

1 OpenMP Fortran Application Program
2 Interface
3 Version 2.0, November 2000

Contents

		<i>Page</i>
5		
6	Introduction [1]	1
7	Scope	1
8	Glossary	1
9	Execution Model	3
10	Compliance	4
11	Organization	5
12	Directives [2]	7
13	OpenMP Directive Format	7
14	Directive Sentinels	8
15	Fixed Source Form Directive Sentinels	8
16	Free Source Form Directive Sentinel	8
17	Comments Inside Directives	10
18	Comments in Directives with Fixed Source Form	10
19	Comments in Directives with Free Source Form	10
20	Conditional Compilation	10
21	Fixed Source Form Conditional Compilation Sentinels	11
22	Free Source Form Conditional Compilation Sentinel	11
23	Parallel Region Construct	12
24	Work-sharing Constructs	15
25	DO Directive	15
26	SECTIONS Directive	18
27	SINGLE Directive	20
28	WORKSHARE Directive	20
29	Combined Parallel Work-sharing Constructs	22
30	PARALLEL DO Directive	23
31	PARALLEL SECTIONS Directive	24
32	PARALLEL WORKSHARE Directive	24
33	Synchronization Constructs and the MASTER Directive	25
34	MASTER Directive	25

	<i>Page</i>
35	35
36 CRITICAL Directive	26
37 BARRIER Directive	26
38 ATOMIC Directive	27
39 FLUSH Directive	29
40 ORDERED Directive	30
41 Data Environment Constructs	31
42 THREADPRIVATE Directive	32
43 Data Scope Attribute Clauses	34
44 PRIVATE Clause	35
45 SHARED Clause	36
46 DEFAULT Clause	36
47 FIRSTPRIVATE Clause	37
48 LASTPRIVATE Clause	38
49 REDUCTION Clause	38
50 COPYIN Clause	41
51 COPYPRIVATE Clause	41
52 Data Environment Rules	42
53 Directive Binding	45
54 Directive Nesting	45
55 Run-time Library Routines [3]	47
56 Execution Environment Routines	47
57 OMP_SET_NUM_THREADS Subroutine	48
58 OMP_GET_NUM_THREADS Function	48
59 OMP_GET_MAX_THREADS Function	49
60 OMP_GET_THREAD_NUM Function	49
61 OMP_GET_NUM_PROCS Function	50
62 OMP_IN_PARALLEL Function	50
63 OMP_SET_DYNAMIC Subroutine	51
64 OMP_GET_DYNAMIC Function	51
65 OMP_SET_NESTED Subroutine	52
66 OMP_GET_NESTED Function	52
67 Lock Routines	52
68 OMP_INIT_LOCK and OMP_INIT_NEST_LOCK Subroutines	54
69 OMP_DESTROY_LOCK and OMP_DESTROY_NEST_LOCK Subroutines	54

	<i>Page</i>
70	
71 OMP_SET_LOCK and OMP_SET_NEST_LOCK Subroutines	54
72 OMP_UNSET_LOCK and OMP_UNSET_NEST_LOCK Subroutines	55
73 OMP_TEST_LOCK and OMP_TEST_NEST_LOCK Functions	55
74 Timing Routines	56
75 OMP_GET_WTIME Function	56
76 OMP_GET_WTICK Function	57
77 Environment Variables [4]	59
78 OMP_SCHEDULE Environment Variable	59
79 OMP_NUM_THREADS Environment Variable	60
80 OMP_DYNAMIC Environment Variable	60
81 OMP_NESTED Environment Variable	61
82 Appendix A Examples	63
83 Executing a Simple Loop in Parallel	63
84 Specifying Conditional Compilation	63
85 Using Parallel Regions	64
86 Using the NOWAIT Clause	64
87 Using the CRITICAL Directive	64
88 Using the LASTPRIVATE Clause	65
89 Using the REDUCTION Clause	65
90 Specifying Parallel Sections	67
91 Using SINGLE Directives	67
92 Specifying Sequential Ordering	68
93 Specifying a Fixed Number of Threads	68
94 Using the ATOMIC Directive	69
95 Using the FLUSH Directive	69
96 Determining the Number of Threads Used	70
97 Using Locks	70
98 Using Nestable Locks	71
99 Nested DO Directives	73
100 Examples Showing Incorrect Nesting of Work-sharing Directives	74
101 Binding of BARRIER Directives	77
102 Scoping Variables with the PRIVATE Clause	78

	<i>Page</i>
103	
104 Examples of Noncompliant Storage Association	78
105 Examples of Syntax of Parallel DO Loops	81
106 Examples of the ATOMIC Directive	82
107 Examples of the ORDERED Directive	83
108 Examples of THREADPRIVATE Data	84
109 Examples of the Data Attribute Clauses: SHARED and PRIVATE	88
110 Examples of the Data Attribute Clause: COPYPRIVATE	89
111 Examples of the WORKSHARE Directive	91
112 Appendix B Stubs for Run-time Library Routines	95
113 Appendix C Using the SCHEDULE Clause	101
114 Appendix D Interface Declaration Module	105
115 Example of an Interface Declaration INCLUDE File	105
116 Example of a Fortran 90 Interface Declaration MODULE	107
117 Example of a Generic Interface for a Library Routine	111
118 Appendix E Implementation-Dependent Behaviors in OpenMP Fortran	113
119 Appendix F New Features in OpenMP Fortran version 2.0	115
120 Tables	
121 Table 1. SCHEDULE Clause Values	17
122 Table 2. Reduction Variable Initialization Values	40

123 Copyright © 1997-2000 OpenMP Architecture Review Board. Permission to copy
124 without fee all or part of this material is granted, provided the OpenMP Architecture
125 Review Board copyright notice and the title of this document appear. Notice is given
126 that copying is by permission of OpenMP Architecture Review Board.

128 This document specifies a collection of compiler directives, library routines, and
 129 environment variables that can be used to specify shared memory parallelism in
 130 Fortran programs. The functionality described in this document is collectively known
 131 as the *OpenMP Fortran Application Program Interface (API)*. The goal of this
 132 specification is to provide a model for parallel programming that is portable across
 133 shared memory architectures from different vendors. The OpenMP Fortran API is
 134 supported by compilers from numerous vendors. More information about OpenMP
 135 can be found at the following web site:

136 <http://www.openmp.org>

137 The directives, library routines, and environment variables defined in this document
 138 will allow users to create and manage parallel programs while ensuring portability.
 139 The directives extend the Fortran sequential programming model with
 140 single-program multiple data (SPMD) constructs, work-sharing constructs and
 141 synchronization constructs, and provide support for the sharing and privatization of
 142 data. The library routines and environment variables provide the functionality to
 143 control the run-time execution environment. The directive sentinels are structured so
 144 that the directives are treated as Fortran comments. Compilers that support the
 145 OpenMP Fortran API include a command line option that activates and allows
 146 interpretation of all OpenMP compiler directives.

147 1.1 Scope

148 This specification describes only user-directed parallelization, wherein the user
 149 explicitly specifies the actions to be taken by the compiler and run-time system in
 150 order to execute the program in parallel. OpenMP Fortran implementations are not
 151 required to check for dependencies, conflicts, deadlocks, race conditions, or other
 152 problems that result in incorrect program execution. The user is responsible for
 153 ensuring that the application using the OpenMP Fortran API constructs executes
 154 correctly.

155 Compiler-generated automatic parallelization is not addressed in this specification. █

156 1.2 Glossary

157 The following terms are used in this document: █

158 *defined* - For the contents of a data object, the property of having or being given a
159 valid value. For the allocation status or association status of a data object, the
160 property of having or being given a valid status.

161 *do-construct* - The Fortran Standard term for the construct that specifies the repeated
162 execution of a sequence of executable statements. The Fortran Standard calls such a
163 repeated sequence a *loop*. The loop that follows a DO or PARALLEL DO directive
164 cannot be a WHILE loop or a DO loop without loop control.

165 *implementation-dependent* - A behavior or value that is implementation-dependent is
166 permitted to vary among different OpenMP-compliant implementations (possibly in
167 response to limitations of hardware or operating system). Implementation-dependent
168 items are listed in Appendix E, page 113, and OpenMP-compliant implementations
169 are required to document how these items are handled.

170 *lexical extent* - Statements lexically contained within a structured block.

171 *master thread* - The thread that creates a team when a parallel region is entered.

172 *nested* - a parallel region is said to be nested if it appears within the dynamic extent
173 of a PARALLEL construct that (1) does not have an IF clause or (2) has an IF clause
174 and the logical expression within the clause evaluates to .TRUE..

175 *noncompliant* - Code structures or arrangements described as noncompliant are not
176 required to be supported by OpenMP-compliant implementations. Upon encountering
177 such structures, an OpenMP-compliant implementation may produce a compiler error.
178 Even if an implementation produces an executable for a program containing such
179 structures, its execution may terminate prematurely or have unpredictable behavior.

180 *parallel region* - Statements that bind to an OpenMP PARALLEL construct and are
181 available for execution by multiple threads.

182 *private* - Accessible to only one thread in the team for a parallel region. Note that
183 there are several ways to specify that a variable is private: use as a local variable in
184 a subprogram called from a parallel region, in a THREADPRIVATE directive, in a
185 PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION clause, or use of the variable
186 as a loop control variable.

187 *serialize* - When a parallel region is serialized, it is executed by a single thread. A
188 parallel region is said to be serialized if and only if at least one of the following are
189 true:

- 190 1. The logical expression in an IF clause attached to the parallel directive evaluates
191 to .FALSE.. .
- 192 2. It is a nested parallel region and nested parallelism is disabled.
- 193 3. It is a nested parallel region and the implementation chooses to serialize nested
194 parallel regions.

195 *serial region* - Statements that do not bind to an OpenMP PARALLEL construct. In
196 other words, these statements are executed by the master thread outside of a parallel
197 region.

198 *shared* - Accessible to all threads in the team for a parallel region.

199 *structured block* - A structured block is a collection of one or more executable
200 statements with a single point of entry at the top and a single point of exit at the
201 bottom. Execution must always proceed with entry at the top of the block and exit at
202 the bottom with only one exception: it is allowed to have a STOP statement inside a
203 structured block. This statement has the well defined behavior of terminating the
204 entire program.

205 *undefined* - For the contents of a data object, the property of not having a determinate
206 value. The result of a reference to a data object with undefined contents is
207 unspecified. For the allocation status or association status of a data object, the
208 property of not having a valid status. The behavior of an operation which relies upon
209 an undefined allocation status or association status is unspecified.

210 *unspecified* - A behavior or result that is unspecified is not constrained by
211 requirements in the OpenMP Fortran API. Possibly resulting from the misuse of a
212 language construct or other error, such a behavior or result may not be knowable prior
213 to the execution of a program, and may lead to premature termination of the program.

214 *variable* - A data object whose value can be defined and redefined during the
215 execution of a program. It may be a named data object, an array element, an array
216 section, a structure component, or a substring.

217 *white space* - A sequence of space or tab characters.

218 1.3 Execution Model

219 The OpenMP Fortran API uses the fork-join model of parallel execution. A program
220 that is written with the OpenMP Fortran API begins execution as a single process,
221 called the *master thread* of execution. The master thread executes sequentially until
222 the first parallel construct is encountered. In the OpenMP Fortran API, the
223 PARALLEL/END PARALLEL directive pair constitutes the parallel construct. When a
224 parallel construct is encountered, the master thread creates a *team* of threads, and
225 the master thread becomes the master of the team. The statements in the program
226 that are enclosed by the parallel construct, including routines called from within the
227 enclosed statements, are executed in parallel by each thread in the team. The
228 statements enclosed lexically within a construct define the *lexical* extent of the
229 construct. The *dynamic* extent further includes the routines called from within the
230 construct.

231 Upon completion of the parallel construct, the threads in the team synchronize and
232 only the master thread continues execution. Any number of parallel constructs can be
233 specified in a single program. As a result, a program may fork and join many times
234 during execution.

235 The OpenMP Fortran API allows programmers to use directives in routines called
236 from within parallel constructs. Directives that do not appear in the lexical extent of
237 the parallel construct but lie in the dynamic extent are called *orphaned* directives.
238 Orphaned directives allow users to execute major portions of their program in parallel
239 with only minimal changes to the sequential program. With this functionality, users
240 can code parallel constructs at the top levels of the program call tree and use
241 directives to control execution in any of the called routines.

242 1.4 Compliance

243 An implementation of the OpenMP Fortran API is *OpenMP-compliant* if it recognizes
244 and preserves the semantics of all the elements of this specification as laid out in
245 chapters 1, 2, 3, and 4. The appendixes are for information purposes only and are not
246 part of the specification.

247 The OpenMP Fortran API is an extension to the base language that is supported by
248 an implementation. If the base language does not support a language construct or
249 extension that appears in this document, the OpenMP implementation is not required
250 to support it.

251 All standard Fortran intrinsics and library routines and Fortran 90 `ALLOCATE` and
252 `DEALLOCATE` statements must be thread-safe in a compliant implementation.
253 Unsynchronized use of such intrinsics and routines by different threads in a parallel
254 region must produce correct results (though not necessarily the same as serial
255 execution results, as in the case of random number generation intrinsics, for example).

256 Unsynchronized use of Fortran output statements to the same unit may result in
257 output in which data written by different threads is interleaved. Similarly,
258 unsynchronized input statements from the same unit may read data in an interleaved
259 fashion. Unsynchronized use of Fortran I/O, such that each thread accesses a
260 different unit, produces the same results as serial execution of the I/O statements.

261 In both Fortran 90 and Fortran 95, a variable that has explicit initialization
262 implicitly has the `SAVE` attribute. This is not the case in FORTRAN 77. However, an
263 implementation of OpenMP Fortran must give such a variable the `SAVE` attribute,
264 regardless of the version of Fortran upon which it is based.

265 The OpenMP Fortran API specifies that certain behavior is
266 “implementation-dependent”. A conforming OpenMP implementation is required to

267 define and document its behavior in these cases. See Appendix E, page 113, for a list
268 of implementation-dependent behaviors.

269 **1.5 Organization**

270 The rest of this document is organized into the following chapters:

- 271 • Chapter 2, page 7, describes the compiler directives.
- 272 • Chapter 3, page 47, describes the run-time library routines.
- 273 • Chapter 4, page 59, describes the environment variables.
- 274 • Appendix A, page 63, contains examples.
- 275 • Appendix B, page 95, describes stub run-time library routines.
- 276 • Appendix C, page 101, has information about using the SCHEDULE clause.
- 277 • Appendix D, page 105, has examples of interfaces for the run-time library routines.
- 278 • Appendix E, page 113, describes implementation-dependent behaviors.
- 279 • Appendix F, page 115, describes the new features in the OpenMP Fortran v2.0 API.

280

281 Directives are special Fortran comments that are identified with a unique *sentinel*.
282 The directive sentinels are structured so that the directives are treated as Fortran
283 comments. Compilers that support the OpenMP Fortran API include a command line
284 option that activates and allows interpretation of all OpenMP compiler directives. In
285 the remainder of this document, the phrase *OpenMP compilation* is used to mean
286 that OpenMP directives are interpreted during compilation.

287 This chapter addresses the following topics:

- 288 • Section 2.1, page 7, describes the directive format.
- 289 • Section 2.2, page 12, describes the parallel region construct.
- 290 • Section 2.3, page 15, describes the work-sharing constructs.
- 291 • Section 2.4, page 22, describes the combined parallel work-sharing constructs.
- 292 • Section 2.5, page 25, describes the synchronization constructs and the MASTER
293 directive.
- 294 • Section 2.6, page 31, describes the data environment, which includes directives
295 and clauses that affect the data environment.
- 296 • Section 2.7, page 45, describes directive binding.
- 297 • Section 2.8, page 45, describes directive nesting.

298

2.1 OpenMP Directive Format

299

The format of an OpenMP directive is as follows:

300

sentinel directive_name [clause[,] clause] . . .]

301

302 All OpenMP compiler directives must begin with a directive *sentinel*. Directives are
303 case-insensitive. Clauses can appear in any order after the directive name. Clauses
304 on directives can be repeated as needed, subject to the restrictions listed in the
305 description of each clause. Directives cannot be embedded within continued
306 statements, and statements cannot be embedded within directives. Comments
preceded by an exclamation point may appear on the same line as a directive.

307

The following sections describe the OpenMP directive format:

- 308
- Section 2.1.1, page 8, describes directive sentinels.
 - Section 2.1.2, page 10, describes comments inside directives.
 - Section 2.1.3, page 10, describes conditional compilation.
- 309
- 310

311 **2.1.1 Directive Sentinels**

312 The directive sentinels accepted by an OpenMP-compliant compiler differ depending
313 on the Fortran source form being used. The !\$OMP sentinel is accepted when
314 compiling either fixed source form files or free source form files. The C\$OMP and
315 *\$OMP sentinels are accepted only when compiling fixed source form files.

316 The following sections contain more information on using the different sentinels.

317 **2.1.1.1 Fixed Source Form Directive Sentinels**

318 The OpenMP Fortran API accepts the following sentinels in fixed source form files:

319 !\$OMP | C\$OMP | *\$OMP

320 Sentinels must start in column one and appear as a single word with no intervening
321 white space (spaces and/or tab characters). Fortran fixed form line length, case
322 sensitivity, white space, continuation, and column rules apply to the directive line.
323 Initial directive lines must have a space or zero in column six, and continuation
324 directive lines must have a character other than a space or a zero in column six.

325 Example: The following formats for specifying directives are equivalent (the first line
326 represents the position of the first 9 columns):

327 C23456789
328 !\$OMP PARALLEL DO SHARED(A,B,C)

329 C\$OMP PARALLEL DO
330 C\$OMP+SHARED(A,B,C)

331 C\$OMP PARALLELDOSHARED(A,B,C)

332 **2.1.1.2 Free Source Form Directive Sentinel**

333 The OpenMP Fortran API accepts the following sentinel in free source form files:

334

```
! $OMP
```

335 The sentinel can appear in any column as long as it is preceded only by white space
336 (spaces and tab characters). It must appear as a single word with no intervening
337 white space. Fortran free form line length, case sensitivity, white space, and
338 continuation rules apply to the directive line. Initial directive lines must have a space
339 after the sentinel. Continued directive lines must have an ampersand as the last
340 nonblank character on the line, prior to any comment placed inside the directive.
341 Continuation directive lines can have an ampersand after the directive sentinel with
342 optional white space before and after the ampersand.

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

346 END CRITICAL
347 END DO
348 END MASTER
349 END ORDERED
350 END PARALLEL
351 END SECTIONS
352 END SINGLE
353 END WORKSHARE
354 PARALLEL DO
355 PARALLEL SECTIONS
356 PARALLEL WORKSHARE

357 Example: The following formats for specifying directives are equivalent (the first line
358 represents the position of the first 9 columns):

359 ! 23456789
360 ! \$OMP PARALLEL DO &
361 ! \$OMP SHARED(A,B,C)

362 ! \$OMP PARALLEL &
363 ! \$OMP&DO SHARED(A,B,C)

364 ! \$OMP PARALLELDO SHARED(A,B,C)

365 In order to simplify the presentation, the remainder of this document uses the !\$OMP
366 sentinel.

367 **2.1.2 Comments Inside Directives**368 The OpenMP Fortran API accepts comments placed inside directives. The rules
369 governing such comments depend on the Fortran source form being used.370 **2.1.2.1 Comments in Directives with Fixed Source Form**371 Comments may appear on the same line as a directive. The exclamation point
372 initiates a comment when it appears after column 6. The comment extends to the end
373 of the source line and is ignored. If the first nonblank character after the directive
374 sentinel of an initial or continuation directive line is an exclamation point, the line is
375 ignored.376 **2.1.2.2 Comments in Directives with Free Source Form**377 Comments may appear on the same line as a directive. The exclamation point
378 initiates a comment. The comment extends to the end of the source line and is
379 ignored. If the first nonblank character after the directive sentinel is an exclamation
380 point, the line is ignored.381 **2.1.3 Conditional Compilation**382 The OpenMP Fortran API permits Fortran lines to be compiled conditionally. The
383 directive sentinels for conditional compilation that are accepted by an
384 OpenMP-compliant compiler depend on the Fortran source form being used. The !\$
385 sentinel is accepted when compiling either fixed source form files or free source form
386 files. The C\$ and *\$ sentinels are accepted only when compiling fixed source form.387 During OpenMP compilation, the sentinel is replaced by two spaces, and the rest of
388 the line is treated as a normal Fortran line.389 If an OpenMP-compliant compiler supports a macro preprocessor (for example, `cpp`),
390 the Fortran processor must define the symbol `_OPENMP` to be used for conditional
391 compilation. This symbol is defined during OpenMP compilation to have the decimal
392 value `YYYYMM` where `YYYY` and `MM` are the year and month designations of the version
393 of the OpenMP Fortran API that the implementation supports.394 The following sections contain more information on using the different sentinels for
395 conditional compilation. (See Section A.2, page 63, for an example.)

396 *2.1.3.1 Fixed Source Form Conditional Compilation Sentinels*397 The OpenMP Fortran API accepts the following conditional compilation sentinels in
398 fixed source form files:

399 !\$ | C\$ | *\$ | c\$

400 The sentinel must start in column 1 and appear as a single word with no intervening
401 white space. Fortran fixed form line length, case sensitivity, white space,
402 continuation, and column rules apply to the line. After the sentinel is replaced with
403 two spaces, initial lines must have a space or zero in column 6 and only white space
404 and numbers in columns 1 through 5. After the sentinel is replaced with two spaces,
405 continuation lines must have a character other than a space or zero in column 6 and
406 only white space in columns 1 through 5. If these criteria are not met, the line is
407 treated as a comment and ignored.408 Example: The following forms for specifying conditional compilation in fixed source
409 form are equivalent:410 C23456789
411 !\$ 10 IAM = OMP_GET_THREAD_NUM() +
412 !\$ & INDEX

413 #ifdef _OPENMP
414 10 IAM = OMP_GET_THREAD_NUM() +
415 & INDEX
416 #endif417 *2.1.3.2 Free Source Form Conditional Compilation Sentinel*418 The OpenMP Fortran API accepts the following conditional compilation sentinel in
419 free source form files:

420 !\$

421 This sentinel can appear in any column as long as it is preceded only by white space.
422 It must appear as a single word with no intervening white space. Fortran free source
423 form line length, case sensitivity, white space, and continuation rules apply to the
424 line. Initial lines must have a space after the sentinel. Continued lines must have an
425 ampersand as the last nonblank character on the line, prior to any comment appearing
426 on the conditionally compiled line. Continuation lines can have an ampersand after
427 the sentinel, with optional white space before and after the ampersand.

