

1           OpenMP Fortran Application Program  
2           Interface  
3           Version2.0 Draft 9



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

	<i>Page</i>
36	
37	23
38	25
39	26
40	27
41	27
42	30
43	31
44	32
45	32
46	33
47	33
48	34
49	36
50	37
51	38
52	40
53	41
54	<b>43</b>
55	43
56	44
57	44
58	45
59	45
60	46
61	46
62	46
63	47
64	47
65	48
66	48
67	50
68	50
69	50
70	51

	<i>Page</i>
71	
72	OMP_TEST_LOCK and OMP_TEST_NEST_LOCK Functions . . . . . 51
73	<b>Timing Routines</b> . . . . . 52
74	OMP_GET_WTIME Function . . . . . 52
75	OMP_GET_WTICK Function . . . . . 53
76	<b>Environment Variables [4]</b> <b>55</b>
77	OMP_SCHEDULE Environment Variable . . . . . 55
78	OMP_NUM_THREADS Environment Variable . . . . . 55
79	OMP_DYNAMIC Environment Variable . . . . . 56
80	OMP_NESTED Environment Variable . . . . . 56
81	<b>Appendix A Examples</b> <b>57</b>
82	Executing a Simple Loop in Parallel . . . . . 57
83	Specifying Conditional Compilation . . . . . 57
84	Using Parallel Regions . . . . . 58
85	Using the NOWAIT Clause . . . . . 58
86	Using the CRITICAL Directive . . . . . 58
87	Using the LASTPRIVATE Clause . . . . . 59
88	Using the REDUCTION Clause . . . . . 59
89	Specifying Parallel Sections . . . . . 61
90	Using SINGLE Directives . . . . . 61
91	Specifying Sequential Ordering . . . . . 62
92	Specifying a Fixed Number of Threads . . . . . 62
93	Using the ATOMIC Directive . . . . . 63
94	Using the FLUSH Directive . . . . . 63
95	Determining the Number of Threads Used . . . . . 64
96	Using Locks . . . . . 64
97	Using Nestable Locks . . . . . 65
98	Nested DO Directives . . . . . 67
99	Examples Showing Incorrect Nesting of Work-sharing Directives . . . . . 68
100	Binding of BARRIER Directives . . . . . 70
101	Scoping Variables with the PRIVATE Clause . . . . . 71
102	Examples of Invalid Storage Association . . . . . 71
103	Examples of Syntax of Parallel DO Loops . . . . . 74

	<i>Page</i>
104	
105 Examples of the <code>ATOMIC</code> Directive . . . . .	75
106 Examples of the <code>ORDERED</code> Directive . . . . .	76
107 Examples of <code>THREADPRIVATE</code> Data . . . . .	77
108 Examples of the Data Attribute Clauses: <code>SHARED</code> and <code>PRIVATE</code> . . . . .	80
109 Examples of the Data Attribute Clause: <code>COPYPRIVATE</code> . . . . .	82
110 Examples of <code>WORKSHARE</code> Directive . . . . .	84
<b>111 Appendix B Stubs for Run-time Library Routines</b>	<b>87</b>
<b>112 Appendix C Using the <code>SCHEDULE</code> Clause</b>	<b>93</b>
<b>113 Appendix D Interface Declaration Module</b>	<b>97</b>
114 Example of an Interface Declaration <code>INCLUDE</code> File . . . . .	97
115 Example of a Fortran 90 Interface Declaration <code>MODULE</code> . . . . .	99
116 Example of a Generic Interface for a Library Routine . . . . .	103
<b>117 Appendix E Implementation Dependent Behaviors in OpenMP Fortran</b>	<b>105</b>
<b>118 Appendix F New Features in OpenMP Fortran version 2.0</b>	<b>107</b>
<b>119 Appendix G Glossary</b>	<b>109</b>
<b>120 Tables</b>	
121 Table 1. <code>SCHEDULE</code> Clause Values . . . . .	14
122 Table 2. Reduction Variable Initialization Values . . . . .	35

123  
124  
125  
126

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





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 execute  
154 correctly.

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

## 156 1.2 Execution Model

157 The OpenMP Fortran API uses the fork-join model of parallel execution. A program  
158 that is written with the OpenMP Fortran API begins execution as a single process,  
159 called the *master thread* of execution. The master thread executes sequentially until  
160 the first parallel construct is encountered. In the OpenMP Fortran API, the `PARALLEL`  
161 and `END PARALLEL` directive pair constitutes the parallel construct. When a parallel  
162 construct is encountered, the master thread creates a *team* of threads, and the master  
163 thread becomes the master of the team. The statements in the program that are  
164 enclosed by the parallel construct, including routines called from within the enclosed  
165 statements, are executed in parallel by each thread in the team. The statements  
166 enclosed lexically within a construct define the *lexical* extent of the construct. The  
167 *dynamic* extent further includes the routines called from within the construct.

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

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

## 179 1.3 Compliance

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

184 The OpenMP Fortran API is an extension to the base language that is supported by  
185 an implementation. If the base language does not support a language construct or  
186 extension that appears in this document, the OpenMP implementation is not required  
187 to support it.

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

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

198           In both Fortran 90 and Fortran 95, a variable that has explicit initialization  
199           implicitly has the *SAVE* attribute. This is not the case in FORTRAN 77. However, an  
200           implementation of OpenMP Fortran must give such a variable the *SAVE* attribute,  
201           regardless of the version of Fortran upon which it is based.

202           The OpenMP Fortran API specifies that certain behavior is “implementation  
203           dependent”. A conforming OpenMP implementation is required to define and  
204           document its behavior in these cases. See Appendix E, page 105, for a list of  
205           implementation dependent behaviors.

## 206    **1.4 Organization**

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

- 208           • Chapter 2, page 5, describes the compiler directives.
- 209           • Chapter 3, page 43, describes the run-time library routines.
- 210           • Chapter 4, page 55, describes the environment variables.
- 211           • Appendix A, page 57, contains examples.
- 212           • Appendix B, page 87, describes stub library routines.
- 213           • Appendix C, page 93, has information about using the *SCHEDULE* clause.
- 214           • Appendix D, page 97, has examples of interfaces for the run-time library routines.
- 215           • Appendix E, page 105, describes implementation-dependent behaviors.



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

223 This chapter addresses the following topics:

- 224 • Section 2.1, page 5, describes the directive format.
- 225 • Section 2.2, page 9, describes the parallel region construct.
- 226 • Section 2.3, page 12, describes work-sharing constructs.
- 227 • Section 2.4, page 19, describes the combined parallel work-sharing constructs.
- 228 • Section 2.5, page 21, describes synchronization constructs and the MASTER  
 229 directive.
- 230 • Section 2.6, page 27, describes the data environment, which includes directives  
 231 and clauses that affect the data environment.
- 232 • Section 2.7, page 40, describes directive binding.
- 233 • Section 2.8, page 41, describes directive nesting.

## 234 2.1 OpenMP Directive Format

235 The format of an OpenMP directive is as follows:

236 

*sentinel directive\_name* [*clause*[[*,*] *clause*]. . .]

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

243 The following sections describe the OpenMP directive format:

- 244 • Section 2.1.1, page 6, describes directive sentinels.
- 245 • Section 2.1.2, page 8, describes comments inside directives.
- 246 • Section 2.1.3, page 8, describes conditional compilation.

### 247 **2.1.1 Directive Sentinels**

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

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

#### 253 *2.1.1.1 Fixed Source Form Directive Sentinels*

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

!\$OMP   C\$OMP   *\$OMP
--------------------------

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

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

```

263 C23456789
264 !$OMP PARALLEL DO SHARED(A,B,C)

265 C$OMP PARALLEL DO
266 C$OMP+SHARED(A,B,C)

267 C$OMP PARALLELDOSHARED(A,B,C)
  
```

#### 268 *2.1.1.2 Free Source Form Directive Sentinel*

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

270

!\$OMP

271

272

273

274

275

276

277

278

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

279

280

281

One or more blanks must be used to separate adjacent keywords in directives in free source form, except in the following cases, where blanks are optional between the given pair of keywords:

282

END CRITICAL

283

END DO

284

END MASTER

285

END ORDERED

286

END PARALLEL

287

END SECTIONS

288

END SINGLE

289

END WORKSHARE

290

PARALLEL DO

291

PARALLEL SECTIONS

292

PARALLEL BLOCK

293

294

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

295

!23456789

296

!\$OMP PARALLEL DO &amp;

297

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

298

!\$OMP PARALLEL &amp;

299

!\$OMP&amp;DO SHARED(A,B,C)

300

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

301

302

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

## 303 **2.1.2 Comments Inside Directives**

304           The OpenMP Fortran API accepts comments placed inside directives. The rules  
305           governing such comments depend on the Fortran source form being used.

