in this section illustrate the region nesting rules. For more information on  
 24 region nesting, see Section 2.10 on page 109.

1 The following example is non-conforming because the inner and outer loop regions are  
2 closely nested:

C/C++

*Example A.37.1c*

```
3 void work(int i, int j) {}  
4  
5 void wrong1(int n)  
6 {  
7     #pragma omp parallel default(shared)  
8     {  
9         int i, j;  
10        #pragma omp for  
11        for (i=0; i<n; i++) {  
12            /* incorrect nesting of loop regions */  
13            #pragma omp for  
14            for (j=0; j<n; j++)  
15                work(i, j);  
16        }  
17    }  
}
```

C/C++

Fortran

*Example A.37.1f*

```
18 SUBROUTINE WORK(I, J)  
19 INTEGER I, J  
20 END SUBROUTINE WORK  
  
21 SUBROUTINE WRONG1(N)  
22 INTEGER N  
  
23 INTEGER I, J  
24 !$OMP PARALLEL DEFAULT(SHARED)  
25 !$OMP DO  
26 DO I = 1, N  
27 !$OMP DO ! incorrect nesting of loop regions  
28 DO J = 1, N  
29 CALL WORK(I, J)  
30 END DO  
31 END DO  
32 !$OMP END PARALLEL  
33 END SUBROUTINE WRONG1
```

Fortran

1 The following orphaned version of the preceding example is also non-conforming:

C/C++

*Example A.37.2c*

```
2 void work(int i, int j) {}
3 void work1(int i, int n)
4 {
5     int j;
6     /* incorrect nesting of loop regions */
7     #pragma omp for
8         for (j=0; j<n; j++)
9         work(i, j);
10 }
11
12 void wrong2(int n)
13 {
14     #pragma omp parallel default(shared)
15     {
16         int i;
17         #pragma omp for
18             for (i=0; i<n; i++)
19             work1(i, n);
20     }
```

C/C++

Fortran

*Example A.37.2f*

```
21 SUBROUTINE WORK1(I,N)
22     INTEGER I, N
23     INTEGER J
24     !$OMP DO ! incorrect nesting of loop regions
25         DO J = 1, N
26             CALL WORK(I,J)
27         END DO
28     END SUBROUTINE WORK1
29 SUBROUTINE WRONG2(N)
30     INTEGER N
31     INTEGER I
32     !$OMP PARALLEL DEFAULT(SHARED)
33     !$OMP DO
34         DO I = 1, N
35             CALL WORK1(I,N)
36         END DO
37     !$OMP END PARALLEL
38     END SUBROUTINE WRONG2
```

Fortran

1 The following example is non-conforming because the loop and **single** regions are  
2 closely nested:

C/C++

*Example A.37.3c*

```
3 void work(int i, int j) {}
4 void wrong3(int n)
5 {
6     #pragma omp parallel default(shared)
7     {
8         int i;
9         #pragma omp for
10        for (i=0; i<n; i++) {
11            /* incorrect nesting of regions */
12            #pragma omp single
13            work(i, 0);
14        }
15    }
16 }
```

C/C++

Fortran

*Example A.37.3f*

```
17 SUBROUTINE WRONG3(N)
18 INTEGER N
19
20 INTEGER I
21 !$OMP PARALLEL DEFAULT(SHARED)
22 !$OMP DO
23     DO I = 1, N
24         !$OMP SINGLE ! incorrect nesting of regions
25             CALL WORK(I, 1)
26         !$OMP END SINGLE
27     END DO
28 !$OMP END PARALLEL
END SUBROUTINE WRONG3
```

Fortran

1 The following example is non-conforming because a **barrier** region cannot be closely  
2 nested inside a loop region:

C/C++

*Example A.37.4c*

```
3 void work(int i, int j) {}  
4 void wrong4(int n)  
5 {  
  
6     #pragma omp parallel default(shared)  
7     {  
8         int i;  
9         #pragma omp for  
10        for (i=0; i<n; i++) {  
11            work(i, 0);  
12        /* incorrect nesting of barrier region in a loop region */  
13            #pragma omp barrier  
14            work(i, 1);  
15        }  
16    }  
17 }
```

C/C++

Fortran

*Example A.37.4f*

```
18 SUBROUTINE WRONG4(N)  
19 INTEGER N  
  
20 INTEGER I  
21 !$OMP PARALLEL DEFAULT(SHARED)  
22 !$OMP DO  
23 DO I = 1, N  
24 CALL WORK(I, 1)  
25 ! incorrect nesting of barrier region in a loop region  
26 !$OMP BARRIER  
27 CALL WORK(I, 2)  
28 END DO  
29 !$OMP END PARALLEL  
30 END SUBROUTINE WRONG4
```

Fortran



1 The following example is non-conforming because the **barrier** region cannot be  
2 closely nested inside the **critical** region. If this were permitted, it would result in  
3 deadlock due to the fact that only one thread at a time can enter the **critical** region:

C/C++

*Example A.37.5c*

```
4 void work(int i, int j) {}
5 void wrong5(int n)
6 {
7     #pragma omp parallel
8     {
9         #pragma omp critical
10        {
11            work(n, 0);
12        /* incorrect nesting of barrier region in a critical region */
13            #pragma omp barrier
14            work(n, 1);
15        }
16    }
17 }
```

C/C++

Fortran

*Example A.37.5f*

```
18 SUBROUTINE WRONG5(N)
19 INTEGER N
20 !$OMP PARALLEL DEFAULT(SHARED)
21 !$OMP CRITICAL
22 CALL WORK(N,1)
23 ! incorrect nesting of barrier region in a critical region
24 !$OMP BARRIER
25 CALL WORK(N,2)
26 !$OMP END CRITICAL
27 !$OMP END PARALLEL
28 END SUBROUTINE WRONG5
```

Fortran

1 The following example is non-conforming because the **barrier** region cannot be  
2 closely nested inside the **single** region. If this were permitted, it would result in  
3 deadlock due to the fact that only one thread executes the **single** region:

C/C++

*Example A.37.6c*

```
4 void work(int i, int j) {}
5 void wrong6(int n)
6 {
7     #pragma omp parallel
8     {
9         #pragma omp single
10        {
11            work(n, 0);
12        /* incorrect nesting of barrier region in a single region */
13            #pragma omp barrier
14            work(n, 1);
15        }
16    }
17 }
```

C/C++

Fortran

*Example A.37.6f*

```
18 SUBROUTINE WRONG6 (N)
19     INTEGER N
20
21     !$OMP PARALLEL DEFAULT(SHARED)
22     !$OMP SINGLE
23         CALL WORK(N,1)
24     ! incorrect nesting of barrier region in a single region
25     !$OMP BARRIER
26         CALL WORK(N,2)
27     !$OMP END SINGLE
28     !$OMP END PARALLEL
29     END SUBROUTINE WRONG6
```

Fortran

---

## 1 A.38 The `omp_set_dynamic` and 2 `omp_set_num_threads` Routines

3 Some programs rely on a fixed, prespecified number of threads to execute correctly.  
4 Because the default setting for the dynamic adjustment of the number of threads is  
5 implementation defined, such programs can choose to turn off the dynamic threads  
6 capability and set the number of threads explicitly to ensure portability. The following  
7 example shows how to do this using `omp_set_dynamic` (Section 3.2.7 on page 121),  
8 and `omp_set_num_threads` (Section 3.2.1 on page 114).

9 In this example, the program executes correctly only if it is executed by 16 threads. If  
10 the implementation is not capable of supporting 16 threads, the behavior of this example  
11 is implementation defined (See Algorithm 2.1 on page 36). Note that the number of  
12 threads executing a `parallel` region remains constant during the region, regardless of  
13 the dynamic threads setting. The dynamic threads mechanism determines the number of  
14 threads to use at the start of the `parallel` region and keeps it constant for the duration  
15 of the region.