428
429 Example: The following forms for specifying conditional compilation in free source
form are equivalent:

```
430 C23456789
431 !$ IAM = OMP_GET_THREAD_NUM( ) +      &
432      !$&      INDEX

433 #ifdef _OPENMP
434     IAM = OMP_GET_THREAD_NUM( ) +      &
435             INDEX
436 #endif
```

437 **2.2 Parallel Region Construct**

438 The `PARALLEL` and `END PARALLEL` directives define a *parallel region*. A parallel
439 region is a block of code that is to be executed by multiple threads in parallel. This is
440 the fundamental parallel construct in OpenMP that starts parallel execution. These
441 directives have the following format:

```
442 !$OMP PARALLEL [clause[, , clause]...]
443
444     block
445
446     !$OMP END PARALLEL
```

445 *clause* can be one of the following:

- 446 • `PRIVATE(list)`
- 447 • `SHARED(list)`
- 448 • `DEFAULT(PRIVATE | SHARED | NONE)`
- 449 • `FIRSTPRIVATE(list)`
- 450 • `REDUCTION({ operator | intrinsic_procedure_name } : list)`
- 451 • `COPYIN(list)`
- 452 • `IF(scalar_logical_expression)`
- 453 • `NUM_THREADS(scalar_integer_expression)`

454 The `IF` and `NUM_THREADS` clauses are described in this section. The `PRIVATE`,
455 `SHARED`, `DEFAULT`, `FIRSTPRIVATE`, `REDUCTION`, and `COPYIN` clauses are described in

456 Section 2.6.2, page 34. For an example of how to implement coarse-grain parallelism
457 using these directives, see Section A.3, page 64.

458 When a thread encounters a parallel region, it creates a team of threads, and it
459 becomes the master of the team. The master thread is a member of the team. The
460 number of threads in the team is controlled by environment variables, the
461 NUM_THREADS clause, and/or library calls. For more information on environment
462 variables, see Chapter 4, page 59. For more information on library routines, see
463 Chapter 3, page 47.

464 If a parallel region is encountered while dynamic adjustment of the number of
465 threads is disabled, and the number of threads specified for the parallel region
466 exceeds the number that the run-time system can supply, the behavior of the program
467 is implementation-dependent. An implementation may, for example, interrupt the
468 execution of the program, or it may serialize the parallel region.

469 The number of physical processors actually hosting the threads at any given time is
470 implementation-dependent. Once created, the number of threads in the team remains
471 constant for the duration of that parallel region. It can be changed either explicitly by
472 the user or automatically by the run-time system from one parallel region to another.
473 The `OMP_SET_DYNAMIC` library routine and the `OMP_DYNAMIC` environment variable
474 can be used to enable and disable the automatic adjustment of the number of threads.
475 For more information on the `OMP_SET_DYNAMIC` library routine, see Section 3.1.7,
476 page 51. For more information on the `OMP_DYNAMIC` environment variable, see
477 Section 4.3, page 60.

478 Within the dynamic extent of a parallel region, thread numbers uniquely identify
479 each thread. Thread numbers are consecutive whole numbers ranging from zero for
480 the master thread up to one less than the number of threads within the team. The
481 value of the thread number is returned by a call to the `OMP_GET_THREAD_NUM` library
482 routine (for more information see Section 3.1.4, page 49). If dynamic threads are
483 disabled when the parallel region is encountered, and remain disabled until a
484 subsequent, non-nested parallel region is encountered, then the thread numbers for
485 the two regions are consistent in that the thread identified with a given thread
486 number in the earlier parallel region will be identified with the same thread number
487 in the later region.

488 *block* denotes a structured block of Fortran statements. It is noncompliant to branch
489 into or out of the block. The code contained within the dynamic extent of the parallel
490 region is executed by each thread. The code path can be different for different threads.

491 The `END PARALLEL` directive denotes the end of the parallel region. There is an
492 implied barrier at this point. Only the master thread of the team continues execution
493 past the end of a parallel region.

494 If a thread in a team executing a parallel region encounters another parallel region, it
495 creates a new team, and it becomes the master of that new team. This second parallel
496 region is called a nested parallel region. By default, nested parallel regions are

497 serialized; that is, they are executed by a team composed of one thread. This default
498 behavior can be changed by using either the `OMP_SET_NESTED` library routine or the
499 `OMP_NESTED` environment variable. For more information on the `OMP_SET_NESTED`
500 library routine, see Section 3.1.9, page 52. For more information on the `OMP_NESTED`
501 environment variable, see Section 4.4, page 61.

502 If an `IF` clause is present, the enclosed code region is executed in parallel only if the
503 *scalar_logical_expression* evaluates to `.TRUE.`. Otherwise, the parallel region is
504 serialized. The expression must be a scalar Fortran logical expression. In the absence
505 of an `IF` clause, the region is executed as if an `IF(.TRUE.)` clause were specified.

506 The `NUM_THREADS` clause is used to request that a specific number of threads is used
507 in a parallel region. It supersedes the number of threads indicated by the
508 `OMP_SET_NUM_THREADS` library routine or the `OMP_NUM_THREADS` environment
509 variable for the parallel region it is applied to. Subsequent parallel regions, however,
510 are not affected unless they have their own `NUM_THREADS` clauses.
511 *scalar_integer_expression* must evaluate to a positive scalar integer value.

512 If execution of the program terminates while inside a parallel region, execution of all
513 threads terminates. All work before the previous barrier encountered by the threads
514 is guaranteed to be completed; none of the work after the next barrier that the
515 threads would have encountered will have been started. The amount of work done by
516 each thread in between the barriers and the order in which the threads terminate are
517 unspecified.

518 The following restrictions apply to parallel regions:

- 519 520 • The `PARALLEL/END PARALLEL` directive pair must appear in the same routine in
the executable section of the code.
- 521 522 • The code enclosed in a `PARALLEL/END PARALLEL` pair must be a structured block.
It is noncompliant to branch into or out of a parallel region.
- 523 524 • Only a single `IF` clause can appear on the directive. The `IF` expression is
evaluated outside the context of the parallel region. Results are unspecified if the
525 `IF` expression contains a function reference that has side effects.
- 526 527 • Only a single `NUM_THREADS` clause can appear on the directive. The `NUM_THREADS`
expression is evaluated outside the context of the parallel region. Results are
528 unspecified if the `NUM_THREADS` expression contains a function reference that has
529 side effects.
- 530 531 • If the dynamic threads mechanism is enabled, then the number of threads
requested by the `NUM_THREADS` clause is the maximum number to use in the
532 parallel region.
- 533 • The order of evaluation of `IF` clauses and `NUM_THREADS` clauses is unspecified.

- 534
- 535
- Unsynchronized use of Fortran I/O statements by multiple threads on the same unit has unspecified behavior.

536

2.3 Work-sharing Constructs

537 A work-sharing construct divides the execution of the enclosed code region among the
538 members of the team that encounter it. A work-sharing construct must be enclosed
539 dynamically within a parallel region in order for the directive to execute in parallel.
540 When a work-sharing construct is not enclosed dynamically within a parallel region,
541 it is treated as though the thread that encounters it were a team of size one. The
542 work-sharing directives do not launch new threads, and there is no implied barrier on
543 entry to a work-sharing construct.

544 The following restrictions apply to the work-sharing directives:

- 545
- 546
- Work-sharing constructs and BARRIER directives must be encountered by all threads in a team or by none at all.
 - Work-sharing constructs and BARRIER directives must be encountered in the same order by all threads in a team.

547 The following sections describe the work-sharing directives:

- 548
- Section 2.3.1, page 15, describes the DO and END DO directives.
 - Section 2.3.2, page 18, describes the SECTIONS, SECTION, and END SECTIONS directives.
 - Section 2.3.3, page 20, describes the SINGLE and END SINGLE directives.
 - Section 2.3.4, page 20, describes the WORKSHARE and END WORKSHARE directives.

549

550 If NOWAIT is specified on the END DO, END SECTIONS, END SINGLE, or
551 END WORKSHARE directive, an implementation may omit any code to synchronize the
552 threads at the end of the worksharing construct. In this case, threads that finish
553 early may proceed straight to the instructions following the work-sharing construct
554 without waiting for the other members of the team to finish the work-sharing
555 construct. (See Section A.4, page 64, for an example with the DO directive.)

561

2.3.1 DO Directive

562 The DO directive specifies that the iterations of the immediately following DO loop
563 must be executed in parallel. The loop that follows a DO directive cannot be a

564 DO WHILE or a DO loop without loop control. The iterations of the DO loop are
 565 distributed across threads that already exist.

566 The format of this directive is as follows:

```
567 !$OMP DO [clause[,] clause]...
568
569      do_loop
[ !$OMP END DO [NOWAIT]]
```

570 The *do_loop* may be a *do_construct*, an *outer_shared_do_construct*, or an
 571 *inner_shared_do_construct*. A DO construct that contains several DO statements that
 572 share the same DO termination statement syntactically consists of a sequence of
 573 *outer_shared_do_constructs*, followed by a single *inner_shared_do_construct*. If an END
 574 DO directive follows such a DO construct, a DO directive can only be specified for the
 575 first (i.e., the outermost) *outer_shared_do_construct*. (See examples in Section A.22,
 576 page 81.)

577 *clause* can be one of the following:

- 578 • PRIVATE(*list*)
- 579 • FIRSTPRIVATE(*list*)
- 580 • LASTPRIVATE(*list*)
- 581 • REDUCTION({ *operator* | *intrinsic_procedure_name* } : *list*)
- 582 • SCHEDULE(*type*[, *chunk*])
- 583 • ORDERED

584 The SCHEDULE and ORDERED clauses are described in this section. The PRIVATE,
 585 FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses are described in Section
 586 2.6.2, page 34.

587 If ordered sections are contained in the dynamic extent of the DO directive, the
 588 ORDERED clause must be present. For more information on ordered sections, see the
 589 ORDERED directive in Section 2.5.6, page 30.

590 The SCHEDULE clause specifies how iterations of the DO loop are divided among the
 591 threads of the team. *chunk* must be a scalar integer expression whose value is
 592 positive. The *chunk* expression is evaluated outside the context of the DO construct.
 593 Results are unspecified if the *chunk* expression contains a function reference that has
 594 side effects. Within the SCHEDULE(*type*[, *chunk*]) clause syntax, *type* can be one of
 595 the following:

596

Table 1. SCHEDULE Clause Values

597

type Effect

598

STATIC

When SCHEDULE(STATIC, *chunk*) is specified, iterations are divided into pieces of a size specified by *chunk*. The pieces are statically assigned to threads in the team in a round-robin fashion in the order of the thread number.

602

603

604

When *chunk* is not specified, the iteration space is divided into contiguous chunks that are approximately equal in size with one chunk assigned to each thread.

605

606

607

608

DYNAMIC

When SCHEDULE(DYNAMIC, *chunk*) is specified, the iterations are broken into pieces of a size specified by *chunk*. As each thread finishes a piece of the iteration space, it dynamically obtains the next set of iterations.

609

When no *chunk* is specified, it defaults to 1.

610

611

612

613

614

615

GUIDED

When SCHEDULE(GUIDED, *chunk*) is specified, the iteration space is divided into pieces such that the size of each successive piece is exponentially decreasing. *chunk* specifies the size of the smallest piece, except possibly the last. The size of the initial piece is implementation-dependent. As each thread finishes a piece of the iteration space, it dynamically obtains the next available piece.

616

When no *chunk* is specified, it defaults to 1.

617

618

619

620

621

622

623

RUNTIME

When SCHEDULE(RUNTIME) is specified, the decision regarding scheduling is deferred until run time. The schedule type and chunk size can be chosen at run time by setting the OMP_SCHEDULE environment variable. If this environment variable is not set, the resulting schedule is implementation-dependent. For more information on the OMP_SCHEDULE environment variable, see Section 4.1, page 59.

624

625

When SCHEDULE(RUNTIME) is specified, it is noncompliant to specify *chunk*.

626

627

628

629

630

631

In the absence of the SCHEDULE clause, the default schedule is implementation-dependent. An OpenMP-compliant program should not rely on a particular schedule for correct execution. Users should not rely on a particular implementation of a schedule type for correct execution, because it is possible to have variations in the implementations of the same schedule type across different compilers.

632

633

Threads that complete execution of their assigned loop iterations wait at a barrier at the END DO directive if the NOWAIT clause is not specified. The functionality of

634 NOWAIT is specified in Section 2.3, page 15. If an END DO directive is not specified, an
635 END DO directive is assumed at the end of the DO loop. If NOWAIT is specified on the
636 END DO directive, the implied FLUSH at the END DO directive is not performed. (See
637 Section A.4, page 64, for an example of using the NOWAIT clause. See Section 2.5.5,
638 page 29, for a description of implied FLUSH.)

639 Parallel DO loop control variables are block-level entities within the DO loop. If the
640 loop control variable also appears in the LASTPRIVATE list of the parallel DO, it is
641 copied out to a variable of the same name in the enclosing PARALLEL region. The
642 variable in the enclosing PARALLEL region must be SHARED if it is specified on the
643 LASTPRIVATE list of a DO directive.

644 The following restrictions apply to the DO directives:

- 645 • It is noncompliant to branch out of a DO loop associated with a DO directive.
- 646 • The values of the loop control parameters of the DO loop associated with a DO
647 directive must be the same for all the threads in the team.
- 648 • The DO loop iteration variable must be of type integer.
- 649 • If used, the END DO directive must appear immediately after the end of the loop.
- 650 • Only a single SCHEDULE clause can appear on a DO directive.
- 651 • Only a single ORDERED clause can appear on a DO directive.
- 652 • *chunk* must be a positive scalar integer expression.
- 653 • The value of the *chunk* parameter must be the same for all of the threads in the
654 team.

655 2.3.2 SECTIONS Directive

656 The SECTIONS directive is a non-iterative work-sharing construct that specifies that
657 the enclosed sections of code are to be divided among threads in the team. Each
658 section is executed once by a thread in the team.

659 The format of this directive is as follows:

```
660 !$OMP SECTIONS [clause[,] clause] . . .
661 [ !$OMP SECTION]
662 block
663 [ !$OMP SECTION
664 block]
665 . . .
666 !$OMP END SECTIONS [NOWAIT]
```

667 *block* denotes a structured block of Fortran statements.

668 *clause* can be one of the following:

- 669 • PRIVATE(*list*)
- 670 • FIRSTPRIVATE(*list*)
- 671 • LASTPRIVATE(*list*)
- 672 • REDUCTION({ *operator* | *intrinsic_procedure_name* } : *list*)

673 The PRIVATE, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses are described
674 in Section 2.6.2, page 34.

675 Each section is preceded by a SECTION directive, though the SECTION directive is
676 optional for the first section. The SECTION directives must appear within the lexical
677 extent of the SECTIONS/END SECTIONS directive pair. The last section ends at the
678 END SECTIONS directive. Threads that complete execution of their sections wait at a
679 barrier at the END SECTIONS directive if the NOWAIT clause is not specified. The
680 functionality of NOWAIT is described in Section 2.3, page 15.

681 The following restrictions apply to the SECTIONS directive:

- 682 • The code enclosed in a SECTIONS/END SECTIONS directive pair must be a
683 structured block. In addition, each constituent section must also be a structured
684 block. It is noncompliant to branch into or out of the constituent section blocks.
- 685 • It is noncompliant for a SECTION directive to be outside the lexical extent of the
686 SECTIONS/END SECTIONS directive pair. (See Section A.8, page 67 for an example
687 that uses these directives.)

688 **2.3.3 SINGLE Directive**

689 The `SINGLE` directive specifies that the enclosed code is to be executed by only one
690 thread in the team. Threads in the team that are not executing the `SINGLE` directive
691 wait at a barrier at the `END SINGLE` directive if the `NOWAIT` clause is not specified.
692 The functionality of `NOWAIT` is described in Section 2.3, page 15.

693 The format of this directive is as follows:

```
694 !$OMP SINGLE [clause[,] clause] . . .
695     block
696     !$OMP END SINGLE [end_single_modifier]
```

697 where `end_single_modifier` is either `COPYPRIVATE(list) [[,] COPYPRIVATE(list) . . .]`
698 or `NOWAIT`.

699 `block` denotes a structured block of Fortran statements.

700 `clause` can be one of the following:

- 701 • `PRIVATE(list)`
- 702 • `FIRSTPRIVATE(list)`

703 The `PRIVATE`, `FIRSTPRIVATE`, and `COPYPRIVATE` clauses are described in Section
704 2.6.2, page 34.

705 The following restriction applies to the `SINGLE` directive:

- 706 • The code enclosed in a `SINGLE/END SINGLE` directive pair must be a structured
707 block. It is noncompliant to branch into or out of the block.

708 See Section A.9, page 67, for an example of the `SINGLE` directive.

709 The following restriction applies to the `END SINGLE` directive:

- 710 • Specification of both a `COPYPRIVATE` clause and a `NOWAIT` clause on the same
711 `END SINGLE` directive is noncompliant.

712 **2.3.4 WORKSHARE Directive**

713 The `WORKSHARE` directive divides the work of executing the enclosed code into separate
714 units of work, and causes the threads of the team to share the work of executing the
715 enclosed code such that each unit is executed only once. The units of work may be
716 assigned to threads in any manner as long as each unit is executed exactly once.

```
717 ! $OMP WORKSHARE  
718   block  
719 ! $OMP END WORKSHARE [NOWAIT]
```

720 A BARRIER is implied following the enclosed code if the NOWAIT clause is not specified
721 on the END WORKSHARE directive. The functionality of NOWAIT is described in Section
722 2.3, page 15. An implementation of the WORKSHARE directive must insert any
723 synchronization that is required to maintain standard Fortran semantics. For
724 example, the effects of one statement within *block* must appear to occur before the
725 execution of succeeding statements, and the evaluation of the right hand side of an
726 assignment must appear to have been completed prior to the effects of assigning to
727 the left hand side.

728 The statements in *block* are divided into units of work as follows:

- 729 • For array expressions within each statement, including transformational array
730 intrinsic functions that compute scalar values from arrays:
 - 731 – Evaluation of each element of the array expression is a unit of work.
 - 732 – Evaluation of transformational array intrinsic functions may be freely
733 subdivided into any number of units of work.
- 734 • If a WORKSHARE directive is applied to an array assignment statement, the
735 assignment of each element is a unit of work.
- 736 • If a WORKSHARE directive is applied to a scalar assignment statement, the
737 assignment operation is a single unit of work.
- 738 • If a WORKSHARE directive is applied to a reference to an elemental function,
739 application of the function to the corresponding elements of any array argument is
740 treated as a unit of work. Hence, if any actual argument in a reference to an
741 elemental function is an array, the reference is treated in the same way as if the
742 function had been applied separately to corresponding elements of each array
743 actual argument.
- 744 • If a WORKSHARE directive is applied to a WHERE statement or construct, the
745 evaluation of the mask expression and the masked assignments are workshared.
- 746 • If a WORKSHARE directive is applied to a FORALL statement or construct, the
747 evaluation of the mask expression, expressions occurring in the specification of the
748 iteration space, and the masked assignments are workshared.
- 749 • For ATOMIC directives and their corresponding assignments, the update of each
750 scalar variable is a single unit of work.
- 751 • For CRITICAL constructs, each construct is a single unit of work.

- 752 • For **PARALLEL** constructs, each construct is a single unit of work with respect to
753 the **WORKSHARE** construct. The statements contained in **PARALLEL** constructs are
754 executed by new teams of threads formed for each **PARALLEL** directive.
- 755 • If none of the rules above apply to a portion of a statement in *block*, then that
756 portion is a single unit of work.

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

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

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

765 The **WORKSHARE** directive causes the sharing of work to occur only in the lexically
766 enclosed block.

767 The following restrictions apply to the **WORKSHARE** directive:

- 768 • *block* may contain statements which bind to lexically enclosed **PARALLEL**
769 constructs. Statements in these **PARALLEL** constructs are not restricted.
- 770 • *block* may contain **ATOMIC** directives and **CRITICAL** constructs.
- 771 • *block* must only contain array assignment statements, scalar assignment
772 statements, **FORALL** statements, **FORALL** constructs, **WHERE** statements, or **WHERE**
773 constructs.
- 774 • *block* must not contain any user defined function calls unless the function is
775 **ELEMENTAL**.
- 776 • The code enclosed in a **WORKSHARE/END WORKSHARE** directive pair must be a
777 structured block. It is noncompliant to branch into or out of the block.

778 2.4 Combined Parallel Work-sharing Constructs

779 The combined parallel work-sharing constructs are shortcuts for specifying a parallel
780 region that contains only one work-sharing construct. The semantics of these
781 directives are identical to that of explicitly specifying a **PARALLEL** directive followed
782 by a single work-sharing construct.

783 The following sections describe the combined parallel work-sharing directives:

- 784 • Section 2.4.1, page 23, describes the PARALLEL DO and END PARALLEL DO
785 directives.
- 786 • Section 2.4.2, page 24, describes the PARALLEL SECTIONS and
787 END PARALLEL SECTIONS directives.
- 788 • Section 2.4.3, page 24, describes the PARALLEL WORKSHARE and
789 END PARALLEL WORKSHARE directives.

790 **2.4.1 PARALLEL DO Directive**

791 The PARALLEL DO directive provides a shortcut form for specifying a parallel region
792 that contains a single DO directive. (See Section A.1, page 63, for an example.)

793 The format of this directive is as follows:

```
794     !$OMP PARALLEL DO [clause[,] clause]...  
795     do_loop  
796     [ !$OMP END PARALLEL DO ]
```

797 The *do_loop* may be a *do_construct*, an *outer_shared_do_construct*, or an
798 *inner_shared_do_construct*. A DO construct that contains several DO statements that
799 share the same DO termination statement syntactically consists of a sequence of
800 *outer_shared_do_constructs*, followed by a single *inner_shared_do_construct*. If an END
801 PARALLEL DO directive follows such a DO construct, a PARALLEL DO directive can
802 only be specified for the first (i.e., the outermost) *outer_shared_do_construct*. (See
803 Section A.22, page 81, for examples.)

804 *clause* can be one of the clauses accepted by either the PARALLEL or the DO directive.
805 For more information about the PARALLEL directive and the IF and NUM_THREADS
806 clauses, see Section 2.2, page 12. For more information about the DO directive and the
807 SCHEDULE and ORDERED clauses, see Section 2.3.1, page 15. For more information on
808 the remaining clauses, see Section 2.6.2, page 34.

809 If the END PARALLEL DO directive is not specified, the PARALLEL DO ends with the
810 DO loop that immediately follows the PARALLEL DO directive. If used, the
811 END PARALLEL DO directive must appear immediately after the end of the DO loop.

812 The semantics are identical to explicitly specifying a PARALLEL directive immediately
813 followed by a DO directive.

814 **2.4.2 PARALLEL SECTIONS Directive**

815 The PARALLEL SECTIONS directive provides a shortcut form for specifying a parallel
816 region that contains a single SECTIONS directive. The semantics are identical to
817 explicitly specifying a PARALLEL directive immediately followed by a SECTIONS
818 directive.

819 The format of this directive is as follows:

```
820 !$OMP PARALLEL SECTIONS [clause[,] clause] . . .
821 [ !$OMP SECTION ]
822 block
823 [ !$OMP SECTION
824 block]
825 . . .
826 !$OMP END PARALLEL SECTIONS
```

827 *block* denotes a structured block of Fortran statements.