### 306 *2.1.2.1 Comments in Directives with Fixed Source Form*

307           Comments may appear on the same line as a directive. The exclamation point  
308           initiates a comment when it appears after column 6. The comment extends to the end  
309           of the source line. If the first nonblank character after the directive sentinel of an  
310           initial or continuation directive line is an exclamation point, the line is ignored.

### 311 *2.1.2.2 Comments in Directives with Free Source Form*

312           Comments may appear on the same line as a directive. The exclamation point  
313           initiates a comment. The comment extends to the end of the source line. If the first  
314           nonblank character after the directive sentinel is an exclamation point, the line is  
315           ignored.

## 316 **2.1.3 Conditional Compilation**

317           The OpenMP Fortran API permits Fortran lines to be compiled conditionally. The  
318           directive sentinels for conditional compilation that are accepted by an  
319           OpenMP-compliant compiler depend on the Fortran source form being used. The !\$  
320           sentinel is accepted when compiling either fixed source form files or free source form  
321           files. The C\$ and \*\$ sentinels are accepted only when compiling fixed source form.

322           During OpenMP compilation, the sentinel is replaced by two spaces, and the rest of  
323           the line is treated as a normal Fortran line.

324           In addition to the Fortran conditional compilation sentinels, a C preprocessor macro,  
325           \_OPENMP, can be used for conditional compilation. OpenMP-compliant compilers  
326           define this macro during OpenMP compilation to have the decimal value YYYYMM  
327           where YYYY and MM are the year and month designations of the version of the  
328           OpenMP Fortran API that the implementation supports.

329           The following sections contain more information on using the different sentinels for  
330           conditional compilation. (See Section A.2, page 57, for an example.)

### 331 *2.1.3.1 Fixed Source Form Conditional Compilation Sentinels*

332           The OpenMP Fortran API accepts the following conditional compilation sentinels in  
333           fixed source form files:



334

!\$ | C\$ | \*\$

335

336

337

338

339

340

341

342

The sentinels must start in column 1 and appear as a single word with no intervening white space. Fortran fixed form line length, case sensitivity, white space, continuation, and column rules apply to the line. After the sentinel is replaced with two spaces, initial lines must have a space or zero in column 6 and only white space and numbers in columns 1 through 5. After the sentinel is replaced with two spaces, continuation lines must have a character other than a space or zero in column 6 and only white space in columns 1 through 5. If these criteria are not met, the line is treated as a comment and ignored.

343

Example: The following forms for specifying conditional compilation are equivalent:

344

C23456789

345

!\$ 10 IAM = OMP\_GET\_THREAD\_NUM +

346

!\$ &amp; INDEX

347

#ifdef \_OPENMP

348

10 IAM = OMP\_GET\_THREAD\_NUM +

349

&amp; INDEX

350

#endif

351

### 2.1.3.2 Free Source Form Conditional Compilation Sentinel

352

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

353

354

!\$

355

356

357

358

359

360

361

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

362

## 2.2 Parallel Region Construct

363

The `PARALLEL` and `END PARALLEL` directives define a *parallel region*. A parallel region is a block of code that is to be executed by multiple threads in parallel. This is

364

365 the fundamental parallel construct in OpenMP that starts parallel execution. These  
366 directives have the following format:

```
367 !$OMP PARALLEL [clause[[,] clause]. . . ]
368 block
369 !$OMP END PARALLEL
```

370 *clause* can be one of the following:

- 371 • PRIVATE(*list*)
- 372 • SHARED(*list*)
- 373 • DEFAULT(PRIVATE | SHARED | NONE)
- 374 • FIRSTPRIVATE(*list*)
- 375 • REDUCTION( {*operator* | *intrinsic\_procedure\_name* } : *list*)
- 376 • IF(*scalar\_logical\_expression*)
- 377 • COPYIN(*list*)
- 378 • NUM\_THREADS(*scalar\_integer\_expression*)

379 For information about the PRIVATE, SHARED, DEFAULT, FIRSTPRIVATE, REDUCTION,  
380 and COPYIN clauses, see Section 2.6.2, page 30. For an example of how to implement  
381 coarse-grain parallelism using these directives, see Section A.3, page 58.

382 When a thread encounters a parallel region, it creates a team of threads, and it  
383 becomes the master of the team. The master thread is a member of the team. The  
384 number of threads in the team is controlled by environment variables, the  
385 NUM\_THREADS clause, and/or library calls. For more information on environment  
386 variables, see Chapter 4, page 55. For more information on library routines, see  
387 Chapter 3, page 43.

388 The number of physical processors actually hosting the threads at any given time is  
389 implementation dependent. Once created, the number of threads in the team remains  
390 constant for the duration of that parallel region. It can be changed either explicitly by  
391 the user or automatically by the run-time system from one parallel region to another.  
392 The OMP\_SET\_DYNAMIC library routine and the OMP\_DYNAMIC environment variable  
393 can be used to enable and disable the automatic adjustment of the number of threads.  
394 For more information on the OMP\_SET\_DYNAMIC library routine, see Section 3.1.7,  
395 page 46. For more information on the OMP\_DYNAMIC environment variable, see  
396 Section 4.3, page 56.

397 Within the dynamic scope of a parallel region, thread numbers uniquely identify each  
398 thread. Thread numbers are consecutive whole numbers ranging from zero for the  
399 master thread up to one less than the number of threads within the team. The value  
400 of the thread number is found by a call to the `OMP_GET_THREAD_NUM` library routine  
401 (for more information see Section 3.1.4, page 45). If dynamic threads are disabled  
402 when the parallel region is encountered, and remain disabled until a subsequent,  
403 non-nested parallel region is encountered, then the thread numbers for the two  
404 regions are consistent in that the thread identified with a given thread number in the  
405 earlier parallel region will be identified with the same thread number in the later  
406 region.

407 *block* denotes a structured block of Fortran statements. It is non-compliant to branch  
408 into or out of the block. The code contained within the dynamic extent of the parallel  
409 region is executed by each thread. The code path can be different for different threads.

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

413 If a thread in a team executing a parallel region encounters another parallel region, it  
414 creates a new team, and it becomes the master of that new team. This second parallel  
415 region is called a nested parallel region. By default, nested parallel regions are  
416 serialized; that is, they are executed by a team composed of one thread. This default  
417 behavior can be changed by using either the `OMP_SET_NESTED` run-time library  
418 routine or the `OMP_NESTED` environment variable. For more information on the  
419 `OMP_SET_NESTED` library routine, see Section 3.1.9, page 47. For more information on  
420 the `OMP_NESTED` environment variable, see Section 4.4, page 56.

421 If an `IF` clause is present, the enclosed code region is executed in parallel only if the  
422 *scalar\_logical\_expression* evaluates to `.TRUE..` Otherwise, the parallel region is  
423 serialized. The expression must be a scalar Fortran logical expression. In the absence  
424 of an `IF` clause, the region is executed as if an `IF(.TRUE.)` clause were specified.

425 The `NUM_THREADS` clause is used to request that a specific number of threads are  
426 used in a parallel region. It supersedes the number of threads indicated by the  
427 `OMP_SET_NUM_THREADS` function or the `OMP_NUM_THREADS` environment variable for  
428 the parallel region it is applied to. Subsequent parallel regions, however, are not  
429 affected unless they have their own `NUM_THREADS` clauses. *scalar\_integer\_expression*  
430 must evaluate to a positive scalar integer value.

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

437 The following restrictions apply to parallel regions:

- 438 • The `PARALLEL/END PARALLEL` directive pair must appear in the same routine in  
439 the executable section of the code.
  - 440 • The code contained by these two directives must be a structured block. It is  
441 non-compliant to branch into or out of a parallel region.
  - 442 • Only a single `IF` clause can appear on the directive. The `IF` expression is  
443 evaluated outside the context of the parallel region. Results are unspecified if the  
444 `IF` expression contains a function reference that has side effects.
  - 445 • Only a single `NUM_THREADS` clause can appear on the directive. The `NUM_THREADS`  
446 expression is evaluated outside the context of the parallel region. Results are  
447 unspecified if the `NUM_THREADS` expression contains a function reference that has  
448 side effects.
  - 449 • If the dynamic threads mechanism is enabled, then the number of threads  
450 requested by the `NUM_THREADS` clause is the maximum number to use in the  
451 parallel region.
  - 452 • The order of evaluation of `IF` clauses and `NUM_THREADS` clauses is unspecified.
- 453 Unsynchronized use of Fortran I/O statements by multiple threads on the same unit  
454 has unspecified behavior.