C/C++

*Example A.38.1c*

```
16 #include <omp.h>
17 #include <stdlib.h>
18
19 void do_by_16(float *x, int iam, int ipoints) {}
20
21 void dynthreads(float *x, int npoints)
22 {
23     int iam, ipoints;
24
25     omp_set_dynamic(0);
26     omp_set_num_threads(16);
27
28     #pragma omp parallel shared(x, npoints) private(iam, ipoints)
29     {
30         if (omp_get_num_threads() != 16)
31             abort();
32
33         iam = omp_get_thread_num();
34         ipoints = npoints/16;
35         do_by_16(x, iam, ipoints);
36     }
37 }
```

C/C++

*Example A.38.1f*

```

1      SUBROUTINE DO_BY_16(X, IAM, IPOINITS)
2          REAL X(*)
3          INTEGER IAM, IPOINITS
4      END SUBROUTINE DO_BY_16

5      SUBROUTINE DYNTHREADS(X, NPOINITS)

6          INCLUDE "omp_lib.h"      ! or USE OMP_LIB

7          INTEGER NPOINITS
8          REAL X(NPOINITS)

9          INTEGER IAM, IPOINITS

10         CALL OMP_SET_DYNAMIC(.FALSE.)
11         CALL OMP_SET_NUM_THREADS(16)

12     !$OMP PARALLEL SHARED(X,NPOINITS) PRIVATE(IAM, IPOINITS)

13         IF (OMP_GET_NUM_THREADS() .NE. 16) THEN
14             STOP
15         ENDIF

16         IAM = OMP_GET_THREAD_NUM()
17         IPOINITS = NPOINITS/16
18         CALL DO_BY_16(X,IAM,IPOINITS)

19     !$OMP END PARALLEL

20     END SUBROUTINE DYNTHREADS

```

## 21 A.39 The `omp_get_num_threads` Routine

22 In the following example, the `omp_get_num_threads` call (see Section 3.2.2 on  
23 page 115) returns 1 in the sequential part of the code, so *np* will always be equal to 1.  
24 To determine the number of threads that will be deployed for the `parallel` region, the  
25 call should be inside the `parallel` region.

*Example A.39.1c*

```

1      #include <omp.h>
2      void work(int i);

3      void incorrect()
4      {
5          int np, i;

6          np = omp_get_num_threads(); /* misplaced */

7          #pragma omp parallel for schedule(static)
8          for (i=0; i < np; i++)
9              work(i);
10     }

```

*Example A.39.1f*

```

11         SUBROUTINE WORK(I)
12             INTEGER I
13             I = I + 1
14         END SUBROUTINE WORK

15         SUBROUTINE INCORRECT()
16             INCLUDE "omp_lib.h"      ! or USE OMP_LIB
17             INTEGER I, NP

18             NP = OMP_GET_NUM_THREADS() !misplaced: will return 1
19 !$OMP   PARALLEL DO SCHEDULE(STATIC)
20             DO I = 0, NP-1
21                 CALL WORK(I)
22             ENDDO
23 !$OMP   END PARALLEL DO
24         END SUBROUTINE INCORRECT

```

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

C/C++

*Example A.39.2c*

```
3     #include <omp.h>
4     void work(int i);
5
6     void correct()
7     {
8         int i;
9
10        #pragma omp parallel private(i)
11        {
12            i = omp_get_thread_num();
13            work(i);
14        }
15    }
```

C/C++

Fortran

*Example A.39.2f*

```
14         SUBROUTINE WORK(I)
15             INTEGER I
16
17             I = I + 1
18
19         END SUBROUTINE WORK
20
21         SUBROUTINE CORRECT()
22             INCLUDE "omp_lib.h"      ! or USE OMP_LIB
23             INTEGER I
24
25             !$OMP PARALLEL PRIVATE(I)
26                 I = OMP_GET_THREAD_NUM()
27                 CALL WORK(I)
28             !$OMP END PARALLEL
29
30         END SUBROUTINE CORRECT
```

Fortran

---

## 1 A.40 The `omp_init_lock` Routine

2 The following example demonstrates how to initialize an array of locks in a `parallel`  
3 region by using `omp_init_lock` (Section 3.3.1 on page 141).

C/C++

*Example A.40.1c*

```
4      #include <omp.h>
5
6      omp_lock_t *new_locks()
7      {
8          int i;
9          omp_lock_t *lock = new omp_lock_t[1000];
10
11         #pragma omp parallel for private(i)
12         for (i=0; i<1000; i++)
13             {
14                 omp_init_lock(&lock[i]);
15             }
16         return lock;
17     }
```

C/C++

Fortran

*Example A.40.1f*

```
16      FUNCTION NEW_LOCKS()
17          USE OMP_LIB          ! or INCLUDE "omp_lib.h"
18          INTEGER(OMP_LOCK_KIND), DIMENSION(1000) :: NEW_LOCKS
19
20          INTEGER I
21
22          !$OMP PARALLEL DO PRIVATE(I)
23              DO I=1,1000
24                  CALL OMP_INIT_LOCK(NEW_LOCKS(I))
25              END DO
26          !$OMP END PARALLEL DO
27
28          END FUNCTION NEW_LOCKS
```

Fortran

## 1 A.41 Ownership of Locks

2 Ownership of locks has changed since OpenMP 2.5. In OpenMP 2.5, locks are owned by  
3 threads; so a lock released by the `omp_unset_lock` routine must be owned by the  
4 same thread executing the routine. With OpenMP 3.0, locks are owned by task regions;  
5 so a lock released by the `omp_unset_lock` routine in a task region must be owned by  
6 the same task region.

7 This change in ownership requires extra care when using locks. The following program  
8 is conforming in OpenMP 2.5 because the thread that releases the lock `lck` in the parallel  
9 region is the same thread that acquired the lock in the sequential part of the program  
10 (master thread of parallel region and the initial thread are the same). However, it is not  
11 conforming in OpenMP 3.0 and 3.1, because the task region that releases the lock `lck` is  
12 different from the task region that acquires the lock.

C/C++

*Example A.41.1c*

```
13 #include <stdlib.h>
14 #include <stdio.h>
15 #include <omp.h>
16
17 int main()
18 {
19     int x;
20     omp_lock_t lck;
21
22     omp_init_lock (&lck);
23     omp_set_lock (&lck);
24     x = 0;
25
26 #pragma omp parallel shared (x)
27 {
28     #pragma omp master
29     {
30         x = x + 1;
31         omp_unset_lock (&lck);
32     }
33
34     /* Some more stuff. */
35
36     omp_destroy_lock (&lck);
37 }
```

C/C++



*Example A.41.1f*

```

1           program lock
2           use omp_lib
3           integer :: x
4           integer (kind=omp_lock_kind) :: lck

5           call omp_init_lock (lck)
6           call omp_set_lock(lck)
7           x = 0

8           !$omp parallel shared (x)
9           !$omp master
10          x = x + 1
11          call omp_unset_lock(lck)
12          !$omp end master

13
14          !      Some more stuff.
15          !$omp end parallel

16          call omp_destroy_lock(lck)
17          end

```

## 18 A.42 Simple Lock Routines

19 In the following example (for Section 3.3 on page 139), the lock routines cause the  
20 threads to be idle while waiting for entry to the first critical section, but to do other work  
21 while waiting for entry to the second. The `omp_set_lock` function blocks, but the  
22 `omp_test_lock` function does not, allowing the work in `skip` to be done.

1 Note that the argument to the lock routines should have type `omp_lock_t`, and that  
 2 there is no need to flush it.

### *Example A.42.1c*

```

3      #include <stdio.h>
4      #include <omp.h>

5      void skip(int i) {}
6      void work(int i) {}

7      int main()
8      {
9          omp_lock_t lck;
10         int id;

11         omp_init_lock(&lck);

12         #pragma omp parallel shared(lck) private(id)
13         {
14             id = omp_get_thread_num();

15             omp_set_lock(&lck);
16             /* only one thread at a time can execute this printf */
17             printf("My thread id is %d.\n", id);
18             omp_unset_lock(&lck);

19             while (! omp_test_lock(&lck)) {
20                 skip(id); /* we do not yet have the lock,
21                          so we must do something else */
22             }

23             work(id); /* we now have the lock
24                      and can do the work */

25             omp_unset_lock(&lck);
26         }

27         omp_destroy_lock(&lck);
28
29         return 0;
30     }

```

1 Note that there is no need to flush the lock variable.

*Example A.42.1f*

```

2          SUBROUTINE SKIP(ID)
3          END SUBROUTINE SKIP

4          SUBROUTINE WORK(ID)
5          END SUBROUTINE WORK

6          PROGRAM SIMPLELOCK

7              INCLUDE "omp_lib.h"      ! or USE OMP_LIB

8              INTEGER(OMP_LOCK_KIND) LCK
9              INTEGER ID

10             CALL OMP_INIT_LOCK(LCK)

11             !$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
12                 ID = OMP_GET_THREAD_NUM()
13                 CALL OMP_SET_LOCK(LCK)
14                 PRINT *, 'My thread id is ', ID
15                 CALL OMP_UNSET_LOCK(LCK)

16                 DO WHILE (.NOT. OMP_TEST_LOCK(LCK))
17                     CALL SKIP(ID)      ! We do not yet have the lock
18                                         ! so we must do something else
19                 END DO

20                 CALL WORK(ID)          ! We now have the lock
21                                         ! and can do the work

22                 CALL OMP_UNSET_LOCK( LCK )

23             !$OMP END PARALLEL

24             CALL OMP_DESTROY_LOCK( LCK )

25         END PROGRAM SIMPLELOCK

```

## 1 A.43 Nestable Lock Routines

2 The following example (for Section 3.3 on page 139) demonstrates how a nestable lock  
3 can be used to synchronize updates both to a whole structure and to one of its members.