828 *clause* can be one of the clauses accepted by either the PARALLEL or the SECTIONS
829 directive. For more information about the PARALLEL directive and the IF and
830 NUM_THREADS clauses, see Section 2.2, page 12. For more information about the
831 SECTIONS directive, see Section 2.3.2, page 18. For more information on the
832 remaining clauses, see Section 2.6.2, page 34.

833 The last section ends at the END PARALLEL SECTIONS directive.

834 **2.4.3 PARALLEL WORKSHARE Directive**

835 The PARALLEL WORKSHARE directive provides a shortcut form for specifying a
836 parallel region that contains a single WORKSHARE directive. The semantics are
837 identical to explicitly specifying a PARALLEL directive immediately followed by a
838 WORKSHARE directive.

839 The format of this directive is as follows:

```
840 ! $OMP PARALLEL WORKSHARE [clause[,] clause] . . .
841   block
842   ! $OMP END PARALLEL WORKSHARE
```

843 *block* denotes a structured block of Fortran statements.

844 *clause* can be one of the clauses accepted by either the PARALLEL or the WORKSHARE
845 directive. For more information about the PARALLEL directive and the IF and
846 NUM_THREADS clauses, see Section 2.2, page 12. For more information about the
847 remaining clauses, see Section 2.3.4, page 20.

848 2.5 Synchronization Constructs and the **MASTER** Directive

849 The following sections describe the synchronization constructs and the **MASTER**
850 directive:

- 851 • Section 2.5.1, page 25, describes the **MASTER** and **END MASTER** directives.
- 852 • Section 2.5.2, page 26, describes the **CRITICAL** and **END CRITICAL** directives.
- 853 • Section 2.5.3, page 26, describes the **BARRIER** directive.
- 854 • Section 2.5.4, page 27, describes the **ATOMIC** directive.
- 855 • Section 2.5.5, page 29, describes the **FLUSH** directive.
- 856 • Section 2.5.6, page 30, describes the **ORDERED** and **END ORDERED** directives.

857 2.5.1 **MASTER** Directive

858 The code enclosed within **MASTER** and **END MASTER** directives is executed by the
859 master thread of the team.

860 The format of this directive is as follows:

```
861 ! $OMP MASTER
862   block
863   ! $OMP END MASTER
```

864 The other threads in the team skip the enclosed section of code and continue
865 execution. There is no implied barrier either on entry to or exit from the master
866 section.

867 The following restriction applies to the MASTER directive:

- 868 • The code enclosed in a MASTER/ END MASTER directive pair must be a structured
869 block. It is noncompliant to branch into or out of the block.

870 **2.5.2 CRITICAL Directive**

871 The CRITICAL and END CRITICAL directives restrict access to the enclosed code to
872 only one thread at a time.

873 The format of this directive is as follows:

```
874        !$OMP CRITICAL [(name)]  
875            block  
876        !$OMP END CRITICAL [(name)]
```

877 The optional *name* argument identifies the critical section.

878 A thread waits at the beginning of a critical section until no other thread is executing
879 a critical section with the same name. All unnamed CRITICAL directives map to the
880 same name. Critical section names are global entities of the program. If a name
881 conflicts with any other entity, the behavior of the program is unspecified.

882 The following restrictions apply to the CRITICAL directive:

- 883 • The code enclosed in a CRITICAL/END CRITICAL directive pair must be a
884 structured block. It is noncompliant to branch into or out of the block.
885 • If a *name* is specified on a CRITICAL directive, the same *name* must also be
886 specified on the END CRITICAL directive. If no *name* appears on the CRITICAL
887 directive, no *name* can appear on the END CRITICAL directive.

888 See Section A.5, page 64, for an example that uses named CRITICAL sections.

889 **2.5.3 BARRIER Directive**

890 The BARRIER directive synchronizes all the threads in a team. When encountered,
891 each thread waits until all of the other threads in that team have reached this point.

892

The format of this directive is as follows:

893

```
!$OMP BARRIER
```

894

The following restrictions apply to the BARRIER directive:

895

- Work-sharing constructs and BARRIER directives must be encountered by all threads in a team or by none at all.
- Work-sharing constructs and BARRIER directives must be encountered in the same order by all threads in a team.

897

898

899

2.5.4 ATOMIC Directive

900

901

The ATOMIC directive ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

902

The format of this directive is as follows:

903

```
!$OMP ATOMIC
```

904

905

This directive applies only to the immediately following statement, which must have one of the following forms:

906

```
x = x operator expr  
x = expr operator x  
x = intrinsic_procedure_name ( x, expr_list )  
x = intrinsic_procedure_name ( expr_list, x )
```

907

908

909

910

911

912

913

914

915

916

In the preceding statements:

- *x* is a scalar variable of intrinsic type.
- *expr* is a scalar expression that does not reference *x*.
- *expr_list* is a comma-separated, non-empty list of scalar expressions that do not reference *x*. When *intrinsic_procedure_name* refers to IAND, IOR, or IEOR, exactly one expression must appear in *expr_list*.
- *intrinsic_procedure_name* is one of MAX, MIN, IAND, IOR, or IEOR.

- 917 • *operator* is one of +, *, -, /, .AND., .OR., .EQV., or .NEQV..
- 918 • The operators in *expr* must have precedence equal to or greater than the
919 precedence of *operator*, *x operator expr* must be mathematically equivalent to *x*
920 *operator (expr)*, and *expr operator x* must be mathematically equivalent to
921 *(expr) operator x*.
- 922 • The function *intrinsic_procedure_name*, the operator *operator*, and the assignment
923 must be the intrinsic procedure name, the intrinsic operator, and intrinsic
924 assignment.

925 This directive permits optimization beyond that of the necessary critical section
926 around the update of *x*. An implementation can rewrite the ATOMIC directive and the
927 corresponding assignment in the following way using a uniquely named critical
928 section for each object:

```
929     !$OMP ATOMIC
930        x = x operator expr
```

931 can be rewritten as

```
932     xtmp = expr
933     !$OMP CRITICAL (name)
934        x = x operator xtmp
935     !$OMP END CRITICAL (name)
```

936 where *name* is a unique name corresponding to the type or address of *x*.

937 Only the load and store of *x* are atomic; the evaluation of *expr* is not atomic. To avoid
938 race conditions, all updates of the location in parallel must be protected with the
939 ATOMIC directive, except those that are known to be free of race conditions.

940 The following restriction applies to the ATOMIC directive:

- 941 • All atomic references to the storage location of variable *x* throughout the program
942 are required to have the same type and type parameters.

943 Example:

```
944     !$OMP ATOMIC
945        Y(INDEX(I)) = Y(INDEX(I)) + B
```

946 See Section A.12, page 69, and Section A.23, page 82, for more examples using the
947 ATOMIC directive.

948 **2.5.5 FLUSH Directive**

949 The FLUSH directive, whether explicit or implied, identifies a sequence point at which
950 the implementation is required to ensure that each thread in the team has a
951 consistent view of certain variables in memory.

952 A consistent view requires that all memory operations (both reads and writes) that
953 occur before the FLUSH directive in the program be performed before the sequence
954 point in the executing thread; similarly, all memory operations that occur after the
955 FLUSH must be performed after the sequence point in the executing thread.

956 Implementations must ensure that modifications made to thread-visible variables
957 within the executing thread are made visible to all other threads at the sequence
958 point. For example, compilers must restore values from registers to memory, and
959 hardware may need to flush write buffers. Furthermore, implementations must
960 assume that thread-visible variables may have been updated by other threads at the
961 sequence point and must be retrieved from memory before their first use past the
962 sequence point.

963 Thread-visible variables are the following data items:

- Globally visible variables (in common blocks and in modules).
- Variables visible through host association.
- Local variables that have the SAVE attribute.
- Variables that appear in an EQUIVALENCE statement with a thread-visible variable.
- Local variables that have had their address taken and saved or have had their address passed to another subprogram.
- Local variables that do not have the SAVE attribute that are declared shared in the enclosing parallel region.
- Dummy arguments.
- All pointer dereferences.

975 The FLUSH directive only provides consistency between operations within the
976 executing thread and global memory. To achieve a globally consistent view across all
977 threads, each thread must execute a FLUSH operation.

978 The format of this directive is as follows:

```
979       !$OMP FLUSH [(List)]
```

980 This directive must appear at the precise point in the code at which the
981 synchronization is required. The optional *List* argument consists of a

982 comma-separated list of variables that need to be flushed in order to avoid flushing
983 all variables. The *list* should contain only named variables (see Section A.13, page
984 69). The FLUSH directive is implied for the following directives:

- 985 • BARRIER
- 986 • CRITICAL and END CRITICAL
- 987 • END DO
- 988 • END SECTIONS
- 989 • END SINGLE
- 990 • END WORKSHARE
- 991 • ORDERED and END ORDERED
- 992 • PARALLEL and END PARALLEL
- 993 • PARALLEL DO and END PARALLEL DO
- 994 • PARALLEL SECTIONS and END PARALLEL SECTIONS
- 995 • PARALLEL WORKSHARE and END PARALLEL WORKSHARE

996 The FLUSH directive is not implied if a NOWAIT clause is present.

997 It should be noted that the FLUSH directive is not implied by the following constructs:

- 998 • DO
- 999 • MASTER and END MASTER
- 1000 • SECTIONS
- 1001 • SINGLE
- 1002 • WORKSHARE

1003 2.5.6 ORDERED Directive

1004 The code enclosed within ORDERED and END ORDERED directives is executed in the
1005 order in which iterations would be executed in a sequential execution of the loop.

1006 The format of this directive is as follows:

1007 ! \$OMP ORDERED

1008 *block*

1009 ! \$OMP END ORDERED

1010 An ORDERED directive can appear only in the dynamic extent of a DO or PARALLEL DO
1011 directive. The DO directive to which the ordered section binds must have the ORDERED
1012 clause specified (see Section 2.3.1, page 15). One thread is allowed in an ordered
1013 section at a time. Threads are allowed to enter in the order of the loop iterations. No
1014 thread can enter an ordered section until it is guaranteed that all previous iterations
1015 have completed or will never execute an ordered section. This sequentializes and
1016 orders code within ordered sections while allowing code outside the section to run in
1017 parallel. ORDERED sections that bind to different DO directives are independent of
1018 each other.

1019 The following restrictions apply to the ORDERED directive:

- The code enclosed in an ORDERED/END ORDERED directive pair must be a
1020 structured block. It is noncompliant to branch into or out of the block.
- An ORDERED directive cannot bind to a DO directive that does not have the
1021 ORDERED clause specified.
- An iteration of a loop to which a DO directive is applied must not execute the same
1022 ORDERED directive more than once, and it must not execute more than one
1023 ORDERED directive.

1024 See Section A.10, page 68, and Section A.24, page 83, for examples using the
1025 ORDERED directive.

1029 2.6 Data Environment Constructs

1030 This section presents constructs for controlling the data environment during the
1031 execution of parallel constructs:

- Section 2.6.1, page 32, describes the THREADPRIVATE directive, which makes
1032 common blocks or variables local to a thread.
- Section 2.6.2, page 34, describes directive clauses that affect the data environment.
- Section 2.6.3, page 42, describes the data environment rules.

1036 **2.6.1 THREADPRIVATE Directive**1037 The THREADPRIVATE directive makes named common blocks and named variables
1038 private to a thread but global within the thread.1039 This directive must appear in the declaration section of a scoping unit in which the
1040 common block or variable is declared. Although variables in common blocks can be
1041 accessed by use association or host association, common block names cannot. This
1042 means that a common block name specified in a THREADPRIVATE directive must be
1043 declared to be a common block in the same scoping unit in which the THREADPRIVATE
1044 directive appears. Each thread gets its own copy of the common block or variable, so
1045 data written to the common block or variable by one thread is not directly visible to
1046 other threads. During serial portions and MASTER sections of the program, accesses
1047 are to the master thread's copy of the common block or variable. (See Section A.25,
1048 page 84, for examples.)1049 On entry to the first parallel region, an instance of a variable or common block that
1050 appears in a THREADPRIVATE directive is created for each thread. A variable is said
1051 to be affected by a COPYIN clause if the variable appears in the COPYIN clause or it is
1052 in a common block that appears in the COPYIN clause. If a THREADPRIVATE variable
1053 or a variable in a THREADPRIVATE common block is not affected by any COPYIN clause
1054 that appears on the first parallel region in a program, the variable or any subobject of
1055 the variable is initially defined or undefined according to the following rules:

- 1056
- 1057 • If it has the ALLOCATABLE attribute, each copy created will have an initial
1058 allocation status of not currently allocated.
 - 1059 • If it has the POINTER attribute:
 - 1060 – if it has an initial association status of disassociated, either through explicit
1061 initialization or default initialization, each copy created will have an
1062 association status of disassociated;
 - 1063 – otherwise, each copy created will have an association status of undefined.
 - 1064 • If it does not have either the POINTER or the ALLOCATABLE attribute:
 - 1065 – if it is initially defined, either through explicit initialization or default
1066 initialization, each copy created is so defined;
 - 1067 – otherwise, each copy created is undefined.

1068 On entry to a subsequent region, if the dynamic threads mechanism has been
1069 disabled, the definition, association, or allocation status of a thread's copy of a
1070 THREADPRIVATE variable or a variable in a THREADPRIVATE common block, that is
1071 not affected by any COPYIN clause that appears on the region, will be retained, and if
1072 it was defined, its value will be retained as well. In this case, if a THREADPRIVATE
1073 variable is referenced in both regions, then threads with the same thread number in
1074 their respective regions will reference the same copy of that variable. If the dynamic

1074 threads mechanism is enabled, the definition and association status of a thread's copy
1075 of the variable is undefined, and the allocation status of an allocatable array will be
1076 implementation-dependent. A variable with the allocatable attribute must not appear
1077 in a COPYIN clause, although a structure that has an ultimate component with the
1078 allocatable attribute may appear in a COPYIN clause. For more information on
1079 dynamic threads, see the OMP_SET_DYNAMIC library routine, Section 3.1.7, page 51,
1080 and the OMP_DYNAMIC environment variable, Section 4.3, page 60.

1081 On entry to any parallel region, each thread's copy of a variable that is affected by a
1082 COPYIN clause for the parallel region will acquire the allocation, association, or
1083 definition status of the master thread's copy, according to the following rules:

- 1084 • If it has the `POINTER` attribute:
 - 1085 – if the master thread's copy is associated with a target that each copy can
1086 become associated with, each copy will become associated with the same target;
 - 1087 – if the master thread's copy is disassociated, each copy will become disassociated;
 - 1088 – otherwise, each copy will have an undefined association status.
- 1089 • If it does not have the `POINTER` attribute, each copy becomes defined with the
1090 value of the master thread's copy as if by intrinsic assignment.

1091 If a common block or a variable that is declared in the scope of a module appears in a
1092 `THREADPRIVATE` directive, it implicitly has the `SAVE` attribute.

1093 The format of this directive is as follows:

```
1094 !$OMP THREADPRIVATE( List )
```

1095 where *List* is a comma-separated list of named variables and named common blocks.
1096 Common block names must appear between slashes.

1097 The following restrictions apply to the `THREADPRIVATE` directive:

- 1098 • The `THREADPRIVATE` directive must appear after every declaration of a thread
1099 private common block.
- 1100 • A blank common block cannot appear in a `THREADPRIVATE` directive.
- 1101 • It is noncompliant for a `THREADPRIVATE` variable or common block or its
1102 constituent variables to appear in any clause other than a `COPYIN` clause or a
1103 `COPYPRIVATE` clause. As a result, they are not permitted in a `PRIVATE`,
1104 `FIRSTPRIVATE`, `LASTPRIVATE`, `SHARED`, or `REDUCTION` clause. They are not
1105 affected by the `DEFAULT` clause.

- 1105 • A variable can only appear in a THREADPRIVATE directive in the scope in which it
1106 is declared. It must not be an element of a common block or be declared in an
1107 EQUIVALENCE statement.
- 1108 • A variable that appears in a THREADPRIVATE directive and is not declared in the
1109 scope of a module must have the SAVE attribute.

1111 2.6.2 Data Scope Attribute Clauses

1112 Several directives accept clauses that allow a user to control the scope attributes of
1113 variables for the duration of the construct. Not all of the following clauses are
1114 allowed on all directives, but the clauses that are valid on a particular directive are
1115 included with the description of the directive. If no data scope clauses are specified
1116 for a directive, the default scope for variables affected by the directive is SHARED. (See
1117 Section 2.6.3, page 42, for exceptions.)

1118 Scope attribute clauses that appear on a PARALLEL directive indicate how the
1119 specified variables are to be treated with respect to the parallel region associated with
1120 the PARALLEL directive. They do not indicate the scope attributes of these variables
1121 for any enclosing parallel regions, if they exist.

1122 In determining the appropriate scope attribute for a variable used in the lexical extent
1123 of a parallel region, all references and definitions of the variable must be considered,
1124 including references and definitions which occur in any nested parallel regions.

1125 Each clause accepts an argument *list*, which is a comma-separated list of named
1126 variables or named common blocks that are accessible in the scoping unit. Subobjects
1127 cannot be specified as items in any of the lists. When named common blocks appear
1128 in a list, their names must appear between slashes.

1129 When a named common block appears in a list, it has the same meaning as if every
1130 explicit member of the common block appeared in the list. A member of a common
1131 block is an explicit member if it is named in a COMMON statement which declares the
1132 common block, and it was declared in the same scoping unit in which the clause
1133 appears.

1134 Although variables in common blocks can be accessed by use association or host
1135 association, common block names cannot. This means that a common block name
1136 specified in a data scope attribute clause must be declared to be a common block in
1137 the same scoping unit in which the data scope attribute clause appears.

1138 The following sections describe the data scope attribute clauses:

- 1139 • Section 2.6.2.1, page 35, describes the PRIVATE clause.
- 1140 • Section 2.6.2.2, page 36, describes the SHARED clause.

- 1141 • Section 2.6.2.3, page 36, describes the DEFAULT clause.
- 1142 • Section 2.6.2.4, page 37, describes the FIRSTPRIVATE clause.
- 1143 • Section 2.6.2.5, page 38, describes the LASTPRIVATE clause.
- 1144 • Section 2.6.2.6, page 38, describes the REDUCTION clause.
- 1145 • Section 2.6.2.7, page 41, describes the COPYIN clause.
- 1146 • Section 2.6.2.8, page 41, describes the COPYPRIVATE clause.

1147 2.6.2.1 PRIVATE Clause

1148 The PRIVATE clause declares the variables in *list* to be private to each thread in a
1149 team.

1150 This clause has the following format:

PRIVATE(*list*)

1152 The behavior of a variable declared in a PRIVATE clause is as follows:

- 1153 1. A new object of the same type is declared once for each thread in the team. One
1154 thread in the team is permitted, but not required, to re-use the existing storage
1155 as the storage for the new object. For all other threads, new storage is created
1156 for the new object.
- 1157 2. All references to the original object in the lexical extent of the directive construct
1158 are replaced with references to the private object.
- 1159 3. Variables declared as PRIVATE are undefined for each thread on entering the
1160 construct, and the corresponding shared variable is undefined on exit from a
1161 parallel construct.
- 1162 4. A variable declared as PRIVATE may be storage-associated with other variables
1163 when the PRIVATE clause is encountered. Storage association may exist because
1164 of constructs such as EQUIVALENCE, COMMON, etc. If A is a variable appearing in
1165 a PRIVATE clause and B is a variable which was storage-associated with A, then:
 - 1166 a. The contents, allocation, and association status of B are undefined on entry
1167 to the parallel construct.
 - 1168 b. Any definition of A, or of its allocation or association status, causes the
1169 contents, allocation, and association status of B to become undefined.
 - 1170 c. Any definition of B, or of its allocation or association status, causes the
1171 contents, allocation, and association status of A to become undefined.

1172 See Section A.20, page 78, and Section A.21, page 78, for examples.

- 1173
- 1174 5. Contents, allocation state, and association status of variables defined as
1175 PRIVATE are undefined when they are referenced outside the lexical extent (but
1176 inside the dynamic extent) of the construct, unless they are passed as actual
1177 arguments to called routines. Scope clauses apply only to variables in the lexical
1178 extent of the directive on which the clause appears, with the exception of
1179 variables passed as actual arguments.
- 1180 6. If a variable is declared as PRIVATE, and the variable is referenced in the
1181 definition of a statement function, and the statement function is used within the
1182 lexical extent of the directive construct, then the statement function may
1183 reference either the SHARED version of the variable or the PRIVATE version.
1184 Which version is referenced is implementation-dependent.

1184 2.6.2.2 SHARED Clause

1185 The SHARED clause makes variables that appear in the *list* shared among all the
1186 threads in a team. All threads within a team access the same storage area for
1187 SHARED data.

1188 This clause has the following format:

1189

SHARED(<i>list</i>)

1190 That each thread in the team access the same storage area for a shared variable does
1191 not guarantee that the threads are immediately aware of changes made to the
1192 variable by another thread. An implementation may store the new values of shared
1193 variables in registers or caches, and those new values may not be stored into the
1194 shared storage area until a FLUSH is performed.

1195 2.6.2.3 DEFAULT Clause

1196 The DEFAULT clause allows the user to specify a PRIVATE, SHARED, or NONE scope
1197 attribute for all variables in the lexical extent of any parallel region. Variables in
1198 THREADPRIVATE common blocks are not affected by this clause.

1199 This clause has the following format:

1200

DEFAULT(PRIVATE SHARED NONE)

1201 The PRIVATE, SHARED, and NONE specifications have the following effects:

- Specifying `DEFAULT(PRIVATE)` makes all named objects in the lexical extent of the parallel region, including common block variables but excluding `THREADPRIVATE` variables, private to a thread as if each variable were listed explicitly in a `PRIVATE` clause.
- Specifying `DEFAULT(SHARED)` makes all named objects in the lexical extent of the parallel region shared among the threads in a team, as if each variable were listed explicitly in a `SHARED` clause. In the absence of an explicit `DEFAULT` clause, the default behavior is the same as if `DEFAULT(SHARED)` were specified.
- Specifying `DEFAULT(NONE)` requires that each variable used in the lexical extent of the parallel region be explicitly listed in a data scope attribute clause on the parallel region, unless it is one of the following:
 - `THREADPRIVATE`.
 - A Cray pointee (Note: the associated Cray pointer must have its data scope attribute implicitly or explicitly specified).
 - A loop iteration variable used only as a loop iteration variable for sequential loops in the lexical extent of the region or parallel `DO` loops that bind to the region.
 - `IMPLIED-DO` or `FORALL` indices.
 - Only used in work-sharing constructs that bind to the region, and is specified in a data scope attribute clause for each such construct.

Only one `DEFAULT` clause can be specified on a `PARALLEL` directive.

Variables can be exempted from a defined default using the `PRIVATE`, `SHARED`, `FIRSTPRIVATE`, `LASTPRIVATE`, and `REDUCTION` clauses. As a result, the following example is legal:

```
!$OMP PARALLEL DO DEFAULT(PRIVATE), FIRSTPRIVATE(I), SHARED(X),  
!$OMP& SHARED(R) LASTPRIVATE(I)
```

2.6.2.4 `FIRSTPRIVATE` Clause

The `FIRSTPRIVATE` clause provides a superset of the functionality provided by the `PRIVATE` clause.