## 455 2.3 Work-sharing Constructs

456 A work-sharing construct divides the execution of the enclosed code region among the  
457 members of the team that encounter it. A work-sharing construct must be enclosed  
458 dynamically within a parallel region in order for the directive to execute in parallel.  
459 The work-sharing directives do not launch new threads, and there is no implied  
460 barrier on entry to a work-sharing construct.

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

- 462 • Work-sharing constructs and `BARRIER` directives must be encountered by all  
463 threads in a team or by none at all.
- 464 • Work-sharing constructs and `BARRIER` directives must be encountered in the same  
465 order by all threads in a team.

466 The following sections describe the work-sharing directives:

- 467 • Section 2.3.1, page 13, describes the `DO` and `END DO` directives.
- 468 • Section 2.3.2, page 15, describes the `SECTIONS`, `SECTION`, and `END SECTIONS`  
469 directives.

- 470 • Section 2.3.3, page 17, describes the SINGLE and END SINGLE directives.
- 471 • Section 2.3.4, page 17, describes the WORKSHARE directive.

### 472 2.3.1 DO Directive

473 The DO directive specifies that the iterations of the immediately following DO loop  
 474 must be executed in parallel. The loop that follows a DO directive cannot be a  
 475 DO WHILE or a DO loop without loop control. The iterations of the DO loop are  
 476 distributed across threads that already exist.

477 The format of this directive is as follows:

```

478 !$OMP DO [clause [, ] clause . . . ]
479
480 do_loop
481
482 [!$OMP END DO [NOWAIT]]
```

481 The *do\_loop* may be a *do\_construct*, an *outer\_shared\_do\_construct*, or an  
 482 *inner\_shared\_do\_construct*. A DO construct that contains several DO statements that  
 483 share the same DO termination statement syntactically consists of a sequence of  
 484 *outer\_shared\_do\_constructs*, followed by a single *inner\_shared\_do\_construct*. If an END  
 485 DO directive follows such a DO construct, a DO directive can only be specified for the  
 486 first (i.e., the outermost) *outer\_shared\_do\_construct*. (See examples in Section A.22,  
 487 page 74.)

488 The *clause* can be one of the following:

- 489 • PRIVATE ( *list* )
- 490 • FIRSTPRIVATE ( *list* )
- 491 • LASTPRIVATE ( *list* )
- 492 • REDUCTION ( { *operator* | *intrinsic\_procedure\_name* } : *list* )
- 493 • SCHEDULE ( *type* [ , *chunk* ] )
- 494 • ORDERED

495 The SCHEDULE and ORDERED clauses are described in this section. The PRIVATE,  
 496 FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses are described in Section  
 497 2.6.2, page 30.

498 If ordered sections are contained in the dynamic extent of the DO directive, the  
 499 ORDERED clause must be present. For more information on ordered sections, see the  
 500 ORDERED directive in Section 2.5.6, page 26.

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

507 Table 1. SCHEDULE Clause Values

508	<u><i>type</i></u>	<u>Effect</u>
509	STATIC	When SCHEDULE(STATIC, <i>chunk</i> ) is specified, iterations are divided
510		into pieces of a size specified by <i>chunk</i> . The pieces are statically
511		assigned to threads in the team in a round-robin fashion in the order
512		of the thread number.
513		When <i>chunk</i> is not specified, the iteration space is divided into
514		contiguous chunks that are approximately equal in size with one
515		chunk assigned to each thread.
516	DYNAMIC	When SCHEDULE(DYNAMIC, <i>chunk</i> ) is specified, the iterations are
517		broken into pieces of a size specified by <i>chunk</i> . As each thread
518		finishes a piece of the iteration space, it dynamically obtains the next
519		set of iterations.
520		When no <i>chunk</i> is specified, it defaults to 1.
521	GUIDED	When SCHEDULE(GUIDED, <i>chunk</i> ) is specified, the iteration space is
522		divided into pieces such that the size of each successive piece is
523		exponentially decreasing. <i>chunk</i> specifies the size of the smallest
524		piece, except possibly the last. The size of the initial piece is
525		implementation dependent. As each thread finishes a piece of the
526		iteration space, it dynamically obtains the next available piece.
527		When no <i>chunk</i> is specified, it defaults to 1.
528	RUNTIME	When SCHEDULE(RUNTIME) is specified, the decision regarding
529		scheduling is deferred until run time. The schedule type and chunk
530		size can be chosen at run time by setting the OMP_SCHEDULE
531		environment variable. If this environment variable is not set, the
532		resulting schedule is implementation dependent. For more
533		information on the OMP_SCHEDULE environment variable, see Section
534		4.1, page 55.

535                   When `SCHEDULE(RUNTIME)` is specified, it is non-compliant to specify  
536                   *chunk*.

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

542                   Threads that complete execution of their assigned loop iterations wait at a barrier at  
543                   the `END DO` directive unless the `NOWAIT` clause is specified. If an `END DO` directive is  
544                   not specified, an `END DO` directive is assumed at the end of the `DO` loop. If `NOWAIT` is  
545                   specified on the `END DO` directive, threads do not synchronize at the end of the  
546                   parallel loop: threads that finish early proceed straight to the instructions following  
547                   the loop without waiting for the other members of the team to finish the `DO` directive.  
548                   (See Section A.4, page 58, for an example.)

549                   Parallel `DO` loop control variables are block-level entities within the `DO` loop. If the  
550                   loop control variable also appears in the `LASTPRIVATE` list of the parallel `DO`, it is  
551                   copied out to a variable of the same name in the enclosing `PARALLEL` region. The  
552                   variable in the enclosing `PARALLEL` region must be `SHARED` if it is specified on the  
553                   `LASTPRIVATE` list of a `DO` directive.

554                   The following restrictions apply to the `DO` directives:

- 555                   • It is illegal to branch out of a `DO` loop associated with a `DO` directive.
- 556                   • The values of the loop control parameters of the `DO` loop associated with a `DO`  
557                    directive must be the same for all the threads in the team.
- 558                   • The `DO` loop iteration variable must be of type integer.
- 559                   • If used, the `END DO` directive must appear immediately after the end of the loop.
- 560                   • Only a single `SCHEDULE` clause can appear on a `DO` directive.
- 561                   • Only a single `ORDERED` clause can appear on a `DO` directive.
- 562                   • *chunk* must be a positive scalar integer expression.
- 563                   • The value of the *chunk* parameter must be the same for all of the threads in the  
564                    team.

### 565   2.3.2 SECTIONS *Directive*

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

569 The format of this directive is as follows:

```

570 !OMP SECTIONS [clause[[,] clause]. . .]
571 [!OMP SECTION]
572 block
573 [!OMP SECTION
574 block]
575 . . .
576 !OMP END SECTIONS [NOWAIT]

```

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

578 *clause* can be one of the following:

- 579 • PRIVATE(*list*)
- 580 • FIRSTPRIVATE(*list*)
- 581 • LASTPRIVATE(*list*)
- 582 • REDUCTION( { *operator* | *intrinsic\_procedure\_name* } : *list*)

583 The PRIVATE, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses are described  
584 in Section 2.6.2, page 30.

585 Each section is preceded by a SECTION directive, though the SECTION directive is  
586 optional for the first section. The SECTION directives must appear within the lexical  
587 extent of the SECTIONS/END SECTIONS directive pair. The last section ends at the  
588 END SECTIONS directive. Threads that complete execution of their sections wait at a  
589 barrier at the END SECTIONS directive unless a NOWAIT is specified.

590 The following restrictions apply to the SECTIONS directive:

- 591 • The code enclosed in a SECTIONS/END SECTIONS directive pair must be a  
592 structured block. In addition, each constituent section must also be a structured  
593 block. It is non-compliant to branch into or out of the constituent section blocks.
- 594 • It is non-compliant for a SECTION directive to be outside the lexical extent of the  
595 SECTIONS/END SECTIONS directive pair. (See Section A.8, page 61 for an example  
596 that uses these directives.)



### 597 **2.3.3 SINGLE Directive**

598 The SINGLE directive specifies that the enclosed code is to be executed by only one  
599 thread in the team. Threads in the team that are not executing the SINGLE directive  
600 wait at the END SINGLE directive unless NOWAIT is specified.