Example A.43.1c C/C++

```
4      #include <omp.h>
5      typedef struct {
6          int a,b;
7          omp_nest_lock_t lck; } pair;

8      int work1();
9      int work2();
10     int work3();
11     void incr_a(pair *p, int a)
12     {
13         /* Called only from incr_pair, no need to lock. */
14         p->a += a;
15     }
16     void incr_b(pair *p, int b)
17     {
18         /* Called both from incr_pair and elsewhere, */
19         /* so need a nestable lock. */

20         omp_set_nest_lock(&p->lck);
21         p->b += b;
22         omp_unset_nest_lock(&p->lck);
23     }
24     void incr_pair(pair *p, int a, int b)
25     {
26         omp_set_nest_lock(&p->lck);
27         incr_a(p, a);
28         incr_b(p, b);
29         omp_unset_nest_lock(&p->lck);
30     }
31     void nestlock(pair *p)
32     {
33         #pragma omp parallel sections
34         {
35             #pragma omp section
36             incr_pair(p, work1(), work2());
37             #pragma omp section
38             incr_b(p, work3());
39         }
40     }
```

C/C++

*Example A.43.1f*

```

1      MODULE DATA
2          USE OMP_LIB, ONLY: OMP_NEST_LOCK_KIND
3          TYPE LOCKED_PAIR
4              INTEGER A
5              INTEGER B
6              INTEGER (OMP_NEST_LOCK_KIND) LCK
7          END TYPE
8      END MODULE DATA

9      SUBROUTINE INCR_A(P, A)
10         ! called only from INCR_PAIR, no need to lock
11         USE DATA
12         TYPE (LOCKED_PAIR) :: P
13         INTEGER A
14         P%A = P%A + A
15     END SUBROUTINE INCR_A

16     SUBROUTINE INCR_B(P, B)
17         ! called from both INCR_PAIR and elsewhere,
18         ! so we need a nestable lock
19         USE OMP_LIB      ! or INCLUDE "omp_lib.h"
20         USE DATA
21         TYPE (LOCKED_PAIR) :: P
22         INTEGER B
23         CALL OMP_SET_NEST_LOCK(P%LCK)
24         P%B = P%B + B
25         CALL OMP_UNSET_NEST_LOCK(P%LCK)
26     END SUBROUTINE INCR_B

27     SUBROUTINE INCR_PAIR(P, A, B)
28         USE OMP_LIB      ! or INCLUDE "omp_lib.h"
29         USE DATA
30         TYPE (LOCKED_PAIR) :: P
31         INTEGER A
32         INTEGER B

33         CALL OMP_SET_NEST_LOCK(P%LCK)
34         CALL INCR_A(P, A)
35         CALL INCR_B(P, B)
36         CALL OMP_UNSET_NEST_LOCK(P%LCK)
37     END SUBROUTINE INCR_PAIR

38     SUBROUTINE NESTLOCK(P)
39         USE OMP_LIB      ! or INCLUDE "omp_lib.h"
40         USE DATA
41         TYPE (LOCKED_PAIR) :: P
42         INTEGER WORK1, WORK2, WORK3
43         EXTERNAL WORK1, WORK2, WORK3

```

```
1      !$OMP  PARALLEL SECTIONS
2
3      !$OMP  SECTION
4          CALL INCR_PAIR(P, WORK1(), WORK2())
5      !$OMP  SECTION
6          CALL INCR_B(P, WORK3())
7      !$OMP  END PARALLEL SECTIONS
8
9      END SUBROUTINE NESTLOCK
```

Fortran



# Stubs for Runtime Library Routines

---

This section provides stubs for the runtime library routines defined in the OpenMP API. The stubs are provided to enable portability to platforms that do not support the OpenMP API. On these platforms, OpenMP programs must be linked with a library containing these stub routines. The stub routines assume that the directives in the OpenMP program are ignored. As such, they emulate serial semantics.

Note that the lock variable that appears in the lock routines must be accessed exclusively through these routines. It should not be initialized or otherwise modified in the user program.

In an actual implementation the lock variable might be used to hold the address of an allocated memory block, but here it is used to hold an integer value. Users should not make assumptions about mechanisms used by OpenMP implementations to implement locks based on the scheme used by the stub procedures.

---

## Fortran

**Note** – In order to be able to compile the Fortran stubs file, the include file `omp_lib.h` was split into two files: `omp_lib_kinds.h` and `omp_lib.h` and the `omp_lib_kinds.h` file included where needed. There is no requirement for the implementation to provide separate files.

---

## Fortran

---

## 1 B.1 C/C++ Stub Routines

```
2      #include <stdio.h>
3      #include <stdlib.h>
4      #include "omp.h"

5      void omp_set_num_threads(int num_threads)
6      {
7      }

8      int omp_get_num_threads(void)
9      {
10         return 1;
11     }

12     int omp_get_max_threads(void)
13     {
14         return 1;
15     }

16     int omp_get_thread_num(void)
17     {
18         return 0;
19     }

20     int omp_get_num_procs(void)
21     {
22         return 1;
23     }

24     int omp_in_parallel(void)
25     {
26         return 0;
27     }

28     void omp_set_dynamic(int dynamic_threads)
29     {
30     }

31     int omp_get_dynamic(void)
32     {
33         return 0;
34     }

35     void omp_set_nested(int nested)
36     {
37     }
```



```

1      int omp_get_nested(void)
2      {
3          return 0;
4      }

5      void omp_set_schedule(omp_sched_t kind, int modifier)
6      {
7      }

8      void omp_get_schedule(omp_sched_t *kind, int *modifier)
9      {
10         *kind = omp_sched_static;
11         *modifier = 0;
12     }

13     int omp_get_thread_limit(void)
14     {
15         return 1;
16     }

17     void omp_set_max_active_levels(int max_active_levels)
18     {
19     }

20     int omp_get_max_active_levels(void)
21     {
22         return 0;
23     }

24     int omp_get_level(void)
25     {
26         return 0;
27     }

28     int omp_get_ancestor_thread_num(int level)
29     {
30         if (level == 0)
31         {
32             return 0;
33         }
34         else
35         {
36             return -1;
37         }
38     }

```

```

1      int omp_get_team_size(int level)
2      {
3          if (level == 0)
4              {
5                  return 1;
6              }
7          else
8              {
9                  return -1;
10             }
11     }

12     int omp_get_active_level(void)
13     {
14         return 0;
15     }

16     struct __omp_lock
17     {
18         int lock;
19     };

20     enum { UNLOCKED = -1, INIT, LOCKED };

21     void omp_init_lock(omp_lock_t *arg)
22     {
23         struct __omp_lock *lock = (struct __omp_lock *)arg;
24         lock->lock = UNLOCKED;
25     }

26     void omp_destroy_lock(omp_lock_t *arg)
27     {
28         struct __omp_lock *lock = (struct __omp_lock *)arg;
29         lock->lock = INIT;
30     }

31     void omp_set_lock(omp_lock_t *arg)
32     {
33         struct __omp_lock *lock = (struct __omp_lock *)arg;
34         if (lock->lock == UNLOCKED)
35             {
36                 lock->lock = LOCKED;
37             }
38         else if (lock->lock == LOCKED)
39             {
40                 fprintf(stderr,
41                     "error: deadlock in using lock variable\n");
42                 exit(1);
43             }
44         else
45             {
46                 fprintf(stderr, "error: lock not initialized\n");
47                 exit(1);
48             }

```

```

1      }

2      void omp_unset_lock(omp_lock_t *arg)
3      {
4          struct __omp_lock *lock = (struct __omp_lock *)arg;
5          if (lock->lock == LOCKED)
6              {
7                  lock->lock = UNLOCKED;
8              }
9          else if (lock->lock == UNLOCKED)
10             {
11                 fprintf(stderr, "error: lock not set\n");
12                 exit(1);
13             }
14          else
15             {
16                 fprintf(stderr, "error: lock not initialized\n");
17                 exit(1);
18             }
19     }

20     int omp_test_lock(omp_lock_t *arg)
21     {
22         struct __omp_lock *lock = (struct __omp_lock *)arg;
23         if (lock->lock == UNLOCKED)
24             {
25                 lock->lock = LOCKED;
26                 return 1;
27             }
28         else if (lock->lock == LOCKED)
29             {
30                 return 0;
31             }
32         else
33             {
34                 fprintf(stderr, "error: lock not initialized\n");
35                 exit(1);
36             }
37     }

38     struct __omp_nest_lock
39     {
40         short owner;
41         short count;
42     };

43     enum { NOOWNER = -1, MASTER = 0 };

44     void omp_init_nest_lock(omp_nest_lock_t *arg)
45     {
46         struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
47         nlock->owner = NOOWNER;
48         nlock->count = 0;