This clause has the following format:

```
FIRSTPRIVATE ( list )
```

1233 Variables that appear in the *list* are subject to PRIVATE clause semantics described in
1234 Section 2.6.2.1, page 35. In addition, private copies of the variables are initialized
1235 from the original object existing before the construct.

1236 **2.6.2.5 LASTPRIVATE Clause**

1237 The LASTPRIVATE clause provides a superset of the functionality provided by the
1238 PRIVATE clause.

1239 This clause has the following format:

1240 LASTPRIVATE (*list*)

1241 Variables that appear in the *list* are subject to the PRIVATE clause semantics
1242 described in Section 2.6.2.1, page 35. When the LASTPRIVATE clause appears on a DO
1243 directive, the thread that executes the sequentially last iteration updates the version
1244 of the object it had before the construct (see Section A.6, page 65, for an example).
1245 When the LASTPRIVATE clause appears in a SECTIONS directive, the thread that
1246 executes the lexically last SECTION updates the version of the object it had before the
1247 construct. Subobjects that are not assigned a value by the last iteration of the DO or
1248 the lexically last SECTION of the SECTIONS directive are undefined after the construct.

1249 If the LASTPRIVATE clause is used on a construct to which NOWAIT is also applied,
1250 the shared variable remains undefined until a barrier synchronization has been
1251 performed to ensure that the thread that executed the sequentially last iteration has
1252 stored that variable.

1253 **2.6.2.6 REDUCTION Clause**

1254 This clause performs a reduction on the variables that appear in *list*, with the
1255 operator *operator* or the intrinsic *intrinsic_procedure_name*, where *operator* is one of
1256 the following: +, *, -, .AND., .OR., .EQV., or .NEQV., and *intrinsic_procedure_name*
1257 refers to one of the following: MAX, MIN, IAND, IOR, or IEOR.

1258 This clause has the following format:

1259 REDUCTION ({ *operator* | *intrinsic_procedure_name* } : *list*)

1260 Variables in *list* must be named variables of intrinsic type. Deferred shape and
1261 assumed size arrays are not allowed on the reduction clause. Since the intermediate
1262 values of the REDUCTION variables may be combined in random order, there is no
1263 guarantee that bit-identical results will be obtained for either integer or floating point
1264 reductions from one parallel run to another.

1265 Variables that appear in a `REDUCTION` clause must be `SHARED` in the enclosing
1266 context. A private copy of each variable in *list* is created for each thread as if the
1267 `PRIVATE` clause had been used. The private copy is initialized according to the
1268 operator. See Table 2, page 40, for more information.

1269 At the end of the `REDUCTION`, the shared variable is updated to reflect the result of
1270 combining the original value of the (shared) reduction variable with the final value of
1271 each of the private copies using the operator specified. The reduction operators are all
1272 associative (except for subtraction), and the compiler can freely reassociate the
1273 computation of the final value (the partial results of a subtraction reduction are
1274 added to form the final value).

1275 The value of the shared variable becomes undefined when the first thread reaches the
1276 containing clause, and it remains so until the reduction computation is complete.
1277 Normally, the computation is complete at the end of the `REDUCTION` construct;
1278 however, if the `REDUCTION` clause is used on a construct to which `NOWAIT` is also
1279 applied, the shared variable remains undefined until a barrier synchronization has
1280 been performed to ensure that all the threads have completed the `REDUCTION` clause.

1281 The `REDUCTION` clause is intended to be used on a region or work-sharing construct
1282 in which the reduction variable or a subobject of the reduction variable is used only in
1283 reduction statements with one of the following forms:

```
1284     x = x operator expr
1285     x = expr operator x  (except for subtraction)
1286     x = intrinsic_procedure_name (x,expr_list)
1287     x = intrinsic_procedure_name (expr_list, x)
```

1288 In the preceding statements:

- 1289 • *x* is a scalar variable of intrinsic type.
- 1290 • *expr* is a scalar expression that does not reference *x*.
- 1291 • *expr_list* is a comma-separated, non-empty list of scalar expressions that do not
1292 reference *x*. When *intrinsic_procedure_name* refers to `IAND`, `IOR`, or `IEOR`, exactly
1293 one expression must appear in *expr_list*.
- 1294 • *intrinsic_procedure_name* is one of `MAX`, `MIN`, `IAND`, `IOR`, or `IEOR`.
- 1295 • *operator* is one of `+`, `*`, `-`, `.AND.`, `.OR.`, `.EQV.`, or `.NEQV.`
- 1296 • The operators in *expr* must have precedence equal to or greater than the
1297 precedence of *operator*; *x operator expr* must be mathematically equivalent to *x*

operator $(expr)$, and $expr$ *operator* x must be mathematically equivalent to $(expr)$ *operator* x .

- The function *intrinsic_procedure_name*, the operator *operator*, and the assignment must be the intrinsic procedure name, the intrinsic operator, and intrinsic assignment.

Some reductions can be expressed in other forms. For instance, a MAX reduction might be expressed as follows:

IF (X .LT. expr) X = expr

Alternatively, the reduction might be hidden inside a subroutine call. The user should be careful that the operator specified in the `REDUCTION` clause matches the reduction operation.

The following table lists the operators and intrinsics that are valid and their canonical initialization values. The actual initialization value will be consistent with the data type of the reduction variable.

Table 2. Reduction Variable Initialization Values

<u>Operator/Intrinsic</u>	<u>Initialization</u>
+	0
*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.
MAX	Smallest representable number
MIN	Largest representable number
IAND	All bits on
IOR	0
IEOR	0

See Section A.7, page 65, for an example that uses the `+` operator.

Any number of reduction clauses can be specified on the directive, but a variable can appear only once in the **REDUCTION** clause(s) for that directive.

Example:

```
!$OMP DO REDUCTION(+: A, Y) REDUCTION(.OR.: AM)
```

1331 2.6.2.7 COPYIN *Clause*

1332 The COPYIN clause applies only to variables, common blocks, and variables in
 1333 common blocks that are declared as THREADPRIVATE. A COPYIN clause on a parallel
 1334 region specifies that the data in the master thread of the team be copied to the thread
 1335 private copies of the common blocks or variables at the beginning of the parallel
 1336 region as described in Section 2.6.1, page 32.

1337 This clause has the following format:

1338 `COPYIN(list)`

1339 If a common block appears in a THREADPRIVATE directive, it is not necessary to
 1340 specify the whole common block. Named variables appearing in the THREADPRIVATE
 1341 common block can be specified in the *list*.

1342 Although variables in common blocks can be accessed by use association or host
 1343 association, common block names cannot. This means that a common block name
 1344 specified in a COPYIN clause must be declared to be a common block in the same
 1345 scoping unit in which the COPYIN clause appears. See Section A.25, page 84, for more
 1346 information.

1347 In the following example, the common blocks `BLK1` and `FIELDS` are specified as
 1348 thread private, but only one of the variables in common block `FIELDS` is specified to
 1349 be copied in.

```
1350      COMMON /BLK1/ SCRATCH
1351      COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD
1352      !$OMP THREADPRIVATE(/BLK1/, /FIELDS/)
1353      !$OMP PARALLEL DEFAULT(PRIVATE) COPYIN(/BLK1/, ZFIELD)
```

1354 An OpenMP-compliant implementation is required to ensure that the value of each
 1355 thread private copy is the same as the value of the master thread copy when the
 1356 master thread reached the directive containing the COPYIN clause.

1357 2.6.2.8 COPYPRIVATE *Clause*

1358 The COPYPRIVATE clause uses a private variable to broadcast a value, or a pointer to
 1359 a shared object, from one member of a team to the other members. It is an
 1360 alternative to using a shared variable for the value, or pointer association, and is
 1361 useful when providing such a shared variable would be difficult (for example, in a
 1362 recursion requiring a different variable at each level). The COPYPRIVATE clause can
 1363 only appear on the `END SINGLE` directive.

1364 This clause has the following format:

1365 **COPYPRIVATE (*list*)**

1366 Variables in the *list* must not appear in a PRIVATE or FIRSTPRIVATE clause for the
1367 SINGLE construct. If the directive is encountered in the dynamic extent of a parallel
1368 region, variables in the list must be private in the enclosing context. If a common
1369 block is specified, then it must be THREADPRIVATE, and the effect is the same as if
1370 the variable names in its common block object list were specified.

1371 The effect of the COPYPRIVATE clause on the variables in its list occurs after the
1372 execution of the code enclosed within the SINGLE construct, and before any threads in
1373 the team have left the barrier at the end of the construct. If the variable is not a
1374 pointer, then in all other threads in the team, that variable becomes defined (as if by
1375 assignment) with the value of the corresponding variable in the thread that executed
1376 the enclosed code. If the variable is a pointer, then in all other threads in the team,
1377 that variable becomes pointer associated (as if by pointer assignment) with the
1378 corresponding variable in the thread that executed the enclosed code. (See Section
1379 A.27, page 89, for examples of the COPYPRIVATE clause.)

1380 **2.6.3 Data Environment Rules**

1381 A program that conforms to the OpenMP Fortran API must adhere to the following
1382 rules and restrictions with respect to data scope:

- 1383 1. Sequential DO loop control variables in the lexical extent of a PARALLEL region
1384 that would otherwise be SHARED based on default rules are automatically made
1385 private on the PARALLEL directive. Sequential DO loop control variables with no
1386 enclosing PARALLEL region are not made private automatically. It is up to the
1387 user to guarantee that these indexes are private if the containing procedures are
1388 called from a PARALLEL region.

1389 All implied DO loop control variables and FORALL indexes are automatically made
1390 private at the enclosing implied DO or FORALL construct.

- 1391 2. Variables that are privatized in a parallel region may be privatized again on an
1392 enclosed work-sharing directive. As a result, variables that appear in a PRIVATE
1393 clause on a work-sharing directive may either have a shared or a private scope in
1394 the enclosing parallel region. Variables that appear on the FIRSTPRIVATE,
1395 LASTPRIVATE, and REDUCTION clauses on a work-sharing directive must have
1396 shared scope in the enclosing parallel region.
- 1397 3. Variables that appear in a reduction list in a parallel region cannot be privatized
1398 on an enclosed work-sharing directive.
- 1399 4. A variable that appears in a PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or
1400 REDUCTION clause must be definable.

- 1401 5. Assumed-size arrays cannot be declared PRIVATE, FIRSTPRIVATE,
1402 LASTPRIVATE, or COPYPRIVATE. Array dummy arguments that are explicitly
1403 shaped (including variable dimensioned) and assumed-shape arrays can be
1404 declared in any scoping clause.
- 1405 6. Fortran pointers and allocatable arrays can be declared PRIVATE or SHARED but
1406 not FIRSTPRIVATE or LASTPRIVATE.
- 1407 Within a parallel region, the initial status of a private pointer is undefined.
1408 Private pointers that become allocated during the execution of a parallel region
1409 should be explicitly deallocated by the program prior to the end of the parallel
1410 region to avoid memory leaks.
- 1411 The association status of a SHARED pointer becomes undefined upon entry to and
1412 on exit from the parallel construct if it is associated with a target or a subobject
1413 of a target that is in a PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION
1414 clause inside the parallel construct. An allocatable array declared PRIVATE must
1415 have an allocation status of “not currently allocated” on entry to and on exit from
1416 the construct.
- 1417 7. PRIVATE or SHARED attributes can be declared for a Cray pointer but not for the
1418 pointee. The scope attribute for the pointee is determined at the point of pointer
1419 definition. It is noncompliant to declare a scope attribute for a pointee. Cray
1420 pointers may not be specified in FIRSTPRIVATE or LASTPRIVATE clauses.
- 1421 8. Scope clauses apply only to variables in the lexical extent of the directive on
1422 which the clause appears, with the exception of variables passed as actual
1423 arguments. Local variables in called routines that do not have the SAVE attribute
1424 are PRIVATE. Common blocks and module variables in called routines in the
1425 dynamic extent of a parallel region always have an implicit SHARED attribute,
1426 unless they are THREADPRIVATE. Local variables in called routines that have the
1427 SAVE attribute are SHARED. (See Section A.26, page 88, for examples.)
- 1428 9. When a named common block is specified in a PRIVATE, FIRSTPRIVATE, or
1429 LASTPRIVATE clause of a directive, none of its constituent elements may be
1430 declared in another data scope attribute clause in that directive. It should be
1431 noted that when individual members of a common block are privatized, the
1432 storage of the specified variables is no longer associated with the storage of the
1433 common block itself. (See Section A.25, page 84, for examples.)
- 1434 10. Variables that are not allowed in the PRIVATE and SHARED clauses are not
1435 affected by DEFAULT(PRIVATE) or DEFAULT(SHARED) clauses, respectively.
- 1436 11. Clauses can be repeated as needed, but each variable and each named common
1437 block can appear explicitly in only one clause per directive, with the following
1438 exceptions:
- 1439 • A variable can be declared both FIRSTPRIVATE and LASTPRIVATE.

- 1440 • Variables affected by the `DEFAULT` clause can be listed explicitly in a clause to
1441 override the default specification.

1442 12. Variables that are declared `LASTPRIVATE` or `REDUCTION` for a work-sharing
1443 directive for which `NOWAIT` appears must not be used prior to a barrier.

1444 13. Variables that appear in namelist statements, in variable format expressions,
1445 and in expressions for statement function definitions must not be specified in
1446 `PRIVATE`, `FIRSTPRIVATE`, or `LASTPRIVATE` clauses.

1447 14. The shared variables that are specified in `REDUCTION` or `LASTPRIVATE` clauses
1448 become defined at the end of the construct. Any concurrent uses or definitions of
1449 those variables must be synchronized with the definition that occurs at the end
1450 of the construct to avoid race conditions.

1451 15. If the following three conditions hold regarding an actual argument in a reference
1452 to a non-intrinsic procedure, then any references to (or definitions of) the shared
1453 storage that is associated with the dummy argument by any other thread must
1454 be synchronized with the procedure reference to avoid possible race conditions:

- 1455 a. The actual argument is one of the following:

- 1456 • A `SHARED` variable
- 1457 • A subobject of a `SHARED` variable
- 1458 • An object associated with a `SHARED` variable
- 1459 • An object associated with a subobject of a `SHARED` variable

- 1460 b. The actual argument is also one of the following:

- 1461 • An array section with a vector subscript
- 1462 • An array section
- 1463 • An assumed-shape array
- 1464 • A pointer array

- 1465 c. The associated dummy argument for this actual argument is an
1466 explicit-shape array or an assumed-size array.

1467 The situations described above may result in the value of the shared variable
1468 being copied into temporary storage before the procedure reference, and back out
1469 of the temporary storage into the actual argument storage after the procedure
1470 reference. This effectively results in references to and definitions of the storage
1471 during the procedure reference.

1472 16. An OpenMP-compliant implementation must adhere to the following rule:

- 1473 • If a variable is specified as FIRSTPRIVATE and LASTPRIVATE, the
1474 implementation must ensure that the update required for LASTPRIVATE
1475 occurs after all initializations for FIRSTPRIVATE.

- 1476 17. An implementation may generate references to any object that appears or an
1477 object in a common block that appears in a REDUCTION, FIRSTPRIVATE,
1478 LASTPRIVATE, COPYPRIVATE, or COPYIN clause, on entry to (for FIRSTPRIVATE
1479 and COPYIN) or exit from (for REDUCTION, LASTPRIVATE, and COPYPRIVATE) a
1480 construct. Except for an object with the pointer attribute in a COPYPRIVATE
1481 clause, if a reference to the object as the expression in an intrinsic assignment
1482 statement would give an exceptional value, or have undefined behavior, at that
1483 point in the program, then the generated reference may have the same behavior.

1484 2.7 Directive Binding

1485 An OpenMP-compliant implementation must adhere to the following rules with
1486 respect to the dynamic binding of directives:

- 1487 • A parallel region is available for binding purposes, whether it is serialized or
1488 executed in parallel.
- 1489 • The DO, SECTIONS, SINGLE, MASTER, BARRIER, and WORKSHARE directives bind to
1490 the dynamically enclosing PARALLEL directive, if one exists. (See Section A.19,
1491 page 77, for an example.) The dynamically enclosing PARALLEL directive is the
1492 closest enclosing PARALLEL directive regardless of the value of the expression in
1493 the IF clause, should the clause be present.
- 1494 • The ORDERED directive binds to the dynamically enclosing DO directive.
- 1495 • The ATOMIC directive enforces exclusive access with respect to ATOMIC directives
1496 in all threads, not just the current team.
- 1497 • The CRITICAL directive enforces exclusive access with respect to CRITICAL
1498 directives in all threads, not just the current team.
- 1499 • A directive can never bind to any directive outside the closest enclosing PARALLEL.

1500 2.8 Directive Nesting

1501 An OpenMP-compliant implementation must adhere to the following rules with
1502 respect to the dynamic nesting of directives:

- 1503 • A **PARALLEL** directive dynamically inside another **PARALLEL** directive logically
1504 establishes a new team, which is composed of only the current thread, unless
1505 nested parallelism is enabled.
- 1506 • DO, SECTIONS, SINGLE, and WORKSHARE directives that bind to the same
1507 **PARALLEL** directive are not allowed to be nested one inside the other.
- 1508 • DO, SECTIONS, SINGLE, and WORKSHARE directives are not permitted in the
1509 dynamic extent of CRITICAL, ORDERED, and MASTER directives.
- 1510 • BARRIER directives are not permitted in the dynamic extent of DO, SECTIONS,
1511 SINGLE, WORKSHARE, MASTER, CRITICAL, and ORDERED directives.
- 1512 • MASTER directives are not permitted in the dynamic extent of DO, SECTIONS,
1513 SINGLE, WORKSHARE, MASTER, CRITICAL, and ORDERED directives.
- 1514 • ORDERED directives must appear in the dynamic extent of a DO or PARALLEL DO
1515 directive which has an ORDERED clause.
- 1516 • ORDERED directives are not allowed in the dynamic extent of SECTIONS, SINGLE,
1517 WORKSHARE, CRITICAL, and MASTER directives.
- 1518 • CRITICAL directives with the same name are not allowed to be nested one inside
1519 the other.
- 1520 • Any directive set that is legal when executed dynamically inside a PARALLEL
1521 region is also legal when executed outside a parallel region. When executed
1522 dynamically outside a user-specified parallel region, the directive is executed with
1523 respect to a team composed of only the master thread.

1524 See Section A.17, page 73, for legal examples of directive nesting, and Section A.18,
1525 page 74, for invalid examples.

1527 This section describes the OpenMP Fortran API run-time library routines that can be
 1528 used to control and query the parallel execution environment. A set of general
 1529 purpose lock routines and two portable timer routines are also provided.

1530 OpenMP Fortran API run-time library routines are external procedures. In the
 1531 following descriptions, *scalar_integer_expression* is a default scalar integer expression,
 1532 and *scalar_logical_expression* is a default scalar logical expression. The return values
 1533 of these routines are also of default kind, unless otherwise specified.

1534 Interface declarations for the OpenMP Fortran runtime library routines described in
 1535 this chapter shall be provided by an OpenMP-compliant implementation in the form
 1536 of a Fortran INCLUDE file named `omp_lib.h` or a Fortran 90 MODULE named
 1537 `omp_lib`. This file must define the following:

- 1538 • The interfaces of all of the routines in this chapter.
- 1539 • The INTEGER PARAMETER `omp_lock_kind` that defines the KIND type
 1540 parameters used for simple lock variables in the `OMP_*_LOCK` routines.
- 1541 • the INTEGER PARAMETER `omp_nest_lock_kind` that defines the KIND type
 1542 parameters used for the nestable lock variables in the `OMP_*_NEST_LOCK` routines.
- 1543 • the INTEGER PARAMETER `openmp_version` with a value of the C preprocessor
 1544 macro `_OPENMP` (see Section 2.1.3, page 10) that has the form `YYYYMM` where `YYYY`
 1545 and `MM` are the year and month designations of the version of the OpenMP Fortran
 1546 API that the implementation supports.

1547 See Appendix D, page 105, for examples of these files.

1548 3.1 Execution Environment Routines

1549 The following sections describe the execution environment routines:

- 1550 • Section 3.1.1, page 48, describes the `OMP_SET_NUM_THREADS` subroutine.
- 1551 • Section 3.1.2, page 48, describes the `OMP_GET_NUM_THREADS` function.
- 1552 • Section 3.1.3, page 49, describes the `OMP_GET_MAX_THREADS` function.
- 1553 • Section 3.1.4, page 49, describes the `OMP_GET_THREAD_NUM` function.
- 1554 • Section 3.1.5, page 50, describes the `OMP_GET_NUM_PROCS` function.
- 1555 • Section 3.1.6, page 50, describes the `OMP_IN_PARALLEL` function.

- 1556 • Section 3.1.7, page 51, describes the `OMP_SET_DYNAMIC` subroutine.
1557 • Section 3.1.8, page 51, describes the `OMP_GET_DYNAMIC` function.
1558 • Section 3.1.9, page 52, describes the `OMP_SET_NESTED` subroutine.
1559 • Section 3.1.10, page 52, describes the `OMP_GET_NESTED` function.

1560 **3.1.1 `OMP_SET_NUM_THREADS` Subroutine**

1561 The `OMP_SET_NUM_THREADS` subroutine sets the number of threads to use for
1562 subsequent parallel regions.

1563 The format of this subroutine is as follows:

1564

SUBROUTINE `OMP_SET_NUM_THREADS`(*scalar_integer_expression*)

1565 The value of the *scalar_integer_expression* must be positive. The effect of this function
1566 depends on whether dynamic adjustment of the number of threads is enabled. If
1567 dynamic adjustment is disabled, the value of the *scalar_integer_expression* is used as
1568 the number of threads for all subsequent parallel regions prior to the next call to this
1569 function; otherwise, the value is used as the maximum number of threads that will be
1570 used. This function has effect only when called from serial portions of the program. If
1571 it is called from a portion of the program where the `OMP_IN_PARALLEL` function
1572 returns `.TRUE.`, the behavior of this function is unspecified. For additional
1573 information on this subject, see the `OMP_SET_DYNAMIC` subroutine described in
1574 Section 3.1.7, page 51, and the `OMP_GET_DYNAMIC` function described in Section 3.1.8,
1575 page 51, and the example in Section A.11, page 68.

1576 Resource constraints on an OpenMP parallel program may change the number of
1577 threads that a user is allowed to create at different phases of a program's execution.
1578 When dynamic adjustment of the number of threads is enabled, requests for more
1579 threads than an implementation can support are satisfied by a smaller number of
1580 threads. If dynamic adjustment of the number of threads is disabled, the behavior of
1581 this function is implementation-dependent.