601 The format of this directive is as follows:

```
602 !$OMP SINGLE [clause[[,] clause]. . . ]  
603 block  
604 !$OMP END SINGLE [end_single_modifier]
```

605 where *end\_single\_modifier* is either COPYPRIVATE(*list*) [[,] COPYPRIVATE(*list*) . . . ]  
606 or NOWAIT.

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

608 *clause* can be one of the following:

- 609 • PRIVATE(*list*)
- 610 • FIRSTPRIVATE(*list*)

611 The PRIVATE, FIRSTPRIVATE, and COPYPRIVATE clauses are described in Section  
612 2.6.2, page 30.

613 The following restriction applies to the SINGLE directive:

- 614 • The code enclosed in a SINGLE/END SINGLE directive pair must be a structured  
615 block. It is non-compliant to branch into or out of the block.

616 See Section A.9, page 61 for an example of the SINGLE directive.

### 617 **2.3.4 WORKSHARE Directive**

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

```

622 !$OMP WORKSHARE [NOWAIT]
623
624 block
625
626 !$OMP END WORKSHARE [NOWAIT]

```

625 A BARRIER is implied following the enclosed code unless the NOWAIT clause is  
626 specified on the END WORKSHARE directive.

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

- 628 • For array expressions within each statement, including transformational intrinsics  
629 that compute scalar values from arrays:
  - 630 – Evaluation of each element of the array expression is a unit of work.
  - 631 – Evaluation of transformational intrinsics may be freely subdivided into any  
632 number of units of work.
- 633 • If a WORKSHARE directive is applied to an array assignment statement, the  
634 assignment of each element is a unit of work.
- 635 • If a WORKSHARE directive is applied to a scalar assignment statement, the  
636 assignment operation is a single unit of work.
- 637 • If any actual argument in a reference to an elemental function is an array, the  
638 reference is treated in the same way as if the function had been applied separately  
639 to corresponding elements of each array actual argument. When a WORKSHARE  
640 directive is applied to a reference to an elemental function, each application of the  
641 function to corresponding elements of any array argument is treated as a unit of  
642 work.
- 643 • If a WORKSHARE directive is applied to a WHERE statement or construct, the  
644 evaluation of the mask expression and the masked assignments are workshared.
- 645 • If a WORKSHARE directive is applied to a FORALL statement or construct, the  
646 evaluation of the mask expression, expressions occurring in the specification of the  
647 iteration space, and the masked assignments are workshared.

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

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

653 The WORKSHARE directive causes the sharing of work to occur only in the lexically  
654 enclosed block. If these statements cause a function to be invoked, the WORKSHARE  
655 directive does not cause work to be shared while executing that subprogram.

- 656           The following restrictions apply to the `WORKSHARE` directive:
- 657           • *block* must only contain array assignment statements, scalar assignment  
658           statements, `FORALL` statements, `FORALL` constructs, `WHERE` statements or `WHERE`  
659           constructs.
  - 660           • *block* must not contain any user defined function calls unless the function is  
661           ELEMENTAL.
  - 662           • The code enclosed in a `WORKSHARE/END WORKSHARE` directive pair must be a  
663           structured block. It is non-compliant to branch into or out of the block.

## 664   2.4 Combined Parallel Work-sharing Constructs

665           The combined parallel work-sharing constructs are shortcuts for specifying a parallel  
666           region that contains only one work-sharing construct. The semantics of these  
667           directives are identical to that of explicitly specifying a `PARALLEL` directive followed  
668           by a single work-sharing construct.

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

- 670           • Section 2.4.1, page 19, describes the `PARALLEL DO` and `END PARALLEL DO`  
671           directives.
- 672           • Section 2.4.2, page 20, describes the `PARALLEL SECTIONS` and  
673           `END PARALLEL SECTIONS` directives.
- 674           • Section 2.4.3, page 21, describes the `PARALLEL WORKSHARE` and  
675           `END PARALLEL WORKSHARE` directives.

### 676   2.4.1 `PARALLEL DO` Directive

677           The `PARALLEL DO` directive provides a shortcut form for specifying a parallel region  
678           that contains a single `DO` directive. (See also Section A.1, page 57, for an example.)

679           The format of this directive is as follows:

```
680           !$OMP PARALLEL DO [clause[[,] clause]. . .]  
681            do_loop  
682           [!$OMP END PARALLEL DO]
```

683       The *do\_loop* may be a *do\_construct*, an *outer\_shared\_do\_construct*, or an  
 684       *inner\_shared\_do\_construct*. A DO construct that contains several DO statements that  
 685       share the same DO termination statement syntactically consists of a sequence of  
 686       *outer\_shared\_do\_constructs*, followed by a single *inner\_shared\_do\_construct*. If an END  
 687       PARALLEL DO directive follows such a DO construct, a PARALLEL DO directive can  
 688       only be specified for the first (i.e., the outermost) *outer\_shared\_do\_construct*. (See  
 689       Section A.22, page 74 for examples.)

690       *clause* can be one of the clauses accepted by either the PARALLEL or the DO directive.  
 691       For information about the PARALLEL directive and the IF clause, see Section 2.2,  
 692       page 9. For information about the DO directive and the SCHEDULE and ORDERED  
 693       clauses, see Section 2.3.1, page 13. For information on the remaining clauses, see  
 694       Section 2.6.2, page 30.

695       If the END PARALLEL DO directive is not specified, the PARALLEL DO ends with the  
 696       DO loop that immediately follows the PARALLEL DO directive. If used, the  
 697       END PARALLEL DO directive must appear immediately after the end of the DO loop.

698       The semantics are identical to explicitly specifying a PARALLEL directive immediately  
 699       followed by a DO directive.

#### 700 **2.4.2 PARALLEL SECTIONS Directive**

701       The PARALLEL SECTIONS directive provides a shortcut form for specifying a parallel  
 702       region that contains a single SECTIONS directive. The semantics are identical to  
 703       explicitly specifying a PARALLEL directive immediately followed by a SECTIONS  
 704       directive.

705       The format of this directive is as follows:

```

706       !$OMP PARALLEL SECTIONS [clause[[,] clause]. . .]
707       [!$OMP SECTION ]
708       block
709       [!$OMP SECTION
710       block]
711       . . .
712       !$OMP END PARALLEL SECTIONS
  
```

713 *clause* can be one of the clauses accepted by either the `PARALLEL` or the `SECTIONS`  
 714 directive. For more information about the `PARALLEL` directive, see Section 2.2, page  
 715 9. For more information about the `SECTIONS` directive, see Section 2.3.2, page 15.  
 716 The `PRIVATE`, `FIRSTPRIVATE`, `LASTPRIVATE`, and `REDUCTION` clauses are described  
 717 in Section 2.6.2, page 30.

718 The last section ends at the `END PARALLEL SECTIONS` directive.

### 719 **2.4.3 `PARALLEL WORKSHARE` Directive**

720 The `PARALLEL WORKSHARE` directive provides a shortcut form for specifying a  
 721 parallel region that contains a single `WORKSHARE` directive. The semantics are  
 722 identical to explicitly specifying a `PARALLEL` directive immediately followed by a  
 723 `WORKSHARE` directive.

724 The format of this directive is as follows:

```
725 !$OMP PARALLEL WORKSHARE [clause[,] clause]. . . ]
726
727 block
728 !$OMP END PARALLEL WORKSHARE
```

728 *clause* can be one of the clauses accepted by either the `PARALLEL` or the `WORKSHARE`  
 729 directive. For more information about the `PARALLEL` directive, see Section 2.2, page  
 730 9. For more information about the `WORKSHARE` directive, see Section 2.3.4, page 17.

## 731 **2.5 Synchronization Constructs and the `MASTER` Directive**

732 The following sections describe the synchronization constructs and the `MASTER`  
 733 directive:

- 734 • Section 2.5.1, page 22, describes the `MASTER` and `END MASTER` directives.
- 735 • Section 2.5.2, page 22, describes the `CRITICAL` and `END CRITICAL` directives.
- 736 • Section 2.5.3, page 23, describes the `BARRIER` directive.
- 737 • Section 2.5.4, page 23, describes the `ATOMIC` directive.
- 738 • Section 2.5.5, page 25, describes the `FLUSH` directive.
- 739 • Section 2.5.6, page 26, describes the `ORDERED` and `END ORDERED` directives.