```

```

1      }

2      void omp_destroy_nest_lock(omp_nest_lock_t *arg)
3      {
4          struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
5          nlock->owner = NOOWNER;
6          nlock->count = UNLOCKED;
7      }

8      void omp_set_nest_lock(omp_nest_lock_t *arg)
9      {
10         struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
11         if (nlock->owner == MASTER && nlock->count >= 1)
12         {
13             nlock->count++;
14         }
15         else if (nlock->owner == NOOWNER && nlock->count == 0)
16         {
17             nlock->owner = MASTER;
18             nlock->count = 1;
19         }
20         else
21         {
22             fprintf(stderr,
23                 "error: lock corrupted or not initialized\n");
24             exit(1);
25         }
26     }

27     void omp_unset_nest_lock(omp_nest_lock_t *arg)
28     {
29         struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
30         if (nlock->owner == MASTER && nlock->count >= 1)
31         {
32             nlock->count--;
33             if (nlock->count == 0)
34             {
35                 nlock->owner = NOOWNER;
36             }
37         }
38         else if (nlock->owner == NOOWNER && nlock->count == 0)
39         {
40             fprintf(stderr, "error: lock not set\n");
41             exit(1);
42         }
43         else
44         {
45             fprintf(stderr,
46                 "error: lock corrupted or not initialized\n");
47             exit(1);

```

```

1         }
2     }

3     int omp_test_nest_lock(omp_nest_lock_t *arg)
4     {
5         struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
6         omp_set_nest_lock(arg);
7         return nlock->count;
8     }

9     double omp_get_wtime(void)
10    {
11    /* This function does not provide a working
12     * wallclock timer. Replace it with a version
13     * customized for the target machine.
14     */
15     return 0.0;
16 }

17    double omp_get_wtick(void)
18    {
19    /* This function does not provide a working
20     * clock tick function. Replace it with
21     * a version customized for the target machine.
22     */
23     return 365. * 86400.;
24 }

```

---

## 1 B.2 Fortran Stub Routines

```
2      C23456
3          subroutine omp_set_num_threads(num_threads)
4              integer num_threads
5              return
6          end subroutine

7          integer function omp_get_num_threads()
8              omp_get_num_threads = 1
9              return
10         end function

11         integer function omp_get_max_threads()
12             omp_get_max_threads = 1
13             return
14         end function

15         integer function omp_get_thread_num()
16             omp_get_thread_num = 0
17             return
18         end function

19         integer function omp_get_num_procs()
20             omp_get_num_procs = 1
21             return
22         end function

23         logical function omp_in_parallel()
24             omp_in_parallel = .false.
25             return
26         end function

27         subroutine omp_set_dynamic(dynamic_threads)
28             logical dynamic_threads
29             return
30         end subroutine

31         logical function omp_get_dynamic()
32             omp_get_dynamic = .false.
33             return
34         end function

35         subroutine omp_set_nested(nested)
36             logical nested
37             return
38         end subroutine
```

```

1      logical function omp_get_nested()
2          omp_get_nested = .false.
3          return
4      end function

5      subroutine omp_set_schedule(kind, modifier)
6          include 'omp_lib_kinds.h'
7          integer (kind=omp_sched_kind) kind
8          integer modifier
9          return
10     end subroutine

11     subroutine omp_get_schedule(kind, modifier)
12         include 'omp_lib_kinds.h'
13         integer (kind=omp_sched_kind) kind
14         integer modifier

15         kind = omp_sched_static
16         modifier = 0
17         return
18     end subroutine

19     integer function omp_get_thread_limit()
20         omp_get_thread_limit = 1
21         return
22     end function

23     subroutine omp_set_max_active_levels( level )
24         integer level
25     end subroutine

26     integer function omp_get_max_active_levels()
27         omp_get_max_active_levels = 0
28         return
29     end function

30     integer function omp_get_level()
31         omp_get_level = 0
32         return
33     end function

34     integer function omp_get_ancestor_thread_num( level )
35         integer level
36         if ( level .eq. 0 ) then
37             omp_get_ancestor_thread_num = 0
38         else
39             omp_get_ancestor_thread_num = -1
40         end if
41         return
42     end function

```

```

1      integer function omp_get_team_size( level )
2          integer level
3          if ( level .eq. 0 ) then
4              omp_get_team_size = 1
5          else
6              omp_get_team_size = -1
7          end if
8          return
9      end function

10     integer function omp_get_active_level()
11         omp_get_active_level = 0
12         return
13     end function

14     subroutine omp_init_lock(lock)
15         ! lock is 0 if the simple lock is not initialized
16         !           -1 if the simple lock is initialized but not set
17         !           1 if the simple lock is set
18         include 'omp_lib_kinds.h'
19         integer(kind=omp_lock_kind) lock

20         lock = -1
21         return
22     end subroutine

23     subroutine omp_destroy_lock(lock)
24         include 'omp_lib_kinds.h'
25         integer(kind=omp_lock_kind) lock

26         lock = 0
27         return
28     end subroutine

29     subroutine omp_set_lock(lock)
30         include 'omp_lib_kinds.h'
31         integer(kind=omp_lock_kind) lock

32         if (lock .eq. -1) then
33             lock = 1
34         elseif (lock .eq. 1) then
35             print *, 'error: deadlock in using lock variable'
36             stop
37         else
38             print *, 'error: lock not initialized'
39             stop
40         endif

41         return
42     end subroutine

```



```

1      subroutine omp_unset_lock(lock)
2          include 'omp_lib_kinds.h'
3          integer(kind=omp_lock_kind) lock

4          if (lock .eq. 1) then
5              lock = -1
6          elseif (lock .eq. -1) then
7              print *, 'error: lock not set'
8              stop
9          else
10             print *, 'error: lock not initialized'
11             stop
12         endif

13         return
14     end subroutine

15     logical function omp_test_lock(lock)
16         include 'omp_lib_kinds.h'
17         integer(kind=omp_lock_kind) lock

18         if (lock .eq. -1) then
19             lock = 1
20             omp_test_lock = .true.
21         elseif (lock .eq. 1) then
22             omp_test_lock = .false.
23         else
24             print *, 'error: lock not initialized'
25             stop
26         endif

27         return
28     end function

29     subroutine omp_init_nest_lock(nlock)
30         ! nlock is
31         ! 0 if the nestable lock is not initialized
32         ! -1 if the nestable lock is initialized but not set
33         ! 1 if the nestable lock is set
34         ! no use count is maintained
35         include 'omp_lib_kinds.h'
36         integer(kind=omp_nest_lock_kind) nlock

37         nlock = -1

38         return
39     end subroutine

```

```

1      subroutine omp_destroy_nest_lock(nlock)
2          include 'omp_lib_kinds.h'
3          integer(kind=omp_nest_lock_kind) nlock

4          nlock = 0

5          return
6      end subroutine

7      subroutine omp_set_nest_lock(nlock)
8          include 'omp_lib_kinds.h'
9          integer(kind=omp_nest_lock_kind) nlock

10         if (nlock .eq. -1) then
11             nlock = 1
12         elseif (nlock .eq. 0) then
13             print *, 'error: nested lock not initialized'
14             stop
15         else
16             print *, 'error: deadlock using nested lock variable'
17             stop
18         endif

19         return
20     end subroutine

21     subroutine omp_unset_nest_lock(nlock)
22         include 'omp_lib_kinds.h'
23         integer(kind=omp_nest_lock_kind) nlock

24         if (nlock .eq. 1) then
25             nlock = -1
26         elseif (nlock .eq. 0) then
27             print *, 'error: nested lock not initialized'
28             stop
29         else
30             print *, 'error: nested lock not set'
31             stop
32         endif

33         return
34     end subroutine

```

```

1      integer function omp_test_nest_lock(nlock)
2          include 'omp_lib_kinds.h'
3          integer(kind=omp_nest_lock_kind) nlock

4          if (nlock .eq. -1) then
5              nlock = 1
6              omp_test_nest_lock = 1
7          elseif (nlock .eq. 1) then
8              omp_test_nest_lock = 0
9          else
10             print *, 'error: nested lock not initialized'
11             stop
12         endif

13         return
14     end function

15     double precision function omp_get_wtime()
16         ! this function does not provide a working
17         ! wall clock timer. replace it with a version
18         ! customized for the target machine.

19         omp_get_wtime = 0.0d0

20         return
21     end function

22     double precision function omp_get_wtick()
23         ! this function does not provide a working
24         ! clock tick function. replace it with
25         ! a version customized for the target machine.
26         double precision one_year
27         parameter (one_year=365.d0*86400.d0)

28         omp_get_wtick = one_year

29         return
30     end function