1582 This call has precedence over the `OMP_NUM_THREADS` environment variable (see
1583 Section 4.2, page 60).

1584 **3.1.2 `OMP_GET_NUM_THREADS` Function**

1585 The `OMP_GET_NUM_THREADS` function returns the number of threads currently in the
1586 team executing the parallel region from which it is called.

1587

The format of this function is as follows:

1588

```
INTEGER FUNCTION OMP_GET_NUM_THREADS()
```

1589

The `OMP_SET_NUM_THREADS` call and the `OMP_NUM_THREADS` environment variable control the number of threads in a team. For more information on the `OMP_SET_NUM_THREADS` library routine, see Section 3.1.1, page 48. For more information on the `OMP_NUM_THREADS` environment variable, see Section 4.2, page 60.

1593

1594

1595

If the number of threads has not been explicitly set by the user, the default is implementation-dependent. This function binds to the closest enclosing `PARALLEL` directive. For more information on the `PARALLEL` directive, see Section 2.2, page 12.

1596

1597

1598

If this call is made from the serial portion of a program, or from a nested parallel region that is serialized, this function returns 1. (See Section A.14, page 70, for an example.)

1599

3.1.3 `OMP_GET_MAX_THREADS` Function

1600

1601

1602

The `OMP_GET_MAX_THREADS` function returns the maximum value that can be returned by calls to the `OMP_GET_NUM_THREADS` function. For more information on `OMP_GET_NUM_THREADS`, see Section 3.1.2, page 48.

1603

The format of this function is as follows:

1604

```
INTEGER FUNCTION OMP_GET_MAX_THREADS()
```

1605

1606

1607

1608

1609

If `OMP_SET_NUM_THREADS` is used to change the number of threads, subsequent calls to `OMP_GET_MAX_THREADS` will return the new value. This function can be used to allocate maximum sized per-thread data structures when the `OMP_SET_DYNAMIC` subroutine is set to `.TRUE.`. For more information on the `OMP_SET_DYNAMIC` library routine, see Section 3.1.7, page 51.

1610

1611

This function has global scope and returns the maximum value whether executing from a serial region or a parallel region.

1612

3.1.4 `OMP_GET_THREAD_NUM` Function

1613

1614

The `OMP_GET_THREAD_NUM` function returns the number of the current thread within the team. The thread number lies between 0 and `OMP_GET_NUM_THREADS()`-1,

1615 inclusive. (See the second example in Section A.14, page 70.) The master thread of
1616 the team is thread 0.

1617 The format of this function is as follows:

```
1618 INTEGER FUNCTION OMP_GET_THREAD_NUM()
```

1619 This function binds to the closest enclosing PARALLEL directive. For more information
1620 on the PARALLEL directive, see Section 2.2, page 12.

1621 When called from a serial region, OMP_GET_THREAD_NUM returns 0. When called from
1622 within a nested parallel region that is serialized, this function returns 0.

1623 **3.1.5 OMP_GET_NUM_PROCS Function**

1624 The OMP_GET_NUM_PROCS function returns the number of processors that are
1625 available to the program.

1626 The format of this function is as follows:

```
1627 INTEGER FUNCTION OMP_GET_NUM_PROCS()
```

1628 **3.1.6 OMP_IN_PARALLEL Function**

1629 OMP_IN_PARALLEL returns the logical OR of the IF clause from all dynamically
1630 enclosing parallel regions.

- 1631 • If a parallel region does not have an IF clause, this is equivalent to IF(.TRUE.)
1632 and OMP_IN_PARALLEL returns .TRUE..
- 1633 • If there are no dynamically enclosing parallel regions, then OMP_IN_PARALLEL
1634 returns .FALSE..

1635 The format of this function is as follows:

```
1636 LOGICAL FUNCTION OMP_IN_PARALLEL()
```

1637 This function has global scope. As a result, it will always return .TRUE. within the
1638 dynamic extent of a region executing in parallel, regardless of nested regions that are
1639 serialized.

1640 **3.1.7 OMP_SET_DYNAMIC Subroutine**1641 The OMP_SET_DYNAMIC subroutine enables or disables dynamic adjustment of the
1642 number of threads available for execution of parallel regions.

1643 The format of this subroutine is as follows:

1644

```
SUBROUTINE OMP_SET_DYNAMIC(scalar_logical_expression)
```

1645 If *scalar_logical_expression* evaluates to .TRUE., the number of threads that are used
1646 for executing subsequent parallel regions can be adjusted automatically by the
1647 run-time environment to obtain the best use of system resources. As a consequence,
1648 the number of threads specified by the user is the maximum thread count. The
1649 number of threads always remains fixed over the duration of each parallel region and
1650 is reported by the OMP_GET_NUM_THREADS library routine. This function has effect
1651 only when called from serial portions of the program. For more information on the
1652 OMP_GET_NUM_THREADS library routine, see Section 3.1.2, page 48.1653 If *scalar_logical_expression* evaluates to .FALSE., dynamic adjustment is disabled.
1654 (See Section A.11, page 68, for an example.)1655 A call to OMP_SET_DYNAMIC has precedence over the OMP_DYNAMIC environment
1656 variable. For more information on the OMP_DYNAMIC environment variable, see
1657 Section 4.3, page 60.1658 The default for dynamic thread adjustment is implementation-dependent. As a result,
1659 user codes that depend on a specific number of threads for correct execution should
1660 explicitly disable dynamic threads. Implementations are not required to provide the
1661 ability to dynamically adjust the number of threads, but they are required to provide
1662 the interface in order to support portability across platforms.1663 **3.1.8 OMP_GET_DYNAMIC Function**1664 The OMP_GET_DYNAMIC function returns .TRUE. if dynamic thread adjustment is
1665 enabled and returns .FALSE. otherwise. For more information on dynamic thread
1666 adjustment, see Section 3.1.7, page 51.

1667 The format of this function is as follows:

1668

```
LOGICAL FUNCTION OMP_GET_DYNAMIC()
```

1669 If the implementation does not implement dynamic adjustment of the number of
1670 threads, this function always returns .FALSE..

1671 **3.1.9 OMP_SET_NESTED Subroutine**

1672 The OMP_SET_NESTED subroutine enables or disables nested parallelism.

1673 The format of this subroutine is as follows:

1674

SUBROUTINE OMP_SET_NESTED(*scalar_logical_expression*)

1675 If *scalar_logical_expression* evaluates to .FALSE., nested parallelism is disabled,
1676 which is the default, and nested parallel regions are serialized and executed by the
1677 current thread. If set to .TRUE., nested parallelism is enabled, and parallel regions
1678 that are nested can deploy additional threads to form the team.1679 This call has precedence over the OMP_NESTED environment variable. For more
1680 information on the OMP_NESTED environment variable, see Section 4.4, page 61.1681 When nested parallelism is enabled, the number of threads used to execute nested
1682 parallel regions is implementation-dependent. As a result, OpenMP-compliant
1683 implementations are allowed to serialize nested parallel regions even when nested
1684 parallelism is enabled.1685 **3.1.10 OMP_GET_NESTED Function**1686 The OMP_GET_NESTED function returns .TRUE. if nested parallelism is enabled and
1687 .FALSE. if nested parallelism is disabled. For more information on nested
1688 parallelism, see Section 3.1.9, page 52.

1689 The format of this function is as follows:

1690

LOGICAL FUNCTION OMP_GET_NESTED()

1691 If an implementation does not implement nested parallelism, this function always
1692 returns .FALSE..1693 **3.2 Lock Routines**1694 The OpenMP run-time library includes a set of general-purpose locking routines that
1695 take lock variables as arguments. A lock variable must be accessed only through the
1696 routines described in this section. For all of these routines, a lock variable should be
1697 of type integer and of a KIND large enough to hold an address.

1698 Two types of locks are supported: simple locks and nestable locks. Nestable locks may
1699 be locked multiple times by the same thread before being unlocked; simple locks may
1700 not be locked if they are already in a locked state. Simple lock variables are
1701 associated with simple locks and may only be passed to simple lock routines.
1702 Nestable lock variables are associated with nestable locks and may only be passed to
1703 nestable lock routines.

1704 In the descriptions that follow, *svar* is a simple lock variable and *nvar* is a nestable
1705 lock variable. Using the defined parameters described at the beginning of this
1706 chapter (Chapter 3, page 47), these lock variables may be declared as follows:

1707

```
INTEGER (KIND=OMP_LOCK_KIND) :: svar
```

1708

```
INTEGER (KIND=OMP_NEST_LOCK_KIND) :: nvar
```

1709 The simple locking routines are as follows:

- 1710 • The **OMP_INIT_LOCK** subroutine initializes a simple lock (see Section 3.2.1, page
1711 54).
1712 • The **OMP_DESTROY_LOCK** subroutine removes a simple lock (see Section 3.2.2, page
1713 54).
1714 • The **OMP_SET_LOCK** subroutine sets a simple lock when it becomes available (see
1715 Section 3.2.3, page 54).
1716 • The **OMP_UNSET_LOCK** subroutine releases a simple lock (see Section 3.2.4, page
1717 55).
1718 • The **OMP_TEST_LOCK** function tests and possibly sets a simple lock (see Section
1719 3.2.5, page 55).

1720 The nestable lock routines are as follows:

- 1721 • The **OMP_INIT_NEST_LOCK** subroutine initializes a nestable lock (see Section
1722 3.2.1, page 54).
1723 • The **OMP_DESTROY_NEST_LOCK** subroutine removes a nestable lock (see Section
1724 3.2.2, page 54).
1725 • The **OMP_SET_NEST_LOCK** subroutine sets a nestable lock when it becomes
1726 available (see Section 3.2.3, page 54).
1727 • The **OMP_UNSET_NEST_LOCK** subroutine releases a nestable lock (see Section 3.2.4,
1728 page 55).
1729 • The **OMP_TEST_NEST_LOCK** function tests and possibly sets a nestable lock (see
1730 Section 3.2.5, page 55).

1731
1732 See Section A.15, page 70, and Section A.16, page 71, for examples of using the
simple and the nestable lock routines.

1733 **3.2.1 OMP_INIT_LOCK and OMP_INIT_NEST_LOCK Subroutines**

1734 These subroutines provide the only means of initializing a lock. Each subroutine
1735 initializes a lock associated with the lock variable argument for use in subsequent
1736 calls.

1737 The format of these subroutines is as follows:

1738

SUBROUTINE OMP_INIT_LOCK(*svar*)

1739

SUBROUTINE OMP_INIT_NEST_LOCK(*nvar*)

1740 The initial state is unlocked (that is, no thread owns the lock). For a nestable lock,
1741 the initial nesting count is zero. *svar* must be an uninitialized simple lock variable.
1742 *nvar* must be an uninitialized nestable lock variable. It is noncompliant to call either
1743 of these routines with a lock variable that is already associated with a lock.

1744 **3.2.2 OMP_DESTROY_LOCK and OMP_DESTROY_NEST_LOCK Subroutines**

1745 These subroutines insure that the lock variable is uninitialized and cause the lock
1746 variable to become undefined.

1747 The format for these subroutines is as follows:

1748

SUBROUTINE OMP_DESTROY_LOCK(*svar*)

1749

SUBROUTINE OMP_DESTROY_NEST_LOCK(*nvar*)

1750 *svar* must be an initialized simple lock variable that is unlocked. *nvar* must be an
1751 initialized nestable lock variable that is unlocked.

1752 **3.2.3 OMP_SET_LOCK and OMP_SET_NEST_LOCK Subroutines**

1753 These subroutines force the thread executing the subroutine to wait until the
1754 specified lock is available and then set the lock. A simple lock is available if it is

1755 unlocked. A nestable lock is available if it is unlocked or if it is already owned by the
1756 thread executing the subroutine.

1757 The format of these subroutines is as follows:

```
1758        SUBROUTINE OMP_SET_LOCK(svar)
```

```
1759        SUBROUTINE OMP_SET_NEST_LOCK(nvar)
```

1760 *svar* must be an initialized simple lock variable. Ownership of the lock is granted to
1761 the thread executing the subroutine.

1762 *nvar* must be an initialized nestable lock variable. The nesting count is incremented,
1763 and the thread is granted, or retains, ownership of the lock.

1764 3.2.4 OMP_UNSET_LOCK and OMP_UNSET_NEST_LOCK *Subroutines*

1765 These subroutines provide the means of releasing ownership of a lock.

1766 The format of these subroutines is as follows:

```
1767        SUBROUTINE OMP_UNSET_LOCK(svar)
```

```
1768        SUBROUTINE OMP_UNSET_NEST_LOCK(nvar)
```

1769 The argument to each of these subroutines must be an initialized lock variable owned
1770 by the thread executing the subroutine. The behavior is unspecified if the thread does
1771 not own the lock.

1772 The OMP_UNSET_LOCK subroutine releases the thread executing the subroutine from
1773 ownership of the simple lock associated with *svar*.

1774 The OMP_UNSET_NEST_LOCK subroutine decrements the nesting count and releases
1775 the thread executing the subroutine from ownership of the nestable lock associated
1776 with *nvar* if the resulting count is zero.

1777 3.2.5 OMP_TEST_LOCK and OMP_TEST_NEST_LOCK *Functions*

1778 These functions attempt to set a lock but do not cause the execution of the thread to
1779 wait.

1780 The format of these functions is as follows:

1781

```
LOGICAL FUNCTION OMP_TEST_LOCK(svar)
```

1782

```
INTEGER FUNCTION OMP_TEST_NEST_LOCK(nvar)
```

1783 The argument must be an initialized lock variable. These functions attempt to set a
1784 lock in the same manner as `OMP_SET_LOCK` and `OMP_SET_NEST_LOCK`, except that
1785 they do not cause execution of the thread to wait if the lock is already set.

1786 The `OMP_TEST_LOCK` function returns `.TRUE.` if the simple lock associated with *svar*
1787 is successfully set; otherwise it returns `.FALSE.`.

1788 The `OMP_TEST_NEST_LOCK` function returns the new nesting count if the nestable
1789 lock associated with *nvar* is successfully set; otherwise, it returns zero.

1790 `OMP_TEST_NEST_LOCK` returns a default integer.

1791 3.3 Timing Routines

1792 The OpenMP run-time library includes two routines supporting a portable wall-clock
1793 timer. The routines are as follows:

- 1794 • The `OMP_GET_WTIME` function, described in Section 3.3.1, page 56.
- 1795 • The `OMP_GET_WTICK` function, described in Section 3.3.2, page 57.

1796 3.3.1 `OMP_GET_WTIME` Function

1797 The `OMP_GET_WTIME` function returns a double precision value equal to the elapsed
1798 wallclock time in seconds since some "time in the past". The actual "time in the past"
1799 is arbitrary, but it is guaranteed not to change during the execution of the application
1800 program.

1801 The format of this function is as follows:

1802

```
DOUBLE PRECISION FUNCTION OMP_GET_WTIME()
```

1803 It is anticipated that the function will be used to measure elapsed times as shown in
1804 the following example:

```
1805      DOUBLE PRECISION START, END
1806      START = OMP_GET_WTIME( )
1807      !.... work to be timed
1808      END = OMP_GET_WTIME()
1809      PRINT *, 'Stuff took ', END-START, ' seconds'
```

1810 The times returned are "per-thread times" by which is meant they are not required to
1811 be globally consistent across all the threads participating in an application.

1812 **3.3.2 OMP_GET_WTICK Function**

1813 The OMP_GET_WTICK function returns a double precision value equal to the number
1814 of seconds between successive clock ticks.

1815 The format of this function is as follows:

```
1816      DOUBLE PRECISION FUNCTION OMP_GET_WTICK( )
```


1818 This chapter describes the OpenMP Fortran API environment variables (or
 1819 equivalent platform-specific mechanisms) that control the execution of parallel code.
 1820 The names of environment variables must be uppercase. Character values assigned
 1821 to them are case insensitive and may have leading or trailing white space.

1822 **4.1 OMP_SCHEDULE Environment Variable**

1823 The OMP_SCHEDULE environment variable applies only to DO and PARALLEL DO
 1824 directives that have the schedule type RUNTIME. For more information on the DO
 1825 directive, see Section 2.3.1, page 15. For more information on the PARALLEL DO
 1826 directive, see Section 2.4.1, page 23.

1827 The schedule type and chunk size for all such loops can be set at run time by setting
 1828 this environment variable to any of the recognized schedule types and to an optional
 1829 chunk size. The value takes the form:

1830 `type[, chunk]`

1831 where *type* is one of STATIC, DYNAMIC, or GUIDED (see Table 1, page 17) and *chunk* is
 1832 an optional chunk size. If a chunk size is specified, it must be a positive scalar
 1833 integer. If *chunk* is present, there may be white space on either side of the “,”.

1834 For DO and PARALLEL DO directives that have a schedule type other than RUNTIME,
 1835 this environment variable is ignored. The default value for this environment variable
 1836 is implementation-dependent. If the optional chunk size is not set, a chunk size of 1
 1837 is assumed, except in the case of a STATIC schedule. For a STATIC schedule, the
 1838 default chunk size is set to the loop iteration count divided by the number of threads
 1839 applied to the loop.

1840 Examples:

1841 `setenv OMP_SCHEDULE "GUIDED, 4"`
 1842 `setenv OMP_SCHEDULE "dynamic"`

1843 **4.2 OMP_NUM_THREADS Environment Variable**

1844 The `OMP_NUM_THREADS` environment variable sets the number of threads to use
1845 during execution, unless that number is explicitly changed by calling the
1846 `OMP_SET_NUM_THREADS` library routine. For more information on the
1847 `OMP_SET_NUM_THREADS` library routine, see Section 3.1.1, page 48.

1848 When dynamic adjustment of the number of threads is enabled, the value of this
1849 environment variable is the maximum number of threads to use. The value specified
1850 must be a positive scalar integer. The default value is implementation dependent.
1851 The behavior of the program is implementation-dependent if the requested value of
1852 `OMP_NUM_THREADS` is more than the number of threads an implementation can
1853 support.

1854 Example:

1855

```
setenv OMP_NUM_THREADS 16
```

1856 **4.3 OMP_DYNAMIC Environment Variable**

1857 The `OMP_DYNAMIC` environment variable enables or disables dynamic adjustment of
1858 the number of threads available for execution of parallel regions. For more
1859 information on parallel regions, see Section 2.2, page 12.

1860 If set to `TRUE`, the number of threads that are used for executing parallel regions can
1861 be adjusted by the run-time environment to best utilize system resources.

1862 If set to `FALSE`, dynamic adjustment is disabled. The default value is
1863 implementation-dependent. For more information on the `OMP_SET_DYNAMIC` library
1864 routine, see Section 3.1.7, page 51.

1865 Example:

1866

```
setenv OMP_DYNAMIC TRUE
```

1867 **4.4 OMP_NESTED Environment Variable**1868 The OMP_NESTED environment variable enables or disables nested parallelism. If set
1869 to TRUE, nested parallelism is enabled; if it is set to FALSE, it is disabled. The default
1870 value is FALSE. For more information on nested parallelism, see Section 3.1.9, page
1871 52.

1872 Example:

1873

```
setenv OMP_NESTED TRUE
```


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

A.1 Executing a Simple Loop in Parallel

The following example shows how to parallelize a simple loop using the PARALLEL DO directive (specified in Section 2.4.1, page 23). The loop iteration variable is private by default, so it is not necessary to declare it explicitly.

```
1880 !$OMP PARALLEL DO !I is private by default
1881     DO I=2,N
1882         B(I) = (A(I) + A(I-1)) / 2.0
1883     ENDDO
1884 !$OMP END PARALLEL DO
```

The END PARALLEL DO directive is optional.

A.2 Specifying Conditional Compilation

The following example illustrates the use of the conditional compilation sentinel (specified in Section 2.1.3, page 10). Assuming Fortran fixed source form, the following statement is illegal when using OpenMP constructs:

```
1890 C234567890
1891 !$ X(I) = X(I) + XLOCAL
```

With OpenMP compilation, the conditional compilation sentinel !\$ is treated as two spaces. As a result, the statement infringes on the statement label field. To be legal, the statement should begin after column 6, like any other fixed source form statement:

```
1895 C234567890
1896 !$     X(I) = X(I) + XLOCAL
```

In other words, conditionally compiled statements need to meet all applicable language rules when the sentinel is replaced with two spaces.

1899 **A.3 Using Parallel Regions**

1900 The PARALLEL directive (specified in Section 2.2, page 12) can be used in coarse-grain
 1901 parallel programs. In the following example, each thread in the parallel region
 1902 decides what part of the global array x to work on based on the thread number:

```
1903 !$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,NPOINTS)
1904     IAM = OMP_GET_THREAD_NUM()
1905     NP = OMP_GET_NUM_THREADS()
1906     IPOINTS = NPOINTS/NP
1907     CALL SUBDOMAIN(X,IAM,IPOINTS)
1908 !$OMP END PARALLEL
```

1909 **A.4 Using the NOWAIT Clause**

1910 If there are multiple independent loops within a parallel region, you can use the
 1911 NOWAIT clause (specified in Section 2.3.1, page 15) to avoid the implied BARRIER at
 1912 the end of the DO directive, as follows:

```
1913 !$OMP PARALLEL
1914   !$OMP DO
1915     DO I=2,N
1916       B(I) = (A(I) + A(I-1)) / 2.0
1917     ENDDO
1918   !$OMP END DO NOWAIT
1919   !$OMP DO
1920     DO I=1,M
1921       Y(I) = SQRT(Z(I))
1922     ENDDO
1923   !$OMP END DO NOWAIT
1924 !$OMP END PARALLEL
```

1925 **A.5 Using the CRITICAL Directive**

1926 The following example (for Section 2.5.2, page 26) includes several CRITICAL
 1927 directives. The example illustrates a queuing model in which a task is dequeued and
 1928 worked on. To guard against multiple threads dequeuing the same task, the
 1929 dequeuing operation must be in a critical section. Because there are two independent

queues in this example, each queue is protected by CRITICAL directives with different names, XAXIS and YAXIS, respectively.

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,Y)
!$OMP CRITICAL(XAXIS)
    CALL DEQUEUE(IX_NEXT, X)
!$OMP END CRITICAL(XAXIS)
    CALL WORK(IX_NEXT, X)
!$OMP CRITICAL(YAXIS)
    CALL DEQUEUE(IY_NEXT, Y)
!$OMP END CRITICAL(YAXIS)
    CALL WORK(IY_NEXT, Y)
!$OMP END PARALLEL
```

A.6 Using the LASTPRIVATE Clause

Correct execution sometimes depends on the value that the last iteration of a loop assigns to a variable. Such programs must list all such variables in a `LASTPRIVATE` clause (specified in Section 2.6.2.5, page 38) so that the values of the variables are the same as when the loop is executed sequentially.