740 **2.5.1 MASTER Directive**

741 The code enclosed within MASTER and END MASTER directives is executed by the  
742 master thread of the team.

743 The format of this directive is as follows:

```
744 !$OMP MASTER
745
746 block
747
748 !$OMP END MASTER
```

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

750 The following restriction applies to the MASTER directive:

- 751 • The section of code enclosed by MASTER and END MASTER directives must be a  
752 structured block. It is non-compliant to branch into or out of the block.

753 **2.5.2 CRITICAL Directive**

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

756 The format of this directive is as follows:

```
757 !$OMP CRITICAL [(name)]
758
759 block
760
761 !$OMP END CRITICAL [(name)]
```

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

761 A thread waits at the beginning of a critical section until no other thread in the team  
762 is executing a critical section with the same name. All unnamed CRITICAL directives  
763 map to the same name. Critical section names are global entities of the program. If a  
764 name conflicts with any other entity, the behavior of the program is unspecified.

765 The following restrictions apply to the CRITICAL directive:

- 766           • The section of code enclosed by the `CRITICAL` and `END CRITICAL` directive pair  
767           must be a structured block. It is non-compliant to branch into or out of the block.
- 768           • If a *name* is specified on a `CRITICAL` directive, the same *name* must also be  
769           specified on the `END CRITICAL` directive. If no *name* appears on the `CRITICAL`  
770           directive, no *name* can appear on the `END CRITICAL` directive.
- 771           See Section A.5, page 58, for an example that uses named `CRITICAL` sections.

### 772   **2.5.3 BARRIER Directive**

773           The `BARRIER` directive synchronizes all the threads in a team. When encountered,  
774           each thread waits until all of the others threads in that team have reached this point.

775           The format of this directive is as follows:

```
776           !$OMP BARRIER
```

777           The following restrictions apply to the `BARRIER` directive:

- 778           • Work-sharing constructs and `BARRIER` directives must be encountered by all  
779           threads in a team or by none at all.
- 780           • Work-sharing constructs and `BARRIER` directives must be encountered in the same  
781           order by all threads in a team.

### 782   **2.5.4 ATOMIC Directive**

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

786           The format of this directive is as follows:

```
787           !$OMP ATOMIC
```

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

```

790   x = x operator expr
791   x = expr operator x
792   x = intrinsic_procedure_name (x, expr_list)
793   x = intrinsic_procedure_name (expr_list, x)

```

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.
- *operator* is one of +, \*, -, /, .AND., .OR., .EQV., or .NEQV. .
- The operators in *expr* must have precedence equal to or greater than the 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.

This directive permits optimization beyond that of the necessary critical section around the assignment. An implementation can replace all ATOMIC directives by enclosing the statement in a critical section. All of these critical sections must use the same unique name.

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

The following restriction applies to the ATOMIC directive:

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

**Example:**

```

820   !$OMP ATOMIC
821     Y(INDEX(I)) = Y(INDEX(I)) + B

```



822 See Section A.12, page 63, and Section A.23, page 75, for more examples using the  
823 `ATOMIC` directive.

### 824 **2.5.5 FLUSH Directive**

825 The `FLUSH` directive, whether explicit or implied, identifies a cross-thread sequence  
826 point at which the implementation is required to ensure that each thread in the team  
827 has a consistent view of certain variables in memory.

828 A consistent view requires that all memory operations (both reads and writes) that  
829 occur before the `FLUSH` directive in the program be performed before the sequence  
830 point in the executing thread; similarly, all memory operations that occur after the  
831 `FLUSH` must be performed after the sequence point in the executing thread.

832 Thread-visible variables are the following data items:

- 833 • Globally visible variables (in common blocks and in modules).
- 834 • Variables visible through host association.
- 835 • Local variables that have the `SAVE` attribute.
- 836 • Variables that appear in an `EQUIVALENCE` statement with a thread-visible  
837 variable.
- 838 • Local variables that do not have the `SAVE` attribute but have had their address  
839 taken and saved or have had their address passed to another subprogram.
- 840 • Local variables that do not have the `SAVE` attribute that are declared shared in a  
841 parallel region within the subprogram.
- 842 • Dummy arguments.
- 843 • All pointer dereferences.

844 Implementations must ensure that modifications made to thread-visible variables  
845 within the executing thread are made visible to all other threads at the sequence  
846 point. For example, compilers must restore values from registers to memory, and  
847 hardware may need to flush write buffers. Furthermore, implementations must  
848 assume that thread-visible variables may have been updated by other threads at the  
849 sequence point and must be retrieved from memory before their first use past the  
850 sequence point.

851 Finally, the `FLUSH` directive only provides consistency between operations within the  
852 executing thread and global memory. To achieve a globally consistent view across all  
853 threads, each thread must execute a `FLUSH` operation.

854 The format of this directive is as follows:

855 `!$OMP FLUSH [( list )]`

856 This directive must appear at the precise point in the code at which the  
 857 synchronization is required. The optional *list* argument consists of a  
 858 comma-separated list of variables that need to be flushed in order to avoid flushing  
 859 all variables. The *list* should contain only named variables (see Section A.13, page  
 860 63). The `FLUSH` directive is implied for the following directives:

- 861 • `BARRIER`
- 862 • `CRITICAL` and `END CRITICAL`
- 863 • `PARALLEL`
- 864 • `PARALLEL DO`
- 865 • `PARALLEL SECTIONS`
- 866 • `END PARALLEL DO`
- 867 • `END PARALLEL SECTIONS`
- 868 • `END DO`
- 869 • `END PARALLEL`
- 870 • `END SECTIONS`
- 871 • `END SINGLE`
- 872 • `ORDERED` and `END ORDERED`

873 The directive is not implied if a `NOWAIT` clause is present.

#### 874 **2.5.6 ORDERED Directive**

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

877 The format of this directive is as follows:

878 `!$OMP ORDERED`

879 *block*

880 `!$OMP END ORDERED`

881 An `ORDERED` directive can appear only in the dynamic extent of a `DO` or `PARALLEL DO`  
882 directive. The `DO` directive to which the ordered section binds must have the `ORDERED`  
883 clause specified (see Section 2.3.1, page 13). One thread is allowed in an ordered  
884 section at a time. Threads are allowed to enter in the order of the loop iterations. No  
885 thread can enter an ordered section until it is guaranteed that all previous iterations  
886 have completed or will never execute an ordered section. This sequentializes and  
887 orders code within ordered sections while allowing code outside the section to run in  
888 parallel. `ORDERED` sections that bind to different `DO` directives are independent of  
889 each other.

890 The following restrictions apply to the `ORDERED` directive:

- 891 • The code enclosed by the `ORDERED` and `END ORDERED` directives must be a  
892 structured block. It is non-compliant to branch into or out of the block.
- 893 • An `ORDERED` directive cannot bind to a `DO` directive that does not have the  
894 `ORDERED` clause specified.
- 895 • An iteration of a loop with a `DO` directive must not execute the same `ORDERED`  
896 directive more than once, and it must not execute more than one `ORDERED`  
897 directive.

898 See Section A.10, page 62, and Section A.24, page 76, for examples using the  
899 `ORDERED` directive.

## 900 2.6 Data Environment Constructs

901 This section presents constructs for controlling the data environment during the  
902 execution of parallel constructs:

- 903 • Section 2.6.1, page 27, describes the `THREADPRIVATE` directive, which makes  
904 common blocks or variables local to a thread.
- 905 • Section 2.6.2, page 30, describes directive clauses that affect the data environment.
- 906 • Section 2.6.3, page 38 describes the data environment rules.

### 907 2.6.1 `THREADPRIVATE` Directive

908 The `THREADPRIVATE` directive makes named common blocks and named variables  
909 private to a thread but global within the thread.

910 This directive must appear in the declaration section of a scoping unit in which the  
911 common block or variable is declared. Although variables in common blocks can be

912 accessed by use association or host association, common block names cannot. This  
913 means that a common block name specified in a `THREADPRIVATE` directive must be  
914 declared to be a common block in the same scoping unit in which the `THREADPRIVATE`  
915 directive appears. Each thread gets its own copy of the common block or variable, so  
916 data written to the common block or variable by one thread is not directly visible to  
917 other threads. During serial portions and `MASTER` sections of the program, accesses  
918 are to the master thread's copy of the common block or variable. (See Section A.25,  
919 page 77 for examples.)