```



# OpenMP C and C++ Grammar

---

---

## 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         task-construct
12         master-construct
13         critical-construct
14         atomic-construct
15         ordered-construct
16     openmp-directive:
17         barrier-directive
18         taskwait-directive
19         taskyield-directive
20         flush-directive
21     structured-block:
22         statement
23     parallel-construct:
24         parallel-directive structured-block
25     parallel-directive:
26         # pragma omp parallel parallel-clauseoptseq new-line

```

```

1      parallel-clause:
2          unique-parallel-clause
3          data-default-clause
4          data-privatization-clause
5          data-privatization-in-clause
6          data-sharing-clause
7          data-reduction-clause
8      unique-parallel-clause:
9          if ( expression )
10         num_threads ( expression )
11         copyin ( variable-list )
12     for-construct:
13         for-directive iteration-statement
14     for-directive:
15         # pragma omp for for-clauseoptseq new-line
16     for-clause:
17         unique-for-clause
18         data-privatization-clause
19         data-privatization-in-clause
20         data-privatization-out-clause
21         data-reduction-clause
22         nowait
23     unique-for-clause:
24         ordered
25         schedule ( schedule-kind )
26         schedule ( schedule-kind , expression )
27         collapse ( expression )

```



```

1      schedule-kind:
2          static
3          dynamic
4          guided
5          auto
6          runtime
7      sections-construct:
8          sections-directive section-scope
9      sections-directive:
10         # pragma omp sections sections-clauseoptseq new-line
11     sections-clause:
12         data-privatization-clause
13         data-privatization-in-clause
14         data-privatization-out-clause
15         data-reduction-clause
16         nowait
17     section-scope:
18         { section-sequence }
19     section-sequence:
20         section-directiveopt structured-block
21         section-sequence section-directive structured-block
22     section-directive:
23         # pragma omp section new-line
24     single-construct:
25         single-directive structured-block
26     single-directive:
27         # pragma omp single single-clauseoptseq new-line

```

```

1      single-clause:
2          unique-single-clause
3          data-privatization-clause
4          data-privatization-in-clause
5          nowait
6      unique-single-clause:
7          copyprivate ( variable-list )
8      task-construct:
9          task-directive structured-block
10     task-directive:
11         # pragma omp task task-clauseoptseq new-line
12     task-clause:
13         unique-task-clause
14         data-default-clause
15         data-privatization-clause
16         data-privatization-in-clause
17         data-sharing-clause
18     unique-task-clause:
19         if ( scalar-expression )
20         untied
21     parallel-for-construct:
22         parallel-for-directive iteration-statement
23     parallel-for-directive:
24         # pragma omp parallel for parallel-for-clauseoptseq new-line
25     parallel-for-clause:
26         unique-parallel-clause
27         unique-for-clause

```

1            *data-default-clause*

2            *data-privatization-clause*

3            *data-privatization-in-clause*

4            *data-privatization-out-clause*

5            *data-sharing-clause*

6            *data-reduction-clause*

7            *parallel-sections-construct:*

8                *parallel-sections-directive section-scope*

9            *parallel-sections-directive:*

10           **# pragma omp parallel sections** *parallel-sections-clause*<sub>optseq</sub> *new-line*

11           *parallel-sections-clause:*

12               *unique-parallel-clause*

13               *data-default-clause*

14               *data-privatization-clause*

15               *data-privatization-in-clause*

16               *data-privatization-out-clause*

17               *data-sharing-clause*

18               *data-reduction-clause*

19           *master-construct:*

20               *master-directive structured-block*

21           *master-directive:*

22               **# pragma omp master** *new-line*

23           *critical-construct:*

24               *critical-directive structured-block*

25           *critical-directive:*

26               **# pragma omp critical** *region-phrase*<sub>opt</sub> *new-line*

27           *region-phrase:*

28               ( *identifier* )

```

1      barrier-directive:
2          # pragma omp barrier new-line
3      taskwait-directive:
4          # pragma omp taskwait new-line
5      taskyield-directive:
6          # pragma omp taskyield new-line
7      atomic-construct:
8          atomic-directive expression-statement
9          atomic-directive structured block
10     atomic-directive:
11         # pragma omp atomic atomic-clauseopt new-line
12     atomic-clause:
13         read
14         write
15         update
16         capture
17     flush-directive:
18         # pragma omp flush flush-varsopt 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

```

```

1      threadprivate-directive:
2          # pragma omp threadprivate ( variable-list ) new-line
3      data-default-clause:
4          default ( shared )
5          default ( none )
6      data-privatization-clause:
7          private ( variable-list )
8      data-privatization-in-clause:
9          firstprivate ( variable-list )
10     data-privatization-out-clause:
11     lastprivate ( variable-list )
12     data-sharing-clause:
13     shared ( variable-list )
14     data-reduction-clause:
15     reduction ( reduction-operator : variable-list )
16     reduction-operator:
17     One of: + * - & ^ | && ||
18     /* in C */
19     variable-list:
20     identifier
21     variable-list , identifier
22     /* in C++ */
23     variable-list:
24     id-expression
25     variable-list , id-expression

```



1

# D

2

## Interface Declarations

3

4

5

6

---

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. This is a non-normative section, implementation files may differ.

---

## 1 D.1 Example of the `omp.h` Header File

```
2     #ifndef _OMP_H_DEF
3     #define _OMP_H_DEF

4     /*
5     * define the lock data types
6     */
7     typedef void *omp_lock_t;

8     typedef void *omp_nest_lock_t;

9     /*
10    * define the schedule kinds
11    */
12    typedef enum omp_sched_t
13    {
14        omp_sched_static = 1,
15        omp_sched_dynamic = 2,
16        omp_sched_guided = 3,
17        omp_sched_auto = 4
18    } /* , Add vendor specific schedule constants here */ omp_sched_t;

19

20    /*
21    * exported OpenMP functions
22    */
23    #ifdef __cplusplus
24    extern "C"
25    {
26    #endif

27    extern void    omp_set_num_threads(int num_threads);
28    extern int    omp_get_num_threads(void);
29    extern int    omp_get_max_threads(void);
30    extern int    omp_get_thread_num(void);
31    extern int    omp_get_num_procs(void);
32    extern int    omp_in_parallel(void);
33    extern void    omp_set_dynamic(int dynamic_threads);
34    extern int    omp_get_dynamic(void);
35    extern void    omp_set_nested(int nested);
36    extern int    omp_get_nested(void);
37    extern int    omp_get_thread_limit(void);
38    extern void    omp_set_max_active_levels(int max_active_levels);
39    extern int    omp_get_max_active_levels(void);
40    extern int    omp_get_level(void);
41    extern int    omp_get_ancestor_thread_num(int level);
```



```

1      extern int      omp_get_team_size(int level);
2      extern int      omp_get_active_level(void);
3      extern void     omp_set_schedule(omp_sched_t kind, int modifier);
4      extern void     omp_get_schedule(omp_sched_t *kind, int *modifier);

5      extern void     omp_init_lock(omp_lock_t *lock);
6      extern void     omp_destroy_lock(omp_lock_t *lock);
7      extern void     omp_set_lock(omp_lock_t *lock);
8      extern void     omp_unset_lock(omp_lock_t *lock);
9      extern int      omp_test_lock(omp_lock_t *lock);

10     extern void     omp_init_nest_lock(omp_nest_lock_t *lock);
11     extern void     omp_destroy_nest_lock(omp_nest_lock_t *lock);
12     extern void     omp_set_nest_lock(omp_nest_lock_t *lock);
13     extern void     omp_unset_nest_lock(omp_nest_lock_t *lock);
14     extern int      omp_test_nest_lock(omp_nest_lock_t *lock);

15     extern double   omp_get_wtime(void);
16     extern double   omp_get_wtick(void);

17     #ifdef __cplusplus
18     }
19     #endif

20     #endif

```

---

## 1 D.2 Example of an Interface Declaration include 2 File

```
3 omp_lib_kinds.h:  
4     integer      omp_lock_kind  
5     integer      omp_nest_lock_kind  
6     ! this selects an integer that is large enough to hold a 64 bit integer  
7     parameter ( omp_lock_kind = selected_int_kind( 10 ) )  
8     parameter ( omp_nest_lock_kind = selected_int_kind( 10 ) )  
9     integer      omp_sched_kind  
10    ! this selects an integer that is large enough to hold a 32 bit integer  
11    parameter ( omp_sched_kind = selected_int_kind( 8 ) )  
12    integer ( omp_sched_kind ) omp_sched_static  
13    parameter ( omp_sched_static = 1 )  
14    integer ( omp_sched_kind ) omp_sched_dynamic  
15    parameter ( omp_sched_dynamic = 2 )  
16    integer ( omp_sched_kind ) omp_sched_guided  
17    parameter ( omp_sched_guided = 3 )  
18    integer ( omp_sched_kind ) omp_sched_auto  
19    parameter ( omp_sched_auto = 4 )  
  