```

!$OMP PARALLEL
!$OMP DO LASTPRIVATE(I)
    DO I=1,N
        A(I) = B(I) + C(I)
    ENDDO
!$OMP END PARALLEL
    CALL REVERSE(I)

```

In the preceding example, the value of I at the end of the parallel region will equal $N+1$, as in the sequential case.

A.7 Using the REDUCTION Clause

The following example (for Section 2.6.2.6, page 38) shows how to use the `REDUCTION` clause:

```
!$OMP PARALLEL DO DEFAULT(PRIVATE) REDUCTION(+: A,B)
    DO I=1,N
```

```
1961      CALL WORK(ALOCAL,BLOCAL)
1962      A = A + ALOCAL
1963      B = B + BLOCAL
1964      ENDDO
1965 !$OMP END PARALLEL DO
```

1966 The following program is noncompliant because the reduction is on the
1967 *intrinsic_procedure_name* MAX but that name has been redefined to be the variable
1968 named MAX.

```
1969      MAX = HUGE(0)
1970      M = 0
1971      !$OMP PARALLEL DO REDUCTION(MAX: M) ! MAX is no longer the
1972                               ! intrinsic so this
1973                               ! is invalid
1974      DO I = 1, 100
1975          CALL SUB(M,I)
1976      END DO
1977      END
1978
1979      SUBROUTINE SUB(M,I)
1980          M = MAX(M,I)
1981      END SUBROUTINE SUB
```

1981 The following compliant program performs the reduction using the
1982 *intrinsic_procedure_name* MAX even though the intrinsic MAX has been renamed to
1983 REN.

```
1984      MODULE M
1985          INTRINSIC MAX
1986      END MODULE M
1987      PROGRAM P
1988          USE M, REN => MAX
1989          M = 0
1990          !$OMP PARALLEL DO REDUCTION(REN: M) ! still does MAX
1991          DO I = 1, 100
1992              M = MAX(M,I)
1993          END DO
1994      END PROGRAM P
```

1995 The following compliant program performs the reduction using
1996 *intrinsic_procedure_name* MAX even though the intrinsic MAX has been renamed to
1997 MIN.

```

1998      MODULE MOD
1999          INTRINSIC MAX, MIN
2000      END MODULE MOD
2001      PROGRAM P
2002          USE MOD, MIN=>MAX, MAX=>MIN
2003          REAL :: R
2004          R = -HUGE(0.0)
2005      !$OMP PARALLEL DO REDUCTION(MIN: R) ! still does MAX
2006          DO I = 1, 1000
2007              R = MIN(R, SIN(REAL(I)))
2008          END DO
2009          PRINT *, R
2010      END PROGRAM P

```

2011 A.8 Specifying Parallel Sections

2012 In the following example (for Section 2.3.2, page 18), subroutines XAXIS, YAXIS, and
 2013 ZAXIS can be executed concurrently. The first SECTION directive is optional. Note
 2014 that all SECTION directives need to appear in the lexical extent of the
 2015 PARALLEL SECTIONS/END PARALLEL SECTIONS construct.

```

2016      !$OMP PARALLEL SECTIONS
2017      !$OMP SECTION
2018          CALL XAXIS()
2019      !$OMP SECTION
2020          CALL YAXIS()
2021      !$OMP SECTION
2022          CALL ZAXIS()
2023      !$OMP END PARALLEL SECTIONS

```

2024 A.9 Using SINGLE Directives

2025 The first thread that encounters the SINGLE directive (specified in Section 2.3.3, page
 2026 20) executes subroutines OUTPUT and INPUT. The user must not make any
 2027 assumptions as to which thread will execute the SINGLE section. All other threads
 2028 will skip the SINGLE section and stop at the barrier at the END SINGLE construct. If
 2029 other threads can proceed without waiting for the thread executing the SINGLE
 2030 section, a NOWAIT clause can be specified on the END SINGLE directive.

```

2031      !$OMP PARALLEL DEFAULT( SHARED )
2032          CALL WORK( X )
2033      !$OMP BARRIER
2034      !$OMP SINGLE
2035          CALL OUTPUT( X )
2036          CALL INPUT( Y )
2037      !$OMP END SINGLE
2038          CALL WORK( Y )
2039      !$OMP END PARALLEL

```

2040 A.10 Specifying Sequential Ordering

2041 ORDERED sections (specified in Section 2.5.6, page 30) are useful for sequentially
 2042 ordering the output from work that is done in parallel. Assuming that a reentrant I/O
 2043 library exists, the following program prints out the indexes in sequential order:

```

2044      !$OMP DO ORDERED SCHEDULE(DYNAMIC)
2045          DO I=LB,UB,ST
2046              CALL WORK( I )
2047          END DO
2048          ...
2049          SUBROUTINE WORK( K )
2050      !$OMP ORDERED
2051          WRITE(*,*) K
2052      !$OMP END ORDERED
2053          END

```

2054 A.11 Specifying a Fixed Number of Threads

2055 Some programs rely on a fixed, prespecified number of threads to execute correctly.
 2056 Because the default setting for the dynamic adjustment of the number of threads is
 2057 implementation-dependent, such programs can choose to turn off the dynamic threads
 2058 capability and set the number of threads explicitly to ensure portability. The
 2059 following example (for Section 3.1.1, page 48) shows how to do this:

```

2060          CALL OMP_SET_DYNAMIC(.FALSE.)
2061          CALL OMP_SET_NUM_THREADS(16)
2062      !$OMP PARALLEL DEFAULT(PRIVATE)SHARED(X,NPOINTS)
2063          IAM = OMP_GET_THREAD_NUM()

```

```

2064      IPOINTS = NPOINTS/16
2065      CALL DO_BY_16(X,IAM,IPOINTS)
2066      !$OMP END PARALLEL

```

2067 In this example, the program executes correctly only if it is executed by 16 threads. If
 2068 the implementation is not capable of supporting 16 threads, the behavior of this
 2069 example is implementation-dependent. Note that the number of threads executing a
 2070 parallel region remains constant during a parallel region, regardless of the dynamic
 2071 threads setting. The dynamic threads mechanism determines the number of threads
 2072 to use at the start of the parallel region and keeps it constant for the duration of the
 2073 region.

2074 A.12 Using the ATOMIC Directive

2075 The following example (for Section 2.5.4, page 27) avoids race conditions by protecting
 2076 all simultaneous updates of the location, by multiple threads, with the ATOMIC
 2077 directive:

```

2078 !$OMP PARALLEL DO DEFAULT(PRIVATE) SHARED(X,Y,INDEX,N)
2079     DO I=1,N
2080         CALL WORK(XLOCAL, YLOCAL)
2081     !$OMP ATOMIC
2082         X(INDEX(I)) = X(INDEX(I)) + XLOCAL
2083         Y(I) = Y(I) + YLOCAL
2084     ENDDO

```

2085 Note that the ATOMIC directive applies only to the Fortran statement immediately
 2086 following it. As a result, Y is not updated atomically in this example.

2087 A.13 Using the FLUSH Directive

2088 The following example (for Section 2.5.5, page 29) uses the FLUSH directive for
 2089 point-to-point synchronization between pairs of threads:

```

2090 !$OMP PARALLEL DEFAULT(PRIVATE) SHARED(ISYNC)
2091     IAM = OMP_GET_THREAD_NUM()
2092     ISYNC(IAM) = 0
2093     NEIGH = GET_NEIGHBOR(IAM)
2094     !$OMP BARRIER
2095     CALL WORK()

```

```
2096      C      I am done with my work, synchronize with my neighbor
2097      ISYNC(IAM) = 1
2098      !$OMP FLUSH(ISYNC)
2099      C      Wait until neighbor is done
2100      DO WHILE (ISYNC(NEIGH) .EQ. 0)
2101      !$OMP FLUSH(ISYNC)
2102      END DO
2103      !$OMP END PARALLEL
```

2104 **A.14 Determining the Number of Threads Used**

2105 Consider the following incorrect example:

```
2106      NP = OMP_GET_NUM_THREADS()
2107      !$OMP PARALLEL DO SCHEDULE(STATIC)
2108      DO I = 0, NP-1
2109      CALL WORK(I)
2110      ENDDO
2111      !$OMP END PARALLEL DO
```

2112 The `OMP_GET_NUM_THREADS` call (specified in Section 3.1.2, page 48) returns 1 in the
2113 serial section of the code, so `NP` will always be equal to 1 in the preceding example. To
2114 determine the number of threads that will be deployed for the parallel region, the call
2115 should be inside the parallel region.

2116 The following example shows how to rewrite this program without including a query
2117 for the number of threads:

```
2118      !$OMP PARALLEL PRIVATE(I)
2119      I = OMP_GET_THREAD_NUM()
2120      CALL WORK(I)
2121      !$OMP END PARALLEL
```

2122 **A.15 Using Locks**

2123 This is an example of the use of the simple lock routines (specified in Section 3.2,
2124 page 52).

2125 **In the following program, note that the argument to the lock routines should be of**
 2126 **type INTEGER and of a KIND large enough to hold an address:**

```

2127      PROGRAM LOCK_USAGE
2128      EXTERNAL OMP_TEST_LOCK
2129      LOGICAL OMP_TEST_LOCK

2130      INTEGER LCK          ! This variable should be pointer sized

2131      CALL OMP_INIT_LOCK(LCK)
2132      !$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
2133      ID = OMP_GET_THREAD_NUM()
2134      CALL OMP_SET_LOCK(LCK)
2135      PRINT *, 'MY THREAD ID IS ', ID
2136      CALL OMP_UNSET_LOCK(LCK)

2137      DO WHILE (.NOT. OMP_TEST_LOCK(LCK))
2138          CALL SKIP(ID)      ! We do not yet have the lock
2139                      ! so we must do something else
2140      END DO

2141          CALL WORK(ID)      ! We now have the lock
2142                      ! and can do the work
2143          CALL OMP_UNSET_LOCK( LCK )
2144      !$OMP END PARALLEL

2145          CALL OMP_DESTROY_LOCK( LCK )

2146      END

```

2147 A.16 Using Nestable Locks

2148 The following example shows how a nestable lock (specified in Section 3.2, page 52)
 2149 can be used to synchronize updates both to a structure and to one of its components.

```

2150      MODULE DATA
2151      USE OMP_LIB, ONLY OMP_NEXT_LOCK_KIND

2152      TYPE LOCKED_PAIR
2153          INTEGER A
2154          INTEGER B
2155          INTEGER (OMP_NEST_LOCK_KIND) LCK

```

```
2155      END TYPE
2157      END MODULE DATA

2158
2159      SUBROUTINE INCR_A(P, A)
2160          ! called only from INCR_PAIR, no need to lock
2161          USE DATA
2162          TYPE(LOCKED_PAIR) :: P
2163          INTEGER A
2164
2165          P%A = P%A + A
2166      END SUBROUTINE INCR_A

2167
2168      SUBROUTINE INCR_B(P, B)
2169          ! called from both INCR_PAIR and elsewhere,
2170          ! so we need a nestable lock
2171          USE OMP_LIB
2172          USE DATA
2173          TYPE(LOCKED_PAIR) :: P
2174          INTEGER B
2175
2176          CALL OMP_SET_NEST_LOCK(P%LCK)
2177          P%B = P%B + B
2178          CALL OMP_UNSET_NEST_LOCK(P%LCK)
2179      END SUBROUTINE INCR_B

2180
2181      SUBROUTINE INCR_PAIR(P, A, B)
2182          USE OMP_LIB
2183          USE DATA
2184          TYPE(LOCKED_PAIR) :: P
2185          INTEGER A
2186          INTEGER B
2187
2188          CALL OMP_SET_NEST_LOCK(P%LCK)
2189          CALL INCR_A(P, A)
2190          CALL INCR_B(P, B)
2191          CALL OMP_UNSET_NEST_LOCK(P%LCK)
2192      END SUBROUTINE INCR_PAIR

2193
2194      SUBROUTINE F(P)
2195          USE OMP_LIB
2196          USE DATA
2197          TYPE(LOCKED_PAIR) :: P
2198          INTEGER WORK1, WORK2, WORK3
2199          EXTERNAL WORK1, WORK2, WORK3
```

```

2193      !$OMP PARALLEL SECTIONS
2194      !$OMP SECTION
2195          CALL INCR_PAIR(P, WORK1, WORK2)
2196      !$OMP SECTION
2197          CALL INCR_B(P, WORK3)
2198      !$OMP END PARALLEL SECTIONS
2199          END SUBROUTINE F

```

2200 A.17 Nested DO Directives

2201 The following example of directive nesting (specified in Section 2.8, page 45) is
 2202 compliant because the inner and outer DO directives bind to different PARALLEL
 2203 regions:

```

2204 !$OMP PARALLEL DEFAULT(SHARED)
2205 !$OMP DO
2206     DO I = 1, N
2207     !$OMP PARALLEL SHARED(I,N)
2208     !$OMP DO
2209         DO J = 1, N
2210             CALL WORK(I,J)
2211         END DO
2212     !$OMP END PARALLEL
2213     END DO
2214 !$OMP END PARALLEL

```

2215 The following variation of the preceding example is also compliant:

```

2216 !$OMP PARALLEL DEFAULT(SHARED)
2217 !$OMP DO
2218     DO I = 1, N
2219         CALL SOME_WORK(I,N)
2220     END DO
2221 !$OMP END PARALLEL

```

```
2222      SUBROUTINE SOME_WORK(I,N)
2223      !$OMP PARALLEL DEFAULT(SHARED)
2224      !$OMP DO
2225          DO J = 1, N
2226              CALL WORK(I,J)
2227          END DO
2228      !$OMP END PARALLEL
2229      RETURN
2230      END
```

2231 A.18 Examples Showing Incorrect Nesting of Work-sharing Directives

2232 The examples in this section illustrate the directive nesting rules (specified in Section
2233 2.8, page 45).

2234 The following example is noncompliant because the inner and outer DO directives are
2235 nested and bind to the same PARALLEL directive:

2236 Example 1: Noncompliant Example

```
2237      !$OMP PARALLEL DEFAULT(SHARED)
2238      !$OMP DO
2239          DO I = 1, N
2240      !$OMP DO
2241          DO J = 1, N
2242              CALL WORK(I,J)
2243          END DO
2244      END DO
2245      !$OMP END PARALLEL
2246      END
```

2247 The following dynamically nested version of the preceding example is also
2248 noncompliant:

2249 **Example 2: Noncompliant Example**

```
2250 !$OMP PARALLEL DEFAULT(SHARED)
2251 !$OMP DO
2252     DO I = 1, N
2253         CALL SOME_WORK(I,N)
2254     END DO
2255 !$OMP END PARALLEL
2256     END
2257     SUBROUTINE SOME_WORK(I,N)
2258 !$OMP DO
2259     DO J = 1, N
2260         CALL WORK(I,J)
2261     END DO
2262     RETURN
2263 END
```

2264 The following example is noncompliant because the `DO` and `SINGLE` directives are
2265 nested, and they bind to the same `PARALLEL` region:

2266 **Example 3: Noncompliant Example**

```
2267 !$OMP PARALLEL DEFAULT(SHARED)
2268 !$OMP DO
2269     DO I = 1, N
2270 !$OMP SINGLE
2271     CALL WORK(I)
2272 !$OMP END SINGLE
2273     END DO
2274 !$OMP END PARALLEL
2275 END
```

2276 The following example is noncompliant because a `BARRIER` directive inside a `SINGLE`
2277 or a `DO` can result in deadlock:

2278 **Example 4: Noncompliant Example**

```
2279      !$OMP PARALLEL DEFAULT( SHARED )
2280      !$OMP DO
2281          DO I = 1, N
2282              CALL WORK( I )
2283      !$OMP BARRIER
2284          CALL MORE_WORK( I )
2285      END DO
2286      !$OMP END PARALLEL
2287          END
```

2288 **The following example is noncompliant because the BARRIER results in deadlock since**
2289 **only one thread at a time can enter the CRITICAL section:**

2290 **Example 5: Noncompliant Example**

```
2291      !$OMP PARALLEL DEFAULT( SHARED )
2292      !$OMP CRITICAL
2293          CALL WORK( N, 1 )
2294      !$OMP BARRIER
2295          CALL MORE_WORK( N, 2 )
2296      !$OMP END CRITICAL
2297      !$OMP END PARALLEL
2298          END
```

2299 **The following example is noncompliant because the BARRIER results in deadlock since**
2300 **only one thread executes the SINGLE section:**

2301 **Example 6: Noncompliant Example**

```
2302      !$OMP PARALLEL DEFAULT( SHARED )
2303          CALL SETUP( N )
2304      !$OMP SINGLE
2305          CALL WORK( N, 1 )
2306      !$OMP BARRIER
2307          CALL MORE_WORK( N, 2 )
2308      !$OMP END SINGLE
2309          CALL FINISH( N )
2310      !$OMP END PARALLEL
2311          END
```

2312 **A.19 Binding of BARRIER Directives**2313 The directive binding rules call for a BARRIER directive to bind to the closest
2314 enclosing PARALLEL directive. For more information, see Section 2.7, page 45.2315 In the following example, the call from MAIN to SUB2 is OpenMP-compliant because
2316 the BARRIER (in SUB3) binds to the PARALLEL region in SUB2. The call from MAIN to
2317 SUB1 is OpenMP-compliant because the BARRIER binds to the PARALLEL region in
2318 subroutine SUB2.2319 The call from MAIN to SUB3 is OpenMP-compliant because the BARRIER does not bind
2320 to any parallel region and is ignored. Also note that the BARRIER only synchronizes
2321 the team of threads in the enclosing parallel region and not all the threads created in
2322 SUB1.

```
2323      PROGRAM MAIN
2324      CALL SUB1(2)
2325      CALL SUB2(2)
2326      CALL SUB3(2)
2327      END

2328      SUBROUTINE SUB1(N)
2329      !$OMP PARALLEL PRIVATE(I) SHARED(N)
2330      !$OMP DO
2331          DO I = 1, N
2332              CALL SUB2(I)
2333          END DO
2334      !$OMP END PARALLEL
2335      END

2336      SUBROUTINE SUB2(K)
2337      !$OMP PARALLEL SHARED(K)
2338          CALL SUB3(K)
2339      !$OMP END PARALLEL
2340      END

2341      SUBROUTINE SUB3(N)
2342          CALL WORK(N)
2343      !$OMP BARRIER
2344          CALL WORK(N)
2345      END
```

2346 A.20 Scoping Variables with the PRIVATE Clause

2347 The values of `I` and `J` in the following example are undefined on exit from the
2348 parallel region:

```
2349      INTEGER I,J
2350      I = 1
2351      J = 2
2352      !$OMP PARALLEL PRIVATE(I) FIRSTPRIVATE(J)
2353          I = 3
2354          J = J+ 2
2355      !$OMP END PARALLEL
2356      PRINT *, I, J
```

2357 (For more information, see Section 2.6.2.1, page 35.)

2358 A.21 Examples of Noncompliant Storage Association

2359 The following examples illustrate the implications of the PRIVATE clause rules (see
2360 Section 2.6.2.1, page 35, rule 4) with regard to storage association:

2361 Example 1: Noncompliant Example

```
2362      COMMON /BLOCK/ X
2363      X = 1.0
2364      !$OMP PARALLEL PRIVATE (X)
2365          X = 2.0
2366          CALL SUB()
2367          ...
2368      !$OMP END PARALLEL
2369          ...
2370          SUBROUTINE SUB()
2371          COMMON /BLOCK/ X
2372          ...
2373          PRINT *,X           ! X is undefined
2374          ...
2375      END SUBROUTINE SUB
2376      END PROGRAM
```

2377

Example 2: Noncompliant Example

```

2378      COMMON /BLOCK/ X
2379      X = 1.0
2380      !$OMP PARALLEL PRIVATE (X)
2381      X = 2.0
2382      CALL SUB( )
2383      ...
2384      !$OMP END PARALLEL
2385      ...
2386      CONTAINS
2387      SUBROUTINE SUB( )
2388      COMMON /BLOCK/ Y
2389      ...
2390      PRINT *,X           ! X is undefined
2391      PRINT *,Y           ! Y is undefined
2392      ...
2393      END SUBROUTINE SUB
2394      END PROGRAM

```

2395

Example 3: Noncompliant Example

```

2396      EQUIVALENCE (X,Y)
2397      X = 1.0
2398      !$OMP PARALLEL PRIVATE(X)
2399      ...
2400      PRINT *,Y           ! Y is undefined
2401      Y = 10
2402      PRINT *,X           ! X is undefined
2403      !$OMP END PARALLEL

```

2404

Example 4: Noncompliant Example

```

2405      INTEGER A(100), B(100)
2406      EQUIVALENCE (A(51), B(1))

2407      !$OMP PARALLEL DO DEFAULT(PRIVATE) PRIVATE(I,J) LASTPRIVATE(A)
2408      DO I=1,100
2409      DO J=1,100
2410      B(J) = J - 1
2411      ENDDO

2412      DO J=1,100
2413      A(J) = J           ! B becomes undefined at this point

```

```
2414           ENDDO
2415           DO J=1,50
2416             B(J) = B(J) + 1 ! B is undefined
2417                         ! A becomes undefined at this point
2418           ENDDO
2419           ENDDO
2420 !$OMP END PARALLEL DO      ! The LASTPRIVATE write for A has
2421                         ! undefined results
2422           PRINT *, B          ! B is undefined since the LASTPRIVATE
2423                         ! write of A was not defined
2424           END
```

Example 5: Noncompliant Example

```
2425           COMMON /FOO/ A
2426           DIMENSION B(10)
2427           EQUIVALENCE (A,B(1))
2428             ! the common block has to be at least 10 words
2429             A = 0
2430 !$OMP PARALLEL PRIVATE(/FOO/)
2431           !
2432             ! Without the private clause,
2433             ! we would be passing a member of a sequence
2434             ! that is at least ten elements long. With the private
2435             ! clause, A may no longer be sequence-associated.
2436             !
2437             CALL BAR(A)
2438 !$OMP MASTER
2439             PRINT *, A
2440 !$OMP END MASTER
2441 !$OMP END PARALLEL
2442           END
2443
2444           SUBROUTINE BAR(X)
2445           DIMENSION X(10)
2446             !
2447             ! This use of X does not conform to the specification.
2448             ! It would be legal Fortran 90, but the OpenMP private
2449             ! directive allows the compiler to break the sequence
2450             ! association that A had with the rest of the common block.
2451             !
2452             FORALL (I = 1:10) X(I) = I
2453           END
```

2454 A.22 Examples of Syntax of Parallel DO Loops

2455 Both block-do and non-block-do are permitted with PARALLEL DO and work-sharing
2456 DO directives. However, if a user specifies an ENDDO directive for a non-block-do
2457 construct with shared termination, then the matching DO directive must precede the
2458 outermost DO. For more information, see Section 2.3.1, page 15, and Section 2.4.1,
2459 page 23.

2460 The following are some examples:

2461 Example 1:

```
2462      DO 100 I = 1,10
2463      !$OMP DO
2464          DO 100 J = 1,10
2465          ...
2466 100    CONTINUE
```

2467 Example 2:

```
2468 !$OMP DO
2469     DO 100 J = 1,10
2470     ...
2471 100     A(I) = I + 1
2472 !$OMP ENDDO
```

2473 Example 3:

```
2474 !$OMP DO
2475     DO 100 I = 1,10
2476         DO 100 J = 1,10
2477         ...
2478 100    CONTINUE
2479 !$OMP ENDDO
```

2480 Example 4: Noncompliant Example

```
2481      DO 100 I = 1,10
2482      !$OMP DO
2483          DO 100 J = 1,10
2484          ...
2485 100    CONTINUE
2486 !$OMP ENDDO
```

2487 A.23 Examples of the ATOMIC Directive

2488 All atomic references to the storage location of each variable that appears on the
2489 left-hand side of an ATOMIC assignment statement throughout the program are
2490 required to have the same type and type parameters. For more information, see
2491 Section 2.5.4, page 27.

2492 The following are some examples:

2493 Example 1: Noncompliant Example

```
2494      INTEGER:: I
2495      REAL:: R
2496      EQUIVALENCE(I,R)
2497      !$OMP PARALLEL
2498      ...
2499      !$OMP ATOMIC
2500      I = I + 1
2501      ...
2502      !$OMP ATOMIC
2503      R = R + 1.0
2504      !$OMP END PARALLEL
```

2505 Example 2: Noncompliant Example

```
2506      SUBROUTINE FRED()
2507      COMMON /BLK/ I
2508      INTEGER I
2509      !$OMP PARALLEL
2510      ...
2511      !$OMP ATOMIC
2512      I = I + 1
2513      ...
2514      CALL SUB()
2515      !$OMP END PARALLEL
2516      END
2517
2518      SUBROUTINE SUB()
2519      COMMON /BLK/ R
2520      REAL R
2521      ...
2522      !$OMP ATOMIC
2523      R = R + 1
2524      END
```

2524 **Example 3: Noncompliant Example**

2525 Although the following example might work on some implementation, this is
2526 considered a noncompliant example.