920 On entry to the first parallel region, an instance of a variable or common block that  
921 appears in a `THREADPRIVATE` directive is created for each thread. A variable is said  
922 to be affected by a `COPYIN` clause if the variable appears in the `COPYIN` clause or it is  
923 in a common block that appears in the `COPYIN` clause. If a `THREADPRIVATE` variable  
924 or a variable in a `THREADPRIVATE` common block is not affected by any `COPYIN` clause  
925 that appears on the first parallel region in a program, the variable or any subobject of  
926 the variable is initially defined or undefined according to the following rules:

- 927 • If it has the `ALLOCATABLE` attribute, each copy created will have an initial  
928 allocation status of not currently allocated.
- 929 • If it has the `POINTER` attribute:
  - 930 – if it has an initial association status of disassociated, either through explicit  
931 initialization or default initialization, each copy created will have an  
932 association status of disassociated;
  - 933 – otherwise, each copy created will have an association status of undefined.
- 934 • If it does not have either the `POINTER` or the `ALLOCATABLE` attribute:
  - 935 – if it is initially defined, either through explicit initialization or default  
936 initialization, each copy created is so defined;
  - 937 – otherwise, each copy created is undefined.

938 On entry to a subsequent region, if the dynamic threads mechanism has been  
939 disabled, the definition, association or allocation status of a thread's copy of a  
940 `THREADPRIVATE` variable or a variable in a `THREADPRIVATE` common block, that is  
941 not affected by any `COPYIN` clause that appears on the region, will be retained, and if  
942 it was defined, its value will be retained as well. In this case, if a `THREADPRIVATE`  
943 variable is referenced in both regions, then threads with the same thread number in  
944 their respective regions will reference the same copy of that variable. If the dynamic  
945 threads mechanism is enabled, the definition and association status of a thread's copy  
946 of the variable is undefined, and the allocation status of an allocatable array will be  
947 implementation dependent. A variable with the allocatable attribute must not appear  
948 in a `COPYIN` clause, although a structure that has an ultimate component with the  
949 allocatable attribute may appear in a `COPYIN` clause. For more information on

950 dynamic threads, see the `OMP_SET_DYNAMIC` library routine, Section 3.1.7, page 46,  
951 and the `OMP_DYNAMIC` environment variable, Section 4.3, page 56.

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

- 955 • If it has the `POINTER` attribute:
  - 956 – if the master thread's copy is associated with a target that each copy can  
957 become associated with, each copy will become associated with the same target;
  - 958 – if the master thread's copy is disassociated, each copy will become disassociated;
  - 959 – otherwise, each copy will have an undefined association status.
- 960 • If it does not have the `POINTER` attribute, each copy becomes defined with the  
961 value of the master thread's copy as if by an intrinsic assignment.

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

964 The format of this directive is as follows:

965 

<code>!\$OMP THREADPRIVATE( <i>list</i> )</code>
--

966 where *list* is a comma-separated list of named variables and named common blocks.  
967 Any common block name must appear between slashes.

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

- 969 • The `THREADPRIVATE` directive must appear after every declaration of a thread  
970 private common block.
- 971 • A blank common block cannot appear in a `THREADPRIVATE` directive.
- 972 • It is non-compliant for a `THREADPRIVATE` variable or common block or its  
973 constituent variables to appear in any clause other than a `COPYIN` clause or a  
974 `COPYPRIVATE` clause. As a result, they are not permitted in a `PRIVATE`,  
975 `FIRSTPRIVATE`, `LASTPRIVATE`, `SHARED`, or `REDUCTION` clause. They are not  
976 affected by the `DEFAULT` clause.
- 977 • A variable can only appear in a `THREADPRIVATE` directive in the scope in which it  
978 is declared. It must not be part of a common block or be declared in an  
979 `EQUIVALENCE` statement.
- 980 • A variable that appears in a `THREADPRIVATE` directive and is not declared in the  
981 scope of a module must have the `SAVE` attribute.

## 982 2.6.2 Data Scope Attribute Clauses

983 Several directives accept clauses that allow a user to control the scope attributes of  
984 variables for the duration of the construct. Not all of the following clauses are  
985 allowed on all directives, but the clauses that are valid on a particular directive are  
986 included with the description of the directive. If no data scope clauses are specified  
987 for a directive, the default scope for variables affected by the directive is `SHARED`. (See  
988 Section 2.6.3, page 38, for exceptions.)

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

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

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

1000 When a named common block appears in a list, it has the same meaning as if every  
1001 explicit member of the common block appeared in the list. A member of a common  
1002 block is an explicit member if it is named in a `COMMON` statement which declares the  
1003 common block, and it was declared in the same scoping unit in which the clause  
1004 appears.

1005 Although variables in common blocks can be accessed by use association or host  
1006 association, common block names cannot. This means that a common block name  
1007 specified in a data scope attribute clause must be declared to be a common block in  
1008 the same scoping unit in which the data scope attribute clause appears.

1009 The following sections describe the data scope attribute clauses:

- 1010 • Section 2.6.2.1, page 31, describes the `PRIVATE` clause.
- 1011 • Section 2.6.2.2, page 32, describes the `SHARED` clause.
- 1012 • Section 2.6.2.3, page 32, describes the `DEFAULT` clause.
- 1013 • Section 2.6.2.4, page 33, describes the `FIRSTPRIVATE` clause.
- 1014 • Section 2.6.2.5, page 33, describes the `LASTPRIVATE` clause.
- 1015 • Section 2.6.2.6, page 34, describes the `REDUCTION` clause.
- 1016 • Section 2.6.2.7, page 36, describes the `COPYIN` clause.
- 1017 • Section 2.6.2.8, page 37, describes the `COPYPRIVATE` clause.

## 1018 2.6.2.1 PRIVATE Clause

1019 The PRIVATE clause declares the variables in *list* to be private to each thread in a  
1020 team.

1021 This clause has the following format:

1022 

PRIVATE ( <i>list</i> )
-------------------------

1023 The behavior of a variable declared in a PRIVATE clause is as follows:

- 1024 1. A new object of the same type is declared once for each thread in the team. One  
1025 thread in the team is permitted, but not required, to re-use the existing storage  
1026 as the storage for the new object. For all other threads, new storage is created  
1027 for the new object.
- 1028 2. All references to the original object in the lexical extent of the directive construct  
1029 are replaced with references to the private object.
- 1030 3. Variables declared as PRIVATE are undefined for each thread on entering the  
1031 construct, and the corresponding shared variable is undefined on exit from a  
1032 parallel construct.
- 1033 4. A variable declared as PRIVATE may be storage-associated with other variables  
1034 when the PRIVATE clause is encountered. Storage association may exist because  
1035 of constructs such as EQUIVALENCE, COMMON, etc. If *a* is a variable appearing in  
1036 a PRIVATE clause and *b* is a variable which was storage-associated with *a*, then:
  - 1037 a. The contents, allocation, and association status of *b* are undefined on entry  
1038 to the parallel construct.
  - 1039 b. Any definition of *a*, or of its allocation or association status, causes the  
1040 contents, allocation, and association status of *b* to become undefined.
  - 1041 c. Any definition of *b*, or of its allocation or association status, causes the  
1042 contents, allocation, and association status of *a* to become undefined.

1043 See Section A.20, page 71 and Section A.21, page 71, for examples.

- 1044 5. If a variable is declared as PRIVATE, and the variable is referenced in the  
1045 definition of a statement function, and the statement function is used within the  
1046 lexical extent of the directive construct, then the statement function may  
1047 reference either the SHARED version of the variable or the PRIVATE version.  
1048 Which version is referenced is implementation-dependent.

1049 2.6.2.2 SHARED *Clause*

1050 The SHARED clause makes variables that appear in the *list* shared among all the  
 1051 threads in a team. All threads within a team access the same storage area for  
 1052 SHARED data.

1053 This clause has the following format:

1054 

SHARED ( <i>list</i> )
------------------------

1055 2.6.2.3 DEFAULT *Clause*

1056 The DEFAULT clause allows the user to specify a PRIVATE, SHARED, or NONE scope  
 1057 attribute for all variables in the lexical extent of any parallel region. Variables in  
 1058 THREADPRIVATE common blocks are not affected by this clause.