20 omp_lib.h:  
21    C                                default integer type assumed below  
22    C                                default logical type assumed below  
23    C                                OpenMP Fortran API v3.1  
  
24    include 'omp_lib_kinds.h'  
25    integer      openmp_version  
26    parameter ( openmp_version = 201004 )  
  
27    external omp_set_num_threads  
28    external omp_get_num_threads  
29    integer      omp_get_num_threads  
30    external omp_get_max_threads  
31    integer      omp_get_max_threads  
32    external omp_get_thread_num  
33    integer      omp_get_thread_num  
34    external omp_get_num_procs  
35    integer      omp_get_num_procs  
36    external omp_in_parallel  
37    logical      omp_in_parallel  
38    external omp_set_dynamic  
39    external omp_get_dynamic  
40    logical      omp_get_dynamic  
41    external omp_set_nested  
42    external omp_get_nested  
43    logical      omp_get_nested  
44    external omp_set_schedule  
45    external omp_get_schedule  
46    external omp_get_thread_limit
```

```

1         integer omp_get_thread_limit
2         external omp_set_max_active_levels
3         external omp_get_max_active_levels
4         integer omp_get_max_active_levels
5         external omp_get_level
6         integer omp_get_level
7         external omp_get_ancestor_thread_num
8         integer omp_get_ancestor_thread_num
9         external omp_get_team_size
10        integer omp_get_team_size
11        external omp_get_active_level
12        integer omp_get_active_level

13        external omp_init_lock
14        external omp_destroy_lock
15        external omp_set_lock
16        external omp_unset_lock
17        external omp_test_lock
18        logical  omp_test_lock

19        external omp_init_nest_lock
20        external omp_destroy_nest_lock
21        external omp_set_nest_lock
22        external omp_unset_nest_lock
23        external omp_test_nest_lock
24        integer  omp_test_nest_lock

25        external omp_get_wtick
26        double precision  omp_get_wtick
27        external omp_get_wtime
28        double precision  omp_get_wtime

```

---

## 1 D.3 Example of a Fortran 90 Interface Declaration

2 module

3 ! the "!" of this comment starts in column 1  
4 !23456

```
5         module omp_lib_kinds
6             integer, parameter :: omp_lock_kind = selected_int_kind( 10 )
7             integer, parameter :: omp_nest_lock_kind = selected_int_kind( 10 )
8             integer, parameter :: omp_sched_kind = selected_int_kind( 8 )
9             integer(kind=omp_sched_kind), parameter ::
10            & omp_sched_static = 1
11            integer(kind=omp_sched_kind), parameter ::
12            & omp_sched_dynamic = 2
13            integer(kind=omp_sched_kind), parameter ::
14            & omp_sched_guided = 3
15            integer(kind=omp_sched_kind), parameter ::
16            & omp_sched_auto = 4
17            end module omp_lib_kinds
```

18 module omp\_lib

```
19         use omp_lib_kinds
20     !                                     OpenMP Fortran API v3.1
21         integer, parameter :: openmp_version = 201004
```

22 interface

```
23         subroutine omp_set_num_threads (number_of_threads_expr)
24             integer, intent(in) ::
25             & number_of_threads_expr
26         end subroutine omp_set_num_threads
```

```
27         function omp_get_num_threads ()
28             integer :: omp_get_num_threads
29         end function omp_get_num_threads
```

```
30         function omp_get_max_threads ()
31             integer :: omp_get_max_threads
32         end function omp_get_max_threads
```

```
33         function omp_get_thread_num ()
34             integer :: omp_get_thread_num
35         end function omp_get_thread_num
```

```
36         function omp_get_num_procs ()
37             integer :: omp_get_num_procs
38         end function omp_get_num_procs
```

```

1      function omp_in_parallel ()
2          logical :: omp_in_parallel
3      end function omp_in_parallel

4      subroutine omp_set_dynamic (enable_expr)
5          logical, intent(in) ::
6      &      enable_expr
7      end subroutine omp_set_dynamic

8      function omp_get_dynamic ()
9          logical :: omp_get_dynamic
10     end function omp_get_dynamic

11     subroutine omp_set_nested (enable_expr)
12         logical, intent(in) ::
13     &         enable_expr
14     end subroutine omp_set_nested

15     function omp_get_nested ()
16         logical :: omp_get_nested
17     end function omp_get_nested

18     subroutine omp_set_schedule (kind, modifier)
19         use omp_lib_kinds
20         integer(kind=omp_sched_kind), intent(in) :: kind
21         integer, intent(in) :: modifier
22     end subroutine omp_set_schedule

23     subroutine omp_get_schedule (kind, modifier)
24         use omp_lib_kinds
25         integer(kind=omp_sched_kind), intent(out) :: kind
26         integer, intent(out)::modifier
27     end subroutine omp_get_schedule

28     function omp_get_thread_limit()
29         integer :: omp_get_thread_limit
30     end function omp_get_thread_limit

31     subroutine omp_set_max_active_levels(var)
32         integer, intent(in) :: var
33     end subroutine omp_set_max_active_levels

34     function omp_get_max_active_levels()
35         integer ::
36     &         omp_get_max_active_levels
37     end function omp_get_max_active_levels

38     function omp_get_level()
39         integer :: omp_get_level
40     end function omp_get_level

41     function omp_get_ancestor_thread_num(level)
42         integer, intent(in) ::
43     &         level

```

```

1         integer ::
2         &         omp_get_ancestor_thread_num
3         end function omp_get_ancestor_thread_num

4         function omp_get_team_size(level)
5             integer, intent(in) ::
6             &         level
7             integer :: omp_get_team_size
8         end function omp_get_team_size

9         function omp_get_active_level()
10            integer ::
11            &         omp_get_active_level
12        end function omp_get_active_level

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

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

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

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

29        function omp_test_lock (var)
30            use omp_lib_kinds
31            logical :: omp_test_lock
32            integer (kind=omp_lock_kind), intent(inout) :: var
33        end function omp_test_lock

34        subroutine omp_init_nest_lock (var)
35            use omp_lib_kinds
36            integer (kind=omp_nest_lock_kind), intent(out) :: var
37        end subroutine omp_init_nest_lock

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

```

```

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

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

9      function omp_test_nest_lock (var)
10         use omp_lib_kinds
11         integer :: omp_test_nest_lock
12         integer (kind=omp_nest_lock_kind), intent(inout) ::
13     &         var
14     end function omp_test_nest_lock

15     function omp_get_wtick ()
16         double precision :: omp_get_wtick
17     end function omp_get_wtick

18     function omp_get_wtime ()
19         double precision :: omp_get_wtime
20     end function omp_get_wtime

21     end interface

22 end module omp_lib

```

---

## 1 D.4 Example of a Generic Interface for a Library 2 Routine

3 Any of the OMP runtime library routines that take an argument may be extended with a  
4 generic interface so arguments of different **KIND** type can be accommodated.

5 The **OMP\_SET\_NUM\_THREADS** interface could be specified in the **omp\_lib** module  
6 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=selected_int_kind( 8 ) ), 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=selected_int_kind( 3 ) ), intent(in) :: &
& number_of_threads_expr
  end subroutine omp_set_num_threads_2

end interface omp_set_num_threads
```



# E

## OpenMP Implementation-Defined Behaviors

This appendix summarizes the behaviors that are described as implementation defined in this API. Each behavior is cross-referenced back to its description in the main specification. An implementation is required to define and document its behavior in these cases.

- **Task scheduling points:** it is implementation defined where task scheduling points occur in untied task regions (see Section 1.3 on page 12).
- **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 13).
- **Internal control variables:** the initial values of *nthreads-var*, *dyn-var*, *run-sched-var*, *def-sched-var*, *stacksize-var*, *wait-policy-var*, *thread-limit-var*, and *max-active-levels-var* are implementation defined (see Section 2.3.2 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 Algorithm 2-1 even if dynamic adjustment is disabled (see Section 2.4.1 on page 36).
- **Loop directive:** the integer type or kind used to compute the iteration count of a collapsed loop is implementation defined. The effect of the `schedule(runtime)` clause when the *run-sched-var* ICV is set to `auto` is implementation defined. See Section 2.5.1 on page 38.
- **sections construct:** the method of scheduling the structured blocks among threads in the team is implementation defined (see Section 2.5.2 on page 47).
- **single construct:** the method of choosing a thread to execute the structured block is implementation defined (see Section 2.5.3 on page 49).