```
2527                   INTEGER:: I
2528                   REAL:: R
2529                   EQUIVALENCE(I,R)
2530                   !$OMP PARALLEL
2531                   ...
2532                   !$OMP ATOMIC
2533                   I = I + 1
2534                   !$OMP END PARALLEL
2535                   ...
2536                   !$OMP PARALLEL
2537                   ...
2538                   !$OMP ATOMIC
2539                   R = R + 1.0
2540                   !$OMP END PARALLEL
```

2541 **A.24 Examples of the ORDERED Directive**

2542 It is possible to have multiple ORDERED sections within a DO specified with the
2543 ORDERED clause. Example 1 is noncompliant, because the API states the following:

2544 An iteration of a loop with a DO directive must not execute the same
2545 ORDERED directive more than once, and it must not execute more than one
2546 ORDERED directive.

2547 For more information, see Section 2.5.6, page 30.

2548 **Example 1: Noncompliant Example**

2549 **In this example, all iterations execute 2 ORDERED sections:**

```
2550           !$OMP DO
2551                DO I = 1, N
2552                ...
2553            !$OMP ORDERED
2554                ...
2555            !$OMP END ORDERED
2556                ...
2557            !$OMP ORDERED
2558                ...
2559            !$OMP END ORDERED
2560                ...
2561                END DO
```

2562 **Example 2:**

2563 **This is a compliant example of a DO with more than one ORDERED section:**

```
2564           !$OMP DO ORDERED
2565                DO I = 1, N
2566                ...
2567                IF (I <= 10) THEN
2568                ...
2569            !$OMP ORDERED
2570                WRITE(4,*) I
2571            !$OMP END ORDERED
2572                ENDIF
2573                ...
2574                IF (I > 10) THEN
2575                ...
2576            !$OMP ORDERED
2577                WRITE(3,*) I
2578            !$OMP END ORDERED
2579                ENDIF
2580                ENDDO
```

2581 **A.25 Examples of THREADPRIVATE Data**

2582 The following examples show noncompliant uses and correct uses of the
2583 THREADPRIVATE directive. For more information, see Section 2.6.1, page 32, item 8 of
2584 Section 2.6.3, page 42, and Section 2.6.2.7, page 41.

2585 **Example 1: Noncompliant Example**

```
2586                   MODULE FOO
2587                   COMMON /T/ A
2588                   END MODULE FOO

2589                   SUBROUTINE BAR( )
2590                   USE FOO
2591                   !$OMP THREADPRIVATE(/T/)
2592                   !noncompliant because /T/ not declared in BAR
2593                   !See Section 2.6.1
2594                   !$OMP PARALLEL
2595                   ...
2596                   !$OMP END PARALLEL
2597                   END SUBROUTINE BAR
```

2598 **Example 2: Noncompliant Example**

```
2599                   COMMON /T/ A
2600                   !$OMP THREADPRIVATE(/T/)
2601                   ...
2602                   CONTAINS
2603                    SUBROUTINE BAR( )
2604                   !$OMP PARALLEL COPYIN(/T/)
2605                   !noncompliant because /T/ not declared in BAR
2606                   !See Section 2.6.2.7
2607                   ...
2608                   !$OMP END PARALLEL
2609                   END SUBROUTINE BAR
2610                   END PROGRAM
```

2611 **Example 3: Correct Rewrite of the Previous Example**

```
2612                   COMMON /T/ A
2613                   !$OMP THREADPRIVATE(/T/)
2614                   ...
2615                   CONTAINS
2616                    SUBROUTINE BAR( )
2617                    COMMON /T/ A
2618                   !$OMP THREADPRIVATE(/T/)
2619                   !$OMP PARALLEL COPYIN(/T/)
2620                   ...
2621                   !$OMP END PARALLEL
2622                   END SUBROUTINE BAR
2623                   END PROGRAM
```

2624 **Example 4: An example of THREADPRIVATE for local variables**

```
2625      PROGRAM P
2626      INTEGER, ALLOCATABLE, SAVE :: A(:)
2627      INTEGER, POINTER, SAVE :: PTR
2628      INTEGER, SAVE :: I
2629      INTEGER, TARGET :: TARG
2630      LOGICAL :: FIRSTIN = .TRUE.
2631      !$OMP THREADPRIVATE(A, B, I, PTR)

2632      ALLOCATE (A(3))
2633      A = (/1,2,3/)
2634      PTR => TARG
2635      I = 5

2636      !$OMP PARALLEL COPYIN(I, PTR)
2637      !$OMP CRITICAL
2638          IF (FIRSTIN) THEN
2639              TARG = 4           ! Update target of ptr
2640              I = I + 10
2641              IF (ALLOCATED(A)) A = A + 10
2642              FIRSTIN = .FALSE.
2643          END IF
2644          IF (ALLOCATED(A)) THEN
2645              PRINT *, 'a = ', A
2646          ELSE
2647              PRINT *, 'A is not allocated'
2648          END IF
2649          PRINT *, 'ptr = ', PTR
2650          PRINT *, 'i = ', I
2651          PRINT *
2652      !$OMP END CRITICAL
2653      !$OMP END PARALLEL
2654      END PROGRAM P
```

2655 This program, if executed by two threads, will print the following.

```
2656      a = 11 12 13
2657      ptr = 4
2658      i = 15

2659      A is not allocated
2660      ptr = 4
2661      i = 5

2662      or
```

```
2663          A is not allocated
2664          ptr = 4
2665          i = 15

2666          a = 1 2 3
2667          ptr = 4
2668          i = 5
```

2669 Example 5: An example of THREADPRIVATE for module variables

```
2670      MODULE FOO
2671          REAL, POINTER :: WORK(:)
2672          SAVE WORK
2673          !$OMP THREADPRIVATE(WORK)
2674      END MODULE FOO

2675      SUBROUTINE SUB1(N)
2676          USE FOO
2677          !$OMP PARALLEL PRIVATE(THE_SUM)
2678              ALLOCATE(WORK(N))
2679              CALL SUB2(N,THE_SUM)
2680              WRITE(*,*)THE_SUM
2681          !$OMP END PARALLEL
2682          END SUBROUTINE SUB1

2683      SUBROUTINE SUB2(N,THE_SUM)
2684          USE FOO
2685          WORK = 10
2686          THE_SUM=SUM(WORK)
2687      END SUBROUTINE SUB2

2688      PROGRAM BONK
2689          USE FOO
2690          N = 10
2691          CALL SUB1(N)
2692      END PROGRAM BONK
```

2693 **A.26 Examples of the Data Attribute Clauses: SHARED and PRIVATE**

2694 When a named common block is specified in a PRIVATE, FIRSTPRIVATE, or
2695 LASTPRIVATE clause of a directive, none of its constituent elements may be declared
2696 in another scope attribute clause in that directive. The following examples, both
2697 compliant and noncompliant, illustrate this point. For more information, see item 8 of
2698 Section 2.6.3, page 42.

2699 **Example 1:**

```
2700      COMMON /C/ X,Y
2701      !$OMP PARALLEL PRIVATE (/C/)
2702      ...
2703      !$OMP END PARALLEL
2704      ...
2705      !$OMP PARALLEL SHARED (X,Y)
2706      ...
2707      !$OMP END PARALLEL
```

2708 **Example 2:**

```
2709      COMMON /C/ X,Y
2710      !$OMP PARALLEL
2711      ...
2712      !$OMP DO PRIVATE(/C/)
2713      ...
2714      !$OMP END DO
2715      !
2716      !$OMP DO PRIVATE(X)
2717      ...
2718      !$OMP END DO
2719      ...
2720      !$OMP END PARALLEL
```

2721 **Example 3: Noncompliant Example**

```
2722      COMMON /C/ X,Y
2723      !$OMP PARALLEL PRIVATE(/C/), SHARED(X)
2724      ...
2725      !$OMP END PARALLEL
```

2726

Example 4:

```

2727           COMMON /C/ X,Y
2728           !$OMP PARALLEL PRIVATE (/C/)
2729           ...
2730           !$OMP END PARALLEL
2731           ...
2732           !$OMP PARALLEL SHARED (/C/)
2733           ...
2734           !$OMP END PARALLEL

```

2735

Example 5: Noncompliant Example

```

2736           COMMON /C/ X,Y
2737           !$OMP PARALLEL PRIVATE(/C/), SHARED(/C/)
2738           ...
2739           !$OMP END PARALLEL

```

2740

Example 6:

```

2741           MODULE M
2742           REAL A
2743           CONTAINS
2744           SUBROUTINE SUB
2745           !$OMP PARALLEL PRIVATE(A)
2746           CALL SUB1()
2747           !$OMP END PARALLEL
2748           END SUBROUTINE SUB
2749           SUBROUTINE SUB1()
2750           A = 5    ! This is A in module M, not the PRIVATE
2751           ! A in SUB
2752           END SUBROUTINE SUB1
2753           END MODULE M

```

2754

A.27 Examples of the Data Attribute Clause: COPYPRIVATE

2755
2756
2757

Example 1. The COPYPRIVATE clause (specified in Section 2.6.2.8, page 41) can be used to broadcast the value resulting from a read statement directly to all instances of a private variable.

2758

```
SUBROUTINE INIT(A,B)
```

```

2759      COMMON /XY/ X,Y
2760      !$OMP THREADPRIVATE (/XY/)
2761      !$OMP SINGLE
2762          READ (11) A,B,X,Y
2763      !$OMP END SINGLE COPYPRIVATE (A,B,/XY/)
2764      END

```

2765 If subroutine `INIT` is called from a serial region, its behavior is not affected by the
 2766 presence of the directives. If it is called from a parallel region, then the actual
 2767 arguments with which `A` and `B` are associated must be private. After the read
 2768 statement has been executed by one thread, no thread leaves the construct until the
 2769 private objects designated by `A`, `B`, `X`, and `Y` in all threads have become defined with
 2770 the values read.

2771 Example 2. In contrast to the previous example, suppose the read must be performed
 2772 by a particular thread, say the master thread. In this case, the `COPYPRIVATE` clause
 2773 cannot be used to do the broadcast directly, but it can be used to provide access to a
 2774 temporary shared object.

```

2775      REAL FUNCTION READ_NEXT()
2776          REAL, POINTER :: TMP
2777      !$OMP SINGLE
2778          ALLOCATE (TMP)
2779      !$OMP END SINGLE COPYPRIVATE (TMP)

2780      !$OMP MASTER
2781          READ (11) TMP
2782      !$OMP END MASTER

2783      !$OMP BARRIER
2784          READ_NEXT = TMP
2785      !$OMP BARRIER

2786      !$OMP SINGLE
2787          DEALLOCATE (TMP)
2788      !$OMP END SINGLE NOWAIT
2789      END FUNCTION READ_NEXT

```

2790 Example 3. Suppose that the number of lock objects required within a parallel region
 2791 cannot easily be determined prior to entering it. The `COPYPRIVATE` clause can be used
 2792 to provide access to shared lock objects that are allocated within that parallel region.

```

2793      FUNCTION NEW_LOCK()
2794          INTEGER(OMP_LOCK_KIND), POINTER :: NEW_LOCK

```

```

2795      !$OMP SINGLE
2796          ALLOCATE(NEW_LOCK)
2797          CALL OMP_INIT_LOCK(NEW_LOCK)
2798      !$OMP END SINGLE COPYPRIVATE(NEW_LOCK)
2799  END FUNCTION NEW_LOCK

```

2800 **Example 4. Note that the effect of the copyprivate clause on a variable with the**
 2801 **allocatable attribute is different than on a variable with the pointer attribute.**

```

2802      SUBROUTINE S(N)
2803          REAL, DIMENSION(:), ALLOCATABLE :: A
2804          REAL, DIMENSION(:), POINTER :: B
2805          ALLOCATE (A(N))
2806      !$OMP SINGLE
2807          ALLOCATE (B(N))
2808          READ (11) A,B
2809      !$OMP END SINGLE COPYPRIVATE(A,B)
2810          ! Variable A designates a private object
2811          ! which has the same value in each thread
2812          ! Variable B designates a shared object
2813          ...
2814      !$OMP BARRIER
2815      !$OMP SINGLE
2816          DEALLOCATE (B)
2817      !$OMP END SINGLE NOWAIT
2818  END SUBROUTINE S

```

2819 **A.28 Examples of the WORKSHARE Directive**

2820 In the following examples of the WORKSHARE directive (specified in Section 2.3.4, page
 2821 20), assume that all 2 letter variable names (e.g., AA, BB) are conformable arrays and
 2822 single letter names (e.g., I, X) are scalars; implicit typing rules hold. Each of the
 2823 examples is enclosed in a parallel region. All of the examples are fixed source form so
 2824 the directives start in column 1.

2825 **Example 1. WORKSHARE spreads work across some number of threads and there is a**
 2826 **barrier after the last statement. Implementations must enforce Fortran execution**
 2827 **rules inside of the WORKSHARE block.**

```

2828      !$OMP WORKSHARE
2829          AA = BB

```

```

2830      CC = DD
2831      EE = FF
2832 !$OMP END WORKSHARE

```

Example 2. The final barrier can be eliminated with NOWAIT:

```

2834 !$OMP WORKSHARE
2835      AA = BB
2836      CC = DD
2837 !$OMP END WORKSHARE NOWAIT

2838 !$OMP WORKSHARE
2839      EE = FF
2840 !$OMP END WORKSHARE

```

Threads doing `CC = DD` immediately begin work on `EE = FF` when they are done with `CC = DD`.

Example 3. ATOMIC can be used with WORKSHARE:

```

!$OMP WORKSHARE
      AA = BB
!$OMP ATOMIC
      I = I + SUM(AA)
      CC = DD
!$OMP END WORKSHARE

```

The computation of `SUM(AA)` is workshared, but the update to `I` is ATOMIC.

Example 4. Fortran WHERE and FORALL statements are *compound statements* of the form:

```

WHERE (EE .ne. 0) FF = 1 / EE
FORALL (I=1:N, XX(I) .ne. 0) YY(I) = 1 / XX(I)

```

They are made up of a *control* part and a *statement* part. When WORKSHARE is applied to one of these compound statements, both the *control* and the *statement* parts are workshared.

```

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

```

Each task gets worked on in order by the threads:

```
2865          AA = BB      then
2866          CC = DD      then
2867          EE .ne. 0      then
2868          FF = 1 / EE then
2869          GG = HH
```

2870 **Example 5.** An assignment to a shared scalar variable is performed by one thread in
2871 a WORKSHARE while all other threads in the team wait. SHR is a shared scalar
2872 variable in this example.

```
2873      !$OMP WORKSHARE
2874          AA = BB
2875          SHR = 1
2876          CC = DD
2877      !$OMP END WORKSHARE
```

2878 **Noncompliant Example 6.** An assignment to a private scalar variable is performed by
2879 one thread in a WORKSHARE while all other threads wait. The private scalar variable
2880 is undefined after the assignment statement. PRI is a private scalar variable in this
2881 example.

```
2882      !$OMP WORKSHARE
2883          AA = BB
2884          PRI = 1
2885          CC = DD
2886      !$OMP END WORKSHARE
```

2887 **Example 7.** Fortran execution rules must be enforced inside a WORKSHARE construct.
2888 Hence, the same result is produced in the following program fragment regardless of
2889 whether the code is executed sequentially or inside an OpenMP program with
2890 multiple threads:

```
2891      !$OMP WORKSHARE
2892          A(1:50) = B(11:60)
2893          G(11:20) = A(1:10)
2894      !$OMP END WORKSHARE
```


Stubs for Run-time Library Routines [B]

2895

2896
2897
2898
2899
2900
2901

This section provides stubs for the runtime library routines defined in the OpenMP Fortran API. The stubs are provided to enable portability to platforms that do not support the OpenMP Fortran API. On such 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.

2902
2903
2904
2905
2906
2907
2908
2909
2910
2911

Note: 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. It is declared as a `POINTER` to guarantee that it is capable of holding an address. Alternatively, for Fortran 90 implementations, it could be declared as an `INTEGER(OMP_LOCK_KIND)` or `INTEGER(OMP_NEST_LOCK_KIND)`, as appropriate. In an actual implementation the lock variable might be used to hold the address of an allocated object, but here it is used to hold an integer value. Users should not make assumptions about mechanisms used by OpenMP Fortran implementations to implement locks based on the scheme used by the stub routines.

2912 SUBROUTINE OMP_SET_NUM_THREADS(NP)
2913 INTEGER NP
2914 RETURN
2915 END

2916 INTEGER FUNCTION OMP_GET_NUM_THREADS()
2917 OMP_GET_NUM_THREADS = 1
2918 RETURN
2919 END

2920 INTEGER FUNCTION OMP_GET_MAX_THREADS()
2921 OMP_GET_MAX_THREADS = 1
2922 RETURN
2923 END

2924 INTEGER FUNCTION OMP_GET_THREAD_NUM()
2925 OMP_GET_THREAD_NUM = 0
2926 RETURN
2927 END

2928 INTEGER FUNCTION OMP_GET_NUM_PROCS()
2929 OMP_GET_NUM_PROCS = 1
2930 RETURN
2931 END

```
2932      LOGICAL FUNCTION OMP_IN_PARALLEL( )
2933      OMP_IN_PARALLEL = .FALSE.
2934      RETURN
2935      END

2936      SUBROUTINE OMP_SET_DYNAMIC(FLAG)
2937      LOGICAL FLAG
2938      RETURN
2939      END

2940      LOGICAL FUNCTION OMP_GET_DYNAMIC( )
2941      OMP_GET_DYNAMIC = .FALSE.
2942      RETURN
2943      END

2944      SUBROUTINE OMP_SET_NESTED(FLAG)
2945      LOGICAL FLAG
2946      RETURN
2947      END

2948      LOGICAL FUNCTION OMP_GET_NESTED( )
2949      OMP_GET_NESTED = .FALSE.
2950      RETURN
2951      END

2952      SUBROUTINE OMP_INIT_LOCK(LOCK)
2953      ! LOCK is 0 if the simple lock is not initialized
2954      !           -1 if the simple lock is initialized but not set
2955      !           1 if the simple lock is set
2956      POINTER (LOCK,IL)
2957      INTEGER IL
2958      LOCK = -1
2959      RETURN
2960      END

2961      SUBROUTINE OMP_INIT_NEST_LOCK(NLOCK)
2962      ! NLOCK is 0 if the nestable lock is not initialized
2963      !           -1 if the nestable lock is initialized but not set
2964      !           1 if the nestable lock is set
2965      ! no use count is maintained
2966      POINTER (NLOCK,NIL)
2967      INTEGER NIL
2968      NLOCK = -1
2969      RETURN
2970      END
```

```
2971      SUBROUTINE OMP_DESTROY_LOCK(LOCK)
2972      POINTER (LOCK,IL)
2973      INTEGER IL
2974      LOCK = 0
2975      RETURN
2976      END

2977      SUBROUTINE OMP_DESTROY_NEST_LOCK(NLOCK)
2978      POINTER (NLOCK,NIL)
2979      INTEGER NIL
2980      NLOCK = 0
2981      RETURN
2982      END

2983      SUBROUTINE OMP_SET_LOCK(LOCK)
2984      POINTER (LOCK,IL)
2985      INTEGER IL

2986      IF (LOCK .EQ. 0) THEN
2987          PRINT *, 'ERROR: LOCK NOT INITIALIZED'
2988          STOP
2989      ELSEIF (LOCK .EQ. 1) THEN
2990          PRINT *, 'ERROR: DEADLOCK IN USING LOCK VARIABLE'
2991          STOP
2992      ELSE
2993          LOCK = 1
2994      ENDIF
2995      RETURN
2996      END

2997      SUBROUTINE OMP_SET_NEST_LOCK(NLOCK)
2998      POINTER (NLOCK,NIL)
2999      INTEGER NIL

3000      IF (NLOCK .EQ. 0) THEN
3001          PRINT *, 'ERROR: NESTED LOCK NOT INITIALIZED'
3002          STOP
3003      ELSEIF (NLOCK .EQ. 1) THEN
3004          PRINT *, 'ERROR: DEADLOCK USING NESTED LOCK VARIABLE'
3005          STOP
3006      ELSE
3007          NLOCK = 1
```

```
3008      ENDIF

3009      RETURN
3010      END

3011      SUBROUTINE OMP_UNSET_LOCK(LOCK)
3012      POINTER (LOCK,IL)
3013      INTEGER IL
3014      IF (LOCK .EQ. 0) THEN
3015          PRINT *, 'ERROR: LOCK NOT INITIALIZED'
3016          STOP
3017      ELSEIF (LOCK .EQ. 1) THEN
3018          LOCK = -1
3019      ELSE
3020          PRINT *, 'ERROR: LOCK NOT SET'
3021          STOP
3022      ENDIF
3023      RETURN
3024      END

3025      SUBROUTINE OMP_UNSET_NEST_LOCK(NLOCK)
3026      POINTER (NLOCK,NIL)
3027      INTEGER NIL

3028      IF (NLOCK .EQ. 0) THEN
3029          PRINT *, 'ERROR: NESTED LOCK NOT INITIALIZED'
3030          STOP
3031      ELSEIF (NLOCK .EQ. 1) THEN
3032          NLOCK = -1
3033      ELSE
3034          PRINT *, 'ERROR: NESTED LOCK NOT SET'
3035          STOP
3036      ENDIF

3037      RETURN
3038      END

3039      LOGICAL FUNCTION OMP_TEST_LOCK(LOCK)
3040      POINTER (LOCK,IL)
3041      INTEGER IL
3042      IF (LOCK .EQ. -1) THEN
3043          LOCK = 1
3044          OMP_TEST_LOCK = .TRUE.
```

```
3045      ELSEIF (LOCK .EQ. 1) THEN
3046          OMP_TEST_LOCK = .FALSE.
3047      ELSE
3048          PRINT *, 'ERROR: LOCK NOT INITIALIZED'
3049          STOP
3050      ENDIF
3051      RETURN
3052  END

3053      INTEGER FUNCTION OMP_TEST_NEST_LOCK(NLOCK)
3054      POINTER (NLOCK,NIL)
3055      INTEGER NIL

3056      IF (NLOCK .EQ. -1) THEN
3057          NLOCK = 1
3058          OMP_TEST_NEST_LOCK = 1
3059      ELSEIF (NLOCK .EQ. 1) THEN
3060          OMP_TEST_NEST_LOCK = 0
3061      ELSE
3062          PRINT *, 'ERROR: NESTED LOCK NOT INITIALIZED'
3063          STOP
3064      ENDIF

3065      RETURN
3066  END

3067      DOUBLE PRECISION OMP_WTIME()
3068      ! This function does not provide a working
3069      ! wall-clock timer. Replace it with a version
3070      ! customized for the target machine.
3071      OMP_WTIME = 0
3072      RETURN
3073  END

3074      DOUBLE PRECISION OMP_WTICK()
3075      ! This function does not provide a working
3076      ! clock tick function. Replace it with
3077      ! a version customized for the target machine.
3078      DOUBLE PRECISION ONE_YEAR
3079      PARAMETER (ONE_YEAR=365.D0*86400.D0)
3080      OMP_WTICK=ONE_YEAR
3081      RETURN
3082  END
```


Using the SCHEDULE Clause [C]

3084 A parallel region has at least one barrier, at its end, and may have additional barriers
 3085 within it. At each barrier, the other members of the team must wait for the last
 3086 thread to arrive. To minimize this wait time, shared work should be distributed so
 3087 that all threads arrive at the barrier at about the same time. If some of that shared
 3088 work is contained in DO constructs, the SCHEDULE clause can be used for this purpose.

3089 When there are repeated references to the same objects, the choice of schedule for a
 3090 DO construct may be determined primarily by characteristics of the memory system,
 3091 such as the presence and size of caches and whether memory access times are
 3092 uniform or nonuniform. Such considerations may make it preferable to have each
 3093 thread consistently refer to the same set of elements of an array in a series of loops,
 3094 even if some threads are assigned relatively less work in some of the loops. This can
 3095 be done by using the STATIC schedule with the same bounds for all the loops. In the
 3096 following example, note that 1 is used as the lower bound in the second loop, even
 3097 though K would be more natural if the schedule were not important.