1059 This clause has the following format:

1060 

DEFAULT ( PRIVATE   SHARED   NONE )
-------------------------------------

1061 The PRIVATE, SHARED, and NONE specifications have the following effects:

- 1062 • Specifying DEFAULT (PRIVATE) makes all named objects in the lexical extent of  
 1063 the parallel region, including common block variables but excluding  
 1064 THREADPRIVATE variables, private to a thread as if each variable were listed  
 1065 explicitly in a PRIVATE clause.
- 1066 • Specifying DEFAULT (SHARED) makes all named objects in the lexical extent of the  
 1067 parallel region shared among the threads in a team, as if each variable were listed  
 1068 explicitly in a SHARED clause. In the absence of an explicit DEFAULT clause, the  
 1069 default behavior is the same as if DEFAULT (SHARED) were specified.
- 1070 • Specifying DEFAULT (NONE) requires that each variable used in the lexical extent  
 1071 of the parallel region be explicitly listed in a data scope attribute clause on the  
 1072 parallel region, unless it is one of the following:
  - 1073 – THREADPRIVATE.
  - 1074 – A Cray pointee.
  - 1075 – A loop iteration variable used only as a loop iteration variable for sequential  
 1076 loops in the lexical extent of the region or parallel DO loops that bind to the  
 1077 region.
  - 1078 – Only used in work-sharing constructs that bind to the region, and is specified  
 1079 in a data scope attribute clause for each such construct.



1080 **Only one** DEFAULT clause can be specified on a PARALLEL directive.

1081 **Variables can be exempted from a defined default using the** PRIVATE, SHARED,  
1082 FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses. As a result, the following  
1083 example is legal:

```
1084 !$OMP PARALLEL DO DEFAULT(PRIVATE), FIRSTPRIVATE(I), SHARED(X),  
1085 !$OMP& SHARED(R) LASTPRIVATE(I)
```

#### 1086 2.6.2.4 FIRSTPRIVATE Clause

1087 The FIRSTPRIVATE clause provides a superset of the functionality provided by the  
1088 PRIVATE clause.

1089 This clause has the following format:

```
1090 FIRSTPRIVATE ( list )
```

1091 Variables that appear in the *list* are subject to PRIVATE clause semantics described in  
1092 Section 2.6.2.1, page 31. In addition, private copies of the variables are initialized  
1093 from the original object existing before the construct.

#### 1094 2.6.2.5 LASTPRIVATE Clause

1095 The LASTPRIVATE clause provides a superset of the functionality provided by the  
1096 PRIVATE clause.

1097 This clause has the following format:

```
1098 LASTPRIVATE ( list )
```

1099 Variables that appear in the *list* are subject to the PRIVATE clause semantics  
1100 described in Section 2.6.2.1, page 31. When the LASTPRIVATE clause appears on a DO  
1101 directive, the thread that executes the sequentially last iteration updates the version  
1102 of the object it had before the construct (see Section A.6, page 59 for an example).  
1103 When the LASTPRIVATE clause appears in a SECTIONS directive, the thread that  
1104 executes the lexically last SECTION updates the version of the object it had before the  
1105 construct. Subobjects that are not assigned a value by the last iteration of the DO or  
1106 the lexically last SECTION of the SECTIONS directive are undefined after the construct.

1107 2.6.2.6 REDUCTION *Clause*

1108 This clause performs a reduction on the variables that appear in *list*, with the  
 1109 operator *operator* or the intrinsic *intrinsic\_procedure\_name*, where *operator* is one of  
 1110 the following: +, \*, -, .AND., .OR., .EQV., or .NEQV., and *intrinsic\_procedure\_name*  
 1111 refers to one of the following: MAX, MIN, IAND, IOR, or IEOR.

1112 This clause has the following format:

1113 REDUCTION( { <i>operator</i>   <i>intrinsic_procedure_name</i> } : <i>list</i> )
---

1114 Variables in *list* must be named variables of intrinsic type. Deferred shape, assumed  
 1115 shape, and assumed size arrays are not allowed on the reduction clause. Since the  
 1116 intermediate values of the REDUCTION variables may be combined in random order,  
 1117 there is no guarantee that bit-identical results will be obtained for either integer or  
 1118 floating point reductions from one parallel run to another.

1119 Variables that appear in a REDUCTION clause must be SHARED in the enclosing  
 1120 context. A private copy of each variable in *list* is created for each thread as if the  
 1121 PRIVATE clause had been used. The private copy is initialized according to the  
 1122 operator. See Table 2, page 35, for more information.

1123 At the end of the REDUCTION, the shared variable is updated to reflect the result of  
 1124 combining the original value of the (shared) reduction variable with the final value of  
 1125 each of the private copies using the operator specified. The reduction operators are all  
 1126 associative (except for subtraction), and the compiler can freely reassociate the  
 1127 computation of the final value (the partial results of a subtraction reduction are  
 1128 added to form the final value).

1129 The value of the shared variable becomes undefined when the first thread reaches the  
 1130 containing clause, and it remains so until the reduction computation is complete.  
 1131 Normally, the computation is complete at the end of the REDUCTION construct;  
 1132 however, if the REDUCTION clause is used on a construct to which NOWAIT is also  
 1133 applied, the shared variable remains undefined until a barrier synchronization has  
 1134 been performed to ensure that all the threads have completed the REDUCTION clause.

1135 The REDUCTION clause is intended to be used on a region or work-sharing construct  
 1136 in which the reduction variable or a subobject of the reduction variable is used only in  
 1137 reduction statements with one of the following forms:

```

1138   x = x operator expr
1139   x = expr operator x (except for subtraction)
1140   x = intrinsic_procedure_name (x, expr_list)
1141   x = intrinsic_procedure_name (expr_list, x)

```

1142 In the preceding statements:

- 1143 • *x* is a scalar variable of intrinsic type.
- 1144 • *expr* is a scalar expression that does not reference *x*.
- 1145 • *expr\_list* is a comma-separated, non-empty list of scalar expressions that do not  
1146 reference *x*. When *intrinsic\_procedure\_name* refers to IAND, IOR, or IEOR, exactly  
1147 one expression must appear in *expr\_list*.
- 1148 • *intrinsic\_procedure\_name* is one of MAX, MIN, IAND, IOR, or IEOR.
- 1149 • *operator* is one of +, \*, -, .AND., .OR., .EQV., or .NEQV. .
- 1150 • The operators in *expr* must have precedence equal to or greater than the  
1151 precedence of *operator*, *x operator expr* must be mathematically equivalent to *x*  
1152 *operator (expr)*, and *expr operator x* must be mathematically equivalent to  
1153 *(expr) operator x*.
- 1154 • The function *intrinsic\_procedure\_name*, the operator *operator*, and the assignment  
1155 must be the intrinsic procedure name, the intrinsic operator, and intrinsic  
1156 assignment.

1157 Some reductions can be expressed in other forms. For instance, a MAX reduction  
1158 might be expressed as follows:

```
1159 IF (x .LT. expr) x = expr
```

1160 Alternatively, the reduction might be hidden inside a subroutine call. The user should  
1161 be careful that the operator specified in the REDUCTION clause matches the reduction  
1162 operation.

1163 The following table lists the operators and intrinsics that are valid and their  
1164 canonical initialization values. The actual initialization value will be consistent with  
1165 the data type of the reduction variable.

1166 Table 2. Reducation Variable Initialization Values

1167	<u>Operator/Intrinsic</u>	<u>Initialization</u>
1168	+	0
1169	*	1

1170	-	0
1171	.AND.	.TRUE.
1172	.OR.	.FALSE.
1173	.EQV.	.TRUE.
1174	.NEQV.	.FALSE.
1175	MAX	Smallest representable number
1176	MIN	Largest representable number
1177	IAND	All bits on
1178	IOR	0
1179	IEOR	0

1180 See Section A.7, page 59, for an example that uses the + operator.

1181 Any number of reduction clauses can be specified on the directive, but a variable can  
1182 appear only once in the REDUCTION clause(s) for that directive.

1183 Example:

```
1184 !$OMP DO REDUCTION(+: A, Y) REDUCTION(.OR.: AM)
```

### 1185 2.6.2.7 COPYIN Clause

1186 The COPYIN clause applies only to variables, common blocks and variables in common  
1187 blocks that are declared as THREADPRIVATE. A COPYIN clause on a parallel region  
1188 specifies that the data in the master thread of the team be copied to the thread  
1189 private copies of the common blocks or variables at the beginning of the parallel  
1190 region as described in Section 2.6.1, page 27.

1191 This clause has the following format:

```
1192 COPYIN( list )
```

1193 If a common block appears in a THREADPRIVATE directive, it is not necessary to  
1194 specify the whole common block. Named variables appearing in the THREADPRIVATE  
1195 common block can be specified in the *list*.