- 1 • **atomic construct**: a compliant implementation may enforce exclusive access  
2 between **atomic** regions which update different storage locations. The  
3 circumstances under which this occurs are implementation defined (see Section 2.8.5  
4 on page 71).
- 5 • **omp\_set\_num\_threads routine**: if the argument is not a positive integer the  
6 behavior is implementation defined (see Section 3.2.1 on page 114).
- 7 • **omp\_set\_schedule routine**: the behavior for implementation defined schedule  
8 types is implementation defined (see Section 3.2.11 on page 125).
- 9 • **omp\_set\_max\_active\_levels routine**: when called from within any explicit  
10 parallel region the binding thread set (and binding region, if required) for the  
11 **omp\_set\_max\_active\_levels** region is implementation defined and the  
12 behavior is implementation defined. If the argument is not a non-negative integer  
13 then the behavior is implementation defined (see Section 3.2.14 on page 130).
- 14 • **omp\_get\_max\_active\_levels routine**: when called from within any explicit  
15 parallel region the binding thread set (and binding region, if required) for the  
16 **omp\_get\_max\_active\_levels** region is implementation defined (see  
17 Section 3.2.15 on page 131).
- 18 • **OMP\_SCHEDULE environment variable**: if the value of the variable does not  
19 conform to the specified format then the result is implementation defined (see  
20 Section 4.1 on page 152).
- 21 • **OMP\_NUM\_THREADS environment variable**: if the value of the variable is greater  
22 than the number of threads the implementation can support or is not a positive integer  
23 then the result is implementation defined (see Section 4.2 on page 153).
- 24 • **OMP\_DYNAMIC environment variable**: if the value is neither **true** nor **false** the  
25 behavior is implementation defined (see Section 4.3 on page 154).
- 26 • **OMP\_NESTED environment variable**: if the value is neither **true** nor **false** the  
27 behavior is implementation defined (see Section 4.5 on page 155).
- 28 • **OMP\_STACKSIZE environment variable**: if the value does not conform to the  
29 specified format or the implementation cannot provide a stack of the specified size  
30 then the behavior is implementation defined (see Section 4.6 on page 155).
- 31 • **OMP\_WAIT\_POLICY environment variable**: the details of the **ACTIVE** and  
32 **PASSIVE** behaviors are implementation defined (see Section 4.7 on page 156).
- 33 • **OMP\_MAX\_ACTIVE\_LEVELS environment variable**: if the value is not a non-  
34 negative integer or is greater than the number of parallel levels an implementation  
35 can support then the behavior is implementation defined (see Section 4.8 on page  
36 157).
- 37 • **OMP\_THREAD\_LIMIT environment variable**: if the requested value is greater than  
38 the number of threads an implementation can support, or if the value is not a positive  
39 integer, the behavior of the program is implementation defined (see Section 4.9 on  
40 page 157).

- 1       • **threadprivate directive:** if the conditions for values of data in the threadprivate
- 2       objects of threads (other than the initial thread) to persist between two consecutive
- 3       active parallel regions do not all hold, the allocation status of an allocatable array in
- 4       the second region is implementation defined (see Section 2.9.2 on page 86).
- 5       • **shared clause:** passing a shared variable to a non-intrinsic procedure may result in
- 6       the value of the shared variable being copied into temporary storage before the
- 7       procedure reference, and back out of the temporary storage into the actual argument
- 8       storage after the procedure reference. Situations where this occurs other than those
- 9       specified are implementation defined (see Section 2.9.3.2 on page 93).
- 10      • **Runtime library definitions:** it is implementation defined whether the include file
- 11      **omp\_lib.h** or the module **omp\_lib** (or both) is provided. It is implementation
- 12      defined whether any of the OpenMP runtime library routines that take an argument
- 13      are extended with a generic interface so arguments of different **KIND** type can be
- 14      accommodated (see Section 3.1 on page 112).



# F

## Features History

---

This appendix summarizes the major changes between the OpenMP API Version 2.5 and Version 3.0, and between Version 3.0 and Version 3.1.

---

### F.1 Version 3.0 to 3.1 Differences

- The **final** and **mergeable** clauses (see Section 2.7.1 on page 59) were added to the **task** construct to support optimization of task data environments.
- The **taskyield** construct (see Section 2.7.2 on page 62) was added to allow user-defined task switching points.
- The **atomic** construct (see Section 2.8.5 on page 71) was extended to include **read**, **write**, and **capture** forms, and an **update** clause was added to apply the already existing form of the **atomic** construct.
- Data environment restrictions were changed to allow **intent(in)** and const-qualified types for the **firstprivate** clause (see Section 2.9.3.4 on page 97).
- New reduction operators **min** and **max** were added for C and C++ (see Section 2.9.3.6 on page 101 and page 104)
- The nesting restrictions in Section 2.10 on page 109 were clarified to disallow closely-nested OpenMP regions within an **atomic** region. This allows an **atomic** region to be consistently defined with other OpenMP regions so that they include all the code in the **atomic** construct.
- The **omp\_in\_final** runtime library routine (see Section 3.2.20 on page 138) was added to support specialization of final or included task regions.
- Descriptions of examples (see Appendix A on page 159) were expanded and clarified.
- Incorrect use of **omp\_integer\_kind** in Fortran interfaces (see Appendix D on page 325) was corrected.

---

## 1 F.2 Version 2.5 to 3.0 Differences

2 The concept of tasks has been added to the OpenMP execution model (see Section 1.2.3  
3 on page 8 and Section 1.3 on page 12).

- 4 • The **task** construct (see Section 2.7 on page 59) has been added, which provides a  
5 mechanism for creating tasks explicitly.
- 6 • The **taskwait** construct (see Section 2.8.4 on page 70) has been added, which  
7 causes a task to wait for all its child tasks to complete.
- 8 • The OpenMP memory model now covers atomicity of memory accesses (see  
9 Section 1.4.1 on page 13). The description of the behavior of **volatile** in terms of  
10 **flush** was removed.
- 11 • In Version 2.5, there was a single copy of of the *nest-var*, *dyn-var*, *nthreads-var* and  
12 *run-sched-var* internal control variables (ICVs) for the whole program. In Version  
13 3.0, there is one copy of these ICVs per task (see Section 2.3 on page 28). As a result,  
14 the **omp\_set\_num\_threads**, **omp\_set\_nested** and **omp\_set\_dynamic**  
15 runtime library routines now have specified effects when called from inside a  
16 **parallel** region (See Section 3.2.1 on page 114, Section 3.2.7 on page 121 and  
17 Section 3.2.9 on page 123).
- 18 • The definition of active **parallel** region has been changed: in Version 3.0 a  
19 **parallel** region is active if it is executed by a team consisting of more than one  
20 thread (see Section 1.2.2 on page 2).
- 21 • The rules for determining the number of threads used in a **parallel** region have  
22 been modified (see Section 2.4.1 on page 36).
- 23 • In Version 3.0, the assignment of iterations to threads in a loop construct with a  
24 **static** schedule kind is deterministic (see Section 2.5.1 on page 38).
- 25 • In Version 3.0, a loop construct may be associated with more than one perfectly  
26 nested loop. The number of associated loops may be controlled by the **collapse**  
27 clause (see Section 2.5.1 on page 38).
- 28 • Random access iterators, and variables of unsigned integer type, may now be used as  
29 loop iterators in loops associated with a loop construct (see Section 2.5.1 on page 38).
- 30 • The schedule kind **auto** has been added, which gives the implementation the  
31 freedom to choose any possible mapping of iterations in a loop construct to threads in  
32 the team (see Section 2.5.1 on page 38).
- 33 • Fortran assumed-size arrays now have predetermined data-sharing attributes (see  
34 Section 2.9.1.1 on page 83).
- 35 • In Fortran, **firstprivate** is now permitted as an argument to the **default**  
36 clause (see Section 2.9.3.1 on page 91).