```
3098 !$OMP PARALLEL
3099 !$OMP DO SCHEDULE(STATIC)
3100   DO I=1,N
3101     A(I) = WORK1(I)
3102   ENDDO
3103 !$OMP DO SCHEDULE(STATIC)
3104   DO I=1,N
3105     IF(I .GE. K) A(I) = A(I) + WORK2(I)
3106   ENDDO
3107 !$OMP END PARALLEL
3108   ENDDO
```

3109 In the remaining examples, it is assumed that memory access is not the dominant
 3110 consideration, and, unless otherwise stated, that all threads receive comparable
 3111 computational resources. In these cases, the choice of schedule for a DO construct
 3112 depends on all the shared work that is to be performed between the nearest preceding
 3113 barrier and either the implied closing barrier or the nearest subsequent barrier, if
 3114 there is a NOWAIT clause. For each kind of schedule, a short example shows how that
 3115 schedule kind is likely to be the best choice. A brief discussion follows each example.

3116 The STATIC schedule is also appropriate for the simplest case, a parallel region
 3117 containing a single DO construct, with each iteration requiring the same amount of
 3118 work.

```
3119 !$OMP PARALLEL DO SCHEDULE(STATIC)
3120   DO I=1,N
3121     CALL INVARIANT_AMOUNT_OF_WORK(I)
```

3122 ENDDO

3123 The STATIC schedule is characterized by the properties that each thread gets
 3124 approximately the same number of iterations as any other thread, and each thread
 3125 can independently determine the iterations assigned to it. Thus no synchronization is
 3126 required to distribute the work, and, under the assumption that each iteration
 3127 requires the same amount of work, all threads should finish at about the same time.

3128 For a team of P threads, let $\text{CEILING}(N/P)$ be the integer Q , which satisfies $N = P*Q$
 3129 - R with $0 \leq R < P$. One implementation of the STATIC schedule for this example
 3130 would assign Q iterations to the first $P-1$ threads, and $Q-R$ iterations to the last
 3131 thread. Another acceptable implementation would assign Q iterations to the first $P-R$
 3132 threads, and $Q-1$ iterations to the remaining R threads. This illustrates why a
 3133 program should not rely on the details of a particular implementation.

3134 The DYNAMIC schedule is appropriate for the case of a DO construct with the
 3135 iterations requiring varying, or even unpredictable, amounts of work.

```
3136 !$OMP PARALLEL DO SCHEDULE(DYNAMIC)
3137   DO I=1,N
3138     CALL UNPREDICTABLE_AMOUNT_OF_WORK(I)
3139   ENDDO
```

3140 The DYNAMIC schedule is characterized by the property that no thread waits at the
 3141 barrier for longer than it takes another thread to execute its final iteration. This
 3142 requires that iterations be assigned one at a time to threads as they become
 3143 available, with synchronization for each assignment. The synchronization overhead
 3144 can be reduced by specifying a minimum chunk size K greater than 1, so that each
 3145 thread is assigned K iterations at a time until fewer than K iterations remain. This
 3146 guarantees that no thread waits at the barrier longer than it takes another thread to
 3147 execute its final chunk of (at most) K iterations.

3148 The DYNAMIC schedule can be useful if the threads receive varying computational
 3149 resources, which has much the same effect as varying amounts of work for each
 3150 iteration. Similarly, the DYNAMIC schedule can also be useful if the threads arrive at
 3151 the DO construct at varying times, though in some of these cases the GUIDED schedule
 3152 may be preferable.

3153 The GUIDED schedule is appropriate for the case in which the threads may arrive at
 3154 varying times at a DO construct with each iteration requiring about the same amount
 3155 of work. This can happen if, for example, the DO construct is preceded by one or more
 3156 SECTIONS or DO constructs with NOWAIT clauses.

```
3157 !$OMP PARALLEL
3158 !$OMP SECTIONS
3159   .....
3160   !$OMP END SECTIONS NOWAIT
```

```
3161 !$OMP DO SCHEDULE(GUIDED)
3162     DO I=1,N
3163         CALL INVARIANT_AMOUNT_OF_WORK(I)
3164     ENDDO
```

3165 Like DYNAMIC, the GUIDED schedule guarantees that no thread waits at the barrier
3166 longer than it takes another thread to execute its final iteration, or final K iterations
3167 if a chunk size of K is specified. Among such schedules, the GUIDED schedule is
3168 characterized by the property that it requires the fewest synchronizations. For chunk
3169 size K , a typical implementation will assign $Q = \text{CEILING}(N/P)$ iterations to the first
3170 available thread, set N to the larger of $N-Q$ and $P*K$, and repeat until all iterations
3171 are assigned.

3172 When the choice of the optimum schedule is not as clear as it is for these examples,
3173 the RUNTIME schedule is convenient for experimenting with different schedules and
3174 chunk sizes without having to modify and recompile the program. It can also be
3175 useful when the optimum schedule depends (in some predictable way) on the input
3176 data to which the program is applied.

3177 To see an example of the trade-offs between different schedules, consider sharing
3178 1000 iterations among 8 threads. Suppose there is an invariant amount of work in
3179 each iteration, and use that as the unit of time.

3180 If all threads start at the same time, the STATIC schedule will cause the construct to
3181 execute in 125 units, with no synchronization. But suppose that one thread is 100
3182 units late in arriving. Then the remaining seven threads wait for 100 units at the
3183 barrier, and the execution time for the whole construct increases to 225.

3184 Because both the DYNAMIC and GUIDED schedules ensure that no thread waits for
3185 more than one unit at the barrier, the delayed thread causes their execution times for
3186 the construct to increase only to 138 units, possibly increased by delays from
3187 synchronization. If such delays are not negligible, it becomes important that the
3188 number of synchronizations is 1000 for DYNAMIC but only 41 for GUIDED, assuming
3189 the default chunk size of one. With a chunk size of 25, DYNAMIC and GUIDED both
3190 finish in 150 units, plus any delays from the required synchronizations, which now
3191 number only 40 and 20, respectively.

3193 This appendix gives examples of the Fortran `INCLUDE` file and Fortran 90 module
 3194 that shall be provided by implementations as specified in Chapter 3, page 47.

3195 It has three sections:

- 3196 • Section D.1, page 105, contains an example of a FORTRAN 77 interface
 3197 declaration `INCLUDE` file.
- 3198 • Section D.2, page 107, contains an example of a Fortran 90 interface declaration
 3199 `MODULE`.
- 3200 • Section D.3, page 111, contains an example of a Fortran 90 generic interface for a
 3201 library routine.

3202

D.1 Example of an Interface Declaration `INCLUDE` File

```
3203       C       the "C" of this comment starts in column 1
 3204       integer    omp_lock_kind
 3205       parameter ( omp_lock_kind = 8 )

 3206       integer    omp_nest_lock_kind
 3207       parameter ( omp_nest_lock_kind = 8 )

 3208       C                    default integer type assumed below
 3209       C                    default logical type assumed below
 3210       C                    OpenMP Fortran API v1.1
 3211       integer    openmp_version
 3212       parameter ( openmp_version = 200011 )

 3213       external   omp_destroy_lock

 3214       external   omp_destroy_nest_lock

 3215       external   omp_get_dynamic
 3216       logical    omp_get_dynamic

 3217       external   omp_get_max_threads
 3218       integer    omp_get_max_threads

 3219       external   omp_get_nested
```

```
3220      logical  omp_get_nested
3221
3222      external omp_get_num_procs
3223      integer   omp_get_num_procs
3224
3225      external omp_get_num_threads
3226      integer   omp_get_num_threads
3227
3228      external omp_get_thread_num
3229      integer   omp_get_thread_num
3230
3231      external omp_get_wtick
3232      double precision  omp_get_wtick
3233
3234      external omp_get_wtime
3235      double precision  omp_get_wtime
3236
3237      external omp_init_lock
3238
3239      external omp_init_nest_lock
3240
3241      external omp_in_parallel
3242      logical   omp_in_parallel
3243
3244      external omp_set_dynamic
3245
3246      external omp_set_lock
3247
3248      external omp_set_nest_lock
3249
3250      external omp_set_nested
3251
3252      external omp_set_num_threads
3253
3254      external omp_test_lock
3255      logical   omp_test_lock
3256
3257      external omp_test_nest_lock
3258      integer   omp_test_nest_lock
3259
3260      external omp_unset_lock
3261
3262      external omp_unset_nest_lock
```

3246

D.2 Example of a Fortran 90 Interface Declaration MODULE

```
3247      !      the "!" of this comment starts in column 1
3248
3249      module omp_lib_kinds
3250
3251      integer, parameter :: omp_integer_kind      = 4
3252      integer, parameter :: omp_logical_kind     = 4
3253      integer, parameter :: omp_lock_kind        = 8
3254      integer, parameter :: omp_nest_lock_kind   = 8
3255
3256      end module omp_lib_kinds
3257
3258      module omp_lib
3259
3260      use omp_lib_kinds
3261
3262      !                         OpenMP Fortran API v1.1
3263      integer, parameter :: openmp_version = 199910
3264
3265      interface
3266          subroutine omp_destroy_lock ( var )
3267          use omp_lib_kinds
3268          integer ( kind=omp_lock_kind ), intent(inout) :: var
3269          end subroutine omp_destroy_lock
3270      end interface
3271
3272      interface
3273          subroutine omp_destroy_nest_lock ( var )
3274          use omp_lib_kinds
3275          integer ( kind=omp_nest_lock_kind ), intent(inout) :: var
3276          end subroutine omp_destroy_nest_lock
3277      end interface
3278
3279      interface
3280          function omp_get_dynamic ( )
3281          use omp_lib_kinds
3282          logical ( kind=omp_logical_kind ) :: omp_get_dynamic
3283          end function omp_get_dynamic
3284      end interface
3285
3286      interface
3287          function omp_get_max_threads ( )
3288          use omp_lib_kinds
3289      end interface
```

```
3279      integer ( kind=omp_integer_kind ) :: omp_get_max_threads
3280      end function omp_get_max_threads
3281
3282  interface
3283    function omp_get_nested ()
3284    use omp_lib_kinds
3285    logical ( kind=omp_logical_kind ) :: omp_get_nested
3286    end function omp_get_nested
3287  end interface
3288
3289  interface
3290    function omp_get_num_procs ()
3291    use omp_lib_kinds
3292    integer ( kind=omp_integer_kind ) :: omp_get_num_procs
3293    end function omp_get_num_procs
3294  end interface
3295
3296  interface
3297    function omp_get_num_threads ()
3298    use omp_lib_kinds
3299    integer ( kind=omp_integer_kind ) :: omp_get_num_threads
3300    end function omp_get_num_threads
3301
3302  interface
3303    function omp_get_thread_num ()
3304    use omp_lib_kinds
3305    integer ( kind=omp_integer_kind ) :: omp_get_thread_num
3306    end function omp_get_thread_num
3307
3308  interface
3309    function omp_get_wtick ()
3310    double precision :: omp_get_wtick
3311    end function omp_get_wtick
3312
3313  interface
3314    function omp_get_wtime ()
3315    double precision :: omp_get_wtime
3316    end function omp_get_wtime
3317
3318  interface
```

```
3317      subroutine omp_init_lock ( var )
3318      use omp_lib_kinds
3319      integer ( kind=omp_lock_kind ), intent(out) :: var
3320      end subroutine omp_init_lock
3321  end interface

3322  interface
3323      subroutine omp_init_nest_lock ( var )
3324      use omp_lib_kinds
3325      integer ( kind=omp_nest_lock_kind ), intent(out) :: var
3326      end subroutine omp_init_nest_lock
3327  end interface

3328  interface
3329      function omp_in_parallel ()
3330      use omp_lib_kinds
3331      logical ( kind=omp_logical_kind ) :: omp_in_parallel
3332      end function omp_in_parallel
3333  end interface

3334  interface
3335      subroutine omp_set_dynamic ( enable_expr )
3336      use omp_lib_kinds
3337      logical ( kind=omp_logical_kind ), intent(in) :: enable_expr
3338      end subroutine omp_set_dynamic
3339  end interface

3340  interface
3341      subroutine omp_set_lock ( var )
3342      use omp_lib_kinds
3343      integer ( kind=omp_lock_kind ), intent(inout) :: var
3344      end subroutine omp_set_lock
3345  end interface

3346  interface
3347      subroutine omp_set_nest_lock ( var )
3348      use omp_lib_kinds
3349      integer ( kind=omp_nest_lock_kind ), intent(inout) :: var
3350      end subroutine omp_set_nest_lock
3351  end interface

3352  interface
3353      subroutine omp_set_nested ( enable_expr )
3354      use omp_lib_kinds
3355      logical ( kind=omp_logical_kind ), intent(in) :: &
```

```
3355      &                                         enable_expr
3357          end subroutine omp_set_nested
3358      end interface

3359      interface
3360          subroutine omp_set_num_threads ( number_of_threads_expr )
3361          use omp_lib_kinds
3362          integer ( kind=omp_integer_kind ), intent(in) :: &
3363                                         number_of_threads_expr
3364          &
3365          end subroutine omp_set_num_threads
3366      end interface

3367      interface
3368          function omp_test_lock ( var )
3369          use omp_lib_kinds
3370          logical ( kind=omp_logical_kind ) :: omp_test_lock
3371          integer ( kind=omp_lock_kind ), intent(inout) :: var
3372          end function omp_test_lock
3373      end interface

3374      interface
3375          function omp_test_nest_lock ( var )
3376          use omp_lib_kinds
3377          integer ( kind=omp_integer_kind ) :: omp_test_nest_lock
3378          integer ( kind=omp_nest_lock_kind ), intent(inout) :: var
3379          end function omp_test_nest_lock
3380      end interface

3381      interface
3382          subroutine omp_unset_lock ( var )
3383          use omp_lib_kinds
3384          integer ( kind=omp_lock_kind ), intent(inout) :: var
3385          end subroutine omp_unset_lock
3386      end interface

3387      interface
3388          subroutine omp_unset_nest_lock ( var )
3389          use omp_lib_kinds
3390          integer ( kind=omp_nest_lock_kind ), intent(inout) :: var
3391          end subroutine omp_unset_nest_lock
3392      end interface
3393  end module omp_lib
```

3393 D.3 Example of a Generic Interface for a Library Routine

3394 Any of the OMP runtime library routines that take an argument may be extended
3395 with a generic interface so arguments of different KIND type can be accommodated.

3396 Assume an implementation supports both default INTEGER as KIND =
3397 OMP_INTEGER_KIND and another INTEGER KIND, KIND = SHORT_INT. Then
3398 OMP_SET_NUM_THREADS could be specified in the `omp_lib` module as the following:

```
3399 !       the "!" of this comment starts in column 1
3400 interface omp_set_num_threads
3401     subroutine omp_set_num_threads_1 ( number_of_threads_expr )
3402         use omp_lib_kinds
3403         integer ( kind=omp_integer_kind ), intent(in) :: &
3404             &                                         number_of_threads_expr
3405     end subroutine omp_set_num_threads_1
3406     subroutine omp_set_num_threads_2 ( number_of_threads_expr )
3407         use omp_lib_kinds
3408         integer ( kind=short_int ), intent(in) :: &
3409             &                                         number_of_threads_expr
3410     end subroutine omp_set_num_threads_2
3411 end interface omp_set_num_threads
```


Implementation-Dependent Behaviors in OpenMP Fortran [E]

This appendix summarizes the behaviors that are described as “implementation dependent” 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.

- `SCHEDULE(GUIDED, chunk)`: *chunk* specifies the size of the smallest piece, except possibly the last. The size of the initial piece is implementation dependent (Table 1, page 17).
- When `SCHEDULE(RUNTIME)` is specified, the decision regarding scheduling is deferred until run time. The schedule type and chunk size can be chosen at run time by setting the `OMP_SCHEDULE` environment variable. If this environment variable is not set, the resulting schedule is implementation-dependent (Table 1, page 17).
- In the absence of the `SCHEDULE` clause, the default schedule is implementation-dependent (Section 2.3.1, page 15).
- `OMP_GET_NUM_THREADS`: If the number of threads has not been explicitly set by the user, the default is implementation-dependent (Section 3.1.2, page 48).
- `OMP_SET_DYNAMIC`: The default for dynamic thread adjustment is implementation-dependent (Section 3.1.7, page 51).
- `OMP_SET_NESTED`: When nested parallelism is enabled, the number of threads used to execute nested parallel regions is implementation-dependent (Section 3.1.9, page 52).
- `OMP_SCHEDULE` environment variable: The default value for this environment variable is implementation-dependent (Section 4.1, page 59).
- `OMP_NUM_THREADS` environment variable: The default value is implementation-dependent (Section 4.2, page 60).
- `OMP_DYNAMIC` environment variable: The default value is implementation-dependent (Section 4.3, page 60).
- An implementation can replace all `ATOMIC` directives by enclosing the statement in a critical section (Section 2.5.4, page 27).
- If the dynamic threads mechanism is enabled on entering a parallel region, the allocation status of an allocatable array that is not affected by a `COPYIN` clause that appears on the region is implementation-dependent (Section 2.6.1, page 32).

- 3445
- 3446 Due to resource constraints, it is not possible for an implementation to document
3447 the maximum number of threads that can be created successfully during a
3448 program's execution. This number is dependent upon the load on the system, the
3449 amount of memory allocated by the program, and the amount of implementation
3450 dependent stack space allocated to each thread. If the dynamic threads
3451 mechanism is disabled, the behavior of the program is implementation-dependent
3452 when more threads are requested than can be successfully created. If the dynamic
3453 threads mechanism is enabled, requests for more threads than an implementation
3454 can support are satisfied by a smaller number of threads (Section 2.3.1, page 15).
 - 3455
 - 3456 If an OMP runtime library routine interface is defined to be generic by an
3457 implementation, use of arguments of kind other than those specified by the
OMP_*_KIND constants is implementation-dependent (Section D.3, page 111).
- 3458

New Features in OpenMP Fortran version 2.0 [F]

This appendix summarizes the key changes made to the OpenMP Fortran specification in moving from version 1.1 to version 2.0. The following items are new features added to the specification:

- The FORTRAN 77 standard does not require that initialized data have the `SAVE` attribute but Fortran 95 does require this. OpenMP Fortran version 2.0 requires this. See Section 1.4, page 4.
- An OpenMP compliant implementation must document its implementation-defined behaviors. See Appendix E, page 113.
- Directives may contain end-of-line comments starting with an exclamation point. See Section 2.1.2, page 10.
- The `_OPENMP` preprocessor macro is defined to be an integer of the form `YYYYMM` where `YYYY` and `MM` are the year and month of the version of the OpenMP Fortran specification supported by the implementation. See Section 2.1.3, page 10.
- `COPYPRIVATE` is a new modifier on `END SINGLE`. See Section 2.6.2.8, page 41.
- `THREADPRIVATE` may now be applied to variables as well as `COMMON` blocks. See Section 2.6.1, page 32.
- `REDUCTION` is now allowed on an array name. See Section 2.6.2.6, page 38.
- `COPYIN` now works on variables as well as `COMMON` blocks. See Section 2.6.2.7, page 41.
- Reprivation of variables is now allowed. See Section 2.6.3, page 42.
- Nested lock routines consistent with those defined in the C/C++ specification have been added. See Section 3.2, page 52.
- Wallclock timers have been added. See Section 3.3, page 56.
- An example of `INTERFACE` definitions for all of the OpenMP runtime routines has been added to the specification. See Appendix D, page 105.
- The `NUM_THREADS` clause on parallel regions defines the number of threads to be used to execute that region. See Section 2.2, page 12.
- The `WORKSHARE` directive allows parallelization of array expressions in Fortran statements. See Section 2.3.4, page 20.

The following items list changes that served to clarify features or to correct errors within the OpenMP Fortran specification:

- 3491 • Under the right circumstances, subsequent parallel regions use the same threads
3492 with the same thread numbers as previous regions. See Section 2.2, page 12.
- 3493 • It is implementation-defined whether global variable references in statement
3494 functions refer to SHARED or PRIVATE copies of those variables. See Section 2.6.2,
3495 page 34
- 3496 • Exceptional values (such as negative infinity) may affect the behavior of a
3497 program. This can occur with REDUCTION, FIRSTPRIVATE, LASTPRIVATE,
3498 COPYPRIVATE, or COPYIN. See Section 2.6.3, page 42.
- 3499 • Additional examples have been added. See Appendix A, page 63.