- 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
- 41
- For list items in the **private** clause, implementations are no longer permitted to use the storage of the original list item to hold the new list item on the master thread. If no attempt is made to reference the original list item inside the **parallel** region, its value is well defined on exit from the **parallel** region (see Section 2.9.3.3 on page 94).
  - In Version 3.0, Fortran allocatable arrays may appear in **private**, **firstprivate**, **lastprivate**, **reduction**, **copyin** and **copyprivate** clauses. (see Section 2.9.2 on page 86, Section 2.9.3.3 on page 94, Section 2.9.3.4 on page 97, Section 2.9.3.5 on page 99, Section 2.9.3.6 on page 101, Section 2.9.4.1 on page 106 and Section 2.9.4.2 on page 107).
  - In Version 3.0, static class members variables may appear in a **threadprivate** directive (see Section 2.9.2 on page 86).
  - Version 3.0 makes clear where, and with which arguments, constructors and destructors of private and threadprivate class type variables are called (see Section 2.9.2 on page 86, Section 2.9.3.3 on page 94, Section 2.9.3.4 on page 97, Section 2.9.4.1 on page 106 and Section 2.9.4.2 on page 107)
  - The runtime library routines **omp\_set\_schedule** and **omp\_get\_schedule** have been added; these routines respectively set and retrieve the value of the *run\_sched\_var* ICV (see Section 3.2.11 on page 125 and Section 3.2.12 on page 127).
  - The *thread-limit-var* ICV has been added, which controls the maximum number of threads participating in the OpenMP program. The value of this ICV can be set with the **OMP\_THREAD\_LIMIT** environment variable and retrieved with the **omp\_get\_thread\_limit** runtime library routine (see Section 2.3.1 on page 28, Section 3.2.13 on page 129 and Section 4.9 on page 157).
  - The *max-active-levels-var* ICV has been added, which controls the number of nested active **parallel** regions. The value of this ICV can be set with the **OMP\_MAX\_ACTIVE\_LEVELS** environment variable and the **omp\_set\_max\_active\_levels** runtime library routine, and it can be retrieved with the **omp\_get\_max\_active\_levels** runtime library routine (see Section 2.3.1 on page 28, Section 3.2.14 on page 130, Section 3.2.15 on page 131 and Section 4.8 on page 157).
  - The *stacksize-var* ICV has been added, which controls the stack size for threads that the OpenMP implementation creates. The value of this ICV can be set with the **OMP\_STACKSIZE** environment variable (see Section 2.3.1 on page 28 and Section 4.6 on page 155).
  - The *wait-policy-var* ICV has been added, which controls the desired behavior of waiting threads. The value of this ICV can be set with the **OMP\_WAIT\_POLICY** environment variable (see Section 2.3.1 on page 28 and Section 4.7 on page 156).
  - The **omp\_get\_level** runtime library routine has been added, which returns the number of nested **parallel** regions enclosing the task that contains the call (see Section 3.2.16 on page 133).

- 1
- 2
- 3
- 4
- 5
- 6
- 7
- 8
- 9
- 10
- 11
- The `omp_get_ancestor_thread_num` runtime library routine has been added, which returns, for a given nested level of the current thread, the thread number of the ancestor (see Section 3.2.17 on page 134).
  - The `omp_get_team_size` runtime library routine has been added, which returns, for a given nested level of the current thread, the size of the thread team to which the ancestor belongs (see Section 3.2.18 on page 135).
  - The `omp_get_active_level` runtime library routine has been added, which returns the number of nested, active `parallel` regions enclosing the task that contains the call (see Section 3.2.19 on page 137).
  - In Version 3.0, locks are owned by tasks, not by threads (see Section 3.3 on page 139).



# Index

---

## Symbols

`_OPENMP` macro, 2-26

## A

`atomic`, 2-71  
attributes, data-sharing, 2-83  
`auto`, 2-43

## B

`barrier`, 2-68

## C

clauses

`copyin`, 2-106  
data-sharing, 2-90  
`default`, 2-91  
`firstprivate`, 2-97  
`lastprivate`, 2-99  
`private`, 2-94  
`reduction`, 2-101  
`schedule`, 2-42  
`shared`, 2-93

compliance, 1-17

conditional compilation, 2-26

constructs

`atomic`, 2-71  
`barrier`, 2-68  
`critical`, 2-67  
`do`, *Fortran*, 2-41  
`flush`, 2-76  
`for`, *C/C++*, 2-38  
`loop`, 2-38

`master`, 2-65  
`ordered`, 2-81  
`parallel`, 2-32  
`parallel for`, *C/C++*, 2-54  
`parallel sections`, 2-56  
`parallel workshare`, *Fortran*, 2-58  
`sections`, 2-47  
`single`, 2-49  
`task`, 2-59  
`taskwait`, 2-70  
`taskyield`, 2-62  
`workshare`, 2-51  
`worksharing`, 2-37

`copyin`, 2-106

`critical`, 2-67

## D

data sharing, 2-83

data-sharing clauses, 2-90

`default`, 2-91

directives, 2-21

format, 2-22

`threadprivate`, 2-86

see also constructs

`do`, *Fortran*, 2-41

`dynamic`, 2-43

## E

environment variables, 4-151

modifying ICV's, 2-29

`OMP_DYNAMIC`, 4-154

`OMP_MAX_ACTIVE_LEVELS`, 4-157

`OMP_NUM_THREADS`, 4-153

`OMP_SCHEDULE`, 4-152

`OMP_STACKSIZE`, 4-155

`OMP_THREAD_LIMIT`, 4-157

`OMP_WAIT_POLICY`, 4-156

execution model, 1-12

## F

`firstprivate`, 2-97

`flush`, 2-76

flush operation, 1-15

`for`, *C/C++*, 2-38

**G**  
glossary, 1-2  
grammar rules, C-302  
**guided**, 2-43

**H**  
header files, D-311

**I**  
ICVs (internal control variables), 2-28  
implementation, E-321  
**include** files, D-311  
internal control variables (ICVs), 2-28

**L**  
**lastprivate**, 2-99  
lock synchronization, 3-139  
loop, scheduling, 2-46

**M**  
**master**, 2-65  
memory model, 1-13  
model  
    execution, 1-12  
    memory, 1-13

**N**  
nesting, 2-109  
number of threads, 2-36

**O**  
**omp\_destroy\_lock**, 3-142  
**omp\_destroy\_nest\_lock**, 3-142  
**OMP\_DYNAMIC**, 4-154  
**omp\_get\_active\_level**, 3-137  
**omp\_get\_ancestor\_thread\_num**, 3-134  
**omp\_get\_dynamic**, 3-122  
**omp\_get\_levels**, 3-133  
**omp\_get\_max\_active\_levels**, 3-131  
**omp\_get\_max\_threads**, 3-116  
**omp\_get\_nested**, 3-124  
**omp\_get\_num\_procs**, 3-119  
**omp\_get\_num\_threads**, 3-115  
**omp\_get\_schedule**, 3-127  
**omp\_get\_team\_size**, 3-135  
**omp\_get\_thread\_limit**, 3-129  
**omp\_get\_thread\_num**, 3-117  
**omp\_get\_wtick**, 3-148  
**omp\_in\_parallel**, 3-120  
**omp\_init\_lock**, 3-141  
**omp\_init\_nest\_lock**, 3-141  
**OMP\_MAX\_ACTIVE\_LEVELS**, 4-157  
**OMP\_NUM\_THREADS**, 4-153  
**OMP\_SCHEDULE**, 4-152  
**omp\_set\_dynamic**, 3-121  
**omp\_set\_lock**, 3-143  
**omp\_set\_max\_active\_levels**, 3-130  
**omp\_set\_nest\_lock**, 3-143  
**omp\_set\_nested**, 3-123  
**omp\_set\_num\_threads**, 3-114  
**omp\_set\_schedule**, 3-125  
**OMP\_STACKSIZE**, 4-155  
**omp\_test\_lockomp\_test\_nest\_lock**, 3-145  
**omp\_tet\_wtime**, 3-147  
**OMP\_THREAD\_LIMIT**, 4-157  
**omp\_unset\_lock**, 3-144  
**omp\_unset\_nest\_lock**, 3-144  
**OMP\_WAIT\_POLICY**, 4-156  
OpenMP  
    compliance, 1-17  
    examples, A-159  
    features history, F-325  
    implementation, E-321  
**ordered**, 2-81

**P**  
**parallel**, 2-32  
**parallel for**, C/C++, 2-54  
**parallel sections**, 2-56  
**parallel workshare**, *Fortran*, 2-58  
pragmas  
    see constructs  
**private**, 2-94

**R**  
**reduction**, 2-101  
references, 1-17  
regions, nesting, 2-109

**runtime**, 2-44  
runtime library  
    interfaces and prototypes, 3-112

## **S**

**schedule**, 2-42  
scheduling  
    loop, 2-46  
    tasks, 2-63  
**sections**, 2-47  
**shared**, 2-93  
**single**, 2-49  
**static**, 2-43  
stubs  
    C/C++ routines, B-288  
    Fortran routines, B-294  
synchronization, 3-139

## **T**

task  
    scheduling, 2-63  
**task**, 2-59  
tasking, 2-59  
**taskwait**, 2-70  
**taskyield**, 2-62  
terminology, 1-2  
**threadprivate**, 2-86  
timer, 3-146

## **V**

variables, environment, 4-151

## **W**

wall clock timer, 3-146  
website  
    [www.openmp.org](http://www.openmp.org)  
**workshare**, 2-51  
worksharing  
    constructs, 2-37  
    scheduling, 2-46