1196 Although variables in common blocks can be accessed by use association or host  
1197 association, common block names cannot. This means that a common block name  
1198 specified in a COPYIN clause must be declared to be a common block in the same  
1199 scoping unit in which the COPYIN clause appears. See Section A.25, page 77, for more  
1200 information.

1201           **In the following example, the common blocks `BLK1` and `FIELDS` are specified as**  
 1202           **thread private, but only one of the variables in common block `FIELDS` is specified to**  
 1203           **be copied in.**

```
1204           COMMON /BLK1/ SCRATCH
1205           COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD
1206           !$OMP THREADPRIVATE(/BLK1/, /FIELDS/)
1207           !$OMP PARALLEL DEFAULT(PRIVATE) COPYIN(/BLK1/, ZFIELD)
```

1208           **An OpenMP-compliant implementation is required to ensure that the value of each**  
 1209           **thread private copy is the same as the value of the master thread copy when the**  
 1210           **master thread reached the directive containing the `COPYIN` clause.**

### 1211    2.6.2.8 `COPYPRIVATE` Clause

1212           **The `COPYPRIVATE` clause uses a private variable to broadcast a value, or a pointer to**  
 1213           **a shared object, from one member of a team to the other members. It is an**  
 1214           **alternative to using a shared variable for the value, or pointer association, and is**  
 1215           **useful when providing such a shared variable would be difficult (for example, in a**  
 1216           **recursion requiring a different variable at each level).**

1217           **This clause has the following format:**

```
1218           COPYPRIVATE( [list ] )
```

1219           **Variables in the *list* must not appear in a `PRIVATE` or `FIRSTPRIVATE` clause for the**  
 1220           **`SINGLE` construct. If the directive is encountered in the dynamic extent of a parallel**  
 1221           **region, variables in the list must be private in the enclosing context. If a common**  
 1222           **block is specified, then it must be `THREADPRIVATE`, and the effect is the same as if**  
 1223           **the variable names in its common block object list were specified.**

1224           **The effect of the `COPYPRIVATE` clause on the variables in its list occurs after the**  
 1225           **execution of the code enclosed within the `SINGLE` construct, and before any threads in**  
 1226           **the team have left the barrier at the end of the construct. If the variable is not a**  
 1227           **pointer, then in all other threads in the team, that variable becomes defined (as if by**  
 1228           **assignment) with the value of the corresponding variable in the thread that executed**  
 1229           **the enclosed code. If the variable is a pointer, then in all other threads in the team,**  
 1230           **that variable becomes pointer associated (as if by pointer assignment) with the**  
 1231           **corresponding variable in the thread that executed the enclosed code. (See Section**  
 1232           **A.27, page 82, for examples of the `COPYPRIVATE` clause.)**

1233 **2.6.3 Data Environment Rules**

1234 A program that conforms to the OpenMP Fortran API must adhere to the following  
1235 rules and restrictions with respect to data scope:

1236 1. Sequential DO loop control variables in the lexical extent of a PARALLEL region  
1237 that would otherwise be SHARED based on default rules are automatically made  
1238 private on the PARALLEL directive. Sequential DO loop control variables with no  
1239 enclosing PARALLEL region are not made private automatically. It is up to the  
1240 user to guarantee that these indexes are private if the containing procedures are  
1241 called from a PARALLEL region.

1242 All implied DO loop control variables and FORALL indexes are automatically made  
1243 private at the enclosing implied DO or FORALL construct.

1244 2. Variables that are privatized in a parallel region may be privatized again on an  
1245 enclosed work-sharing directive. As a result, variables that appear in a PRIVATE  
1246 clause on a work-sharing directive may either have a shared or a private scope in  
1247 the enclosing parallel region. Variables that appear on the FIRSTPRIVATE,  
1248 LASTPRIVATE, and REDUCTION clauses on a work-sharing directive must have  
1249 shared scope in the enclosing parallel region.

1250 3. Variables that appear in a reduction list in a parallel region cannot be privatized  
1251 on an enclosed work-sharing directive.

1252 4. A variable that appears in a PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or  
1253 REDUCTION clause must be definable.

1254 5. Assumed-size and assumed-shape arrays cannot be declared PRIVATE,  
1255 FIRSTPRIVATE, or LASTPRIVATE. Array dummy arguments that are explicitly  
1256 shaped (including variably dimensioned) can be declared in any scoping clause.

1257 6. Fortran pointers and allocatable arrays can be declared PRIVATE or SHARED but  
1258 not FIRSTPRIVATE or LASTPRIVATE.

1259 Within a parallel region, the initial status of a private pointer is undefined.  
1260 Private pointers that become allocated during the execution of a parallel region  
1261 should be explicitly deallocated by the program prior to the end of the parallel  
1262 region to avoid memory leaks.

1263 The association status of a SHARED pointer becomes undefined upon entry to and  
1264 on exit from the parallel construct if it is associated with a target or a subobject  
1265 of a target that is in a PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION  
1266 clause inside the parallel construct. An allocatable array declared PRIVATE must  
1267 have an allocation status of “not currently allocated” on entry to and on exit from  
1268 the construct.

1269 7. PRIVATE or SHARED attributes can be declared for a Cray pointer but not for the  
1270 pointee. The scope attribute for the pointee is determined at the point of pointer

- 1271 definition. It is non-compliant to declare a scope attribute for a pointee. Cray  
1272 pointers may not be specified in `FIRSTPRIVATE` or `LASTPRIVATE` clauses.
- 1273 8. Scope clauses apply only to variables in the lexical extent of the directive on  
1274 which the clause appears, with the exception of variables passed as actual  
1275 arguments. Local variables in called routines that do not have the `SAVE` attribute  
1276 are `PRIVATE`. Common blocks and modules in called routines in the dynamic  
1277 extent of a parallel region always have an implicit `SHARED` attribute, unless they  
1278 are `THREADPRIVATE` common blocks. Local variables in called routines that have  
1279 the `SAVE` attribute are `SHARED`. (See Section A.26, page 80, for examples.)
- 1280 9. When a named common block is specified in a `PRIVATE`, `FIRSTPRIVATE`, or  
1281 `LASTPRIVATE` clause of a directive, none of its constituent elements may be  
1282 declared in another data scope attribute clause in that directive. It should be  
1283 noted that when individual members of a common block are privatized, the  
1284 storage of the specified variables is no longer associated with the storage of the  
1285 common block itself. (See Section A.25, page 77, for examples.)
- 1286 10. Variables that are not allowed in the `PRIVATE` and `SHARED` clauses are not  
1287 affected by `DEFAULT(PRIVATE)` or `DEFAULT(SHARED)` clauses, respectively.
- 1288 11. Clauses can be repeated as needed, but each variable and each named common  
1289 block can appear explicitly in only one clause per directive, with the following  
1290 exceptions:
- 1291 • A variable can be declared both `FIRSTPRIVATE` and `LASTPRIVATE`.
  - 1292 • Variables affected by the `DEFAULT` clause can be listed explicitly in a clause to  
1293 override the default specification.
- 1294 12. Variables that are declared `LASTPRIVATE` for a work-sharing directive for which  
1295 `NOWAIT` appears must not be used prior to a barrier.
- 1296 13. Variables that appear in namelist statements, in variable format expressions,  
1297 and in expressions for statement function definitions must not be specified in  
1298 `PRIVATE`, `FIRSTPRIVATE`, or `LASTPRIVATE` clauses.
- 1299 14. The shared variables that are specified in `REDUCTION` or `LASTPRIVATE` clauses  
1300 become defined at the end of the construct. Any concurrent uses or definitions of  
1301 those variables must be synchronized with the definition that occurs at the end  
1302 of the construct to avoid race conditions.
- 1303 15. If the following three conditions hold regarding an actual argument in a reference  
1304 to a non-intrinsic procedure, then any references to (or definitions of) the shared  
1305 storage that is associated with the dummy argument by any other thread must  
1306 be synchronized with the procedure reference to avoid possible race conditions:
- 1307 a. The actual argument is one of the following:
    - 1308 • A `SHARED` variable

- 1309                   • A subobject of a SHARED variable
- 1310                   • An object associated with a SHARED variable
- 1311                   • An object associated with a subobject of a SHARED variable
- 1312           b. The actual argument is also one of the following:
- 1313                   • An array section with a vector subscript
- 1314                   • An array section
- 1315                   • An assumed-shape array
- 1316                   • A pointer array
- 1317           c. The associated dummy argument for this actual argument is an
- 1318                   explicit-shape array or an assumed-size array.

1319           The situations described above may result in the value of the shared variable  
 1320           being copied into temporary storage before the procedure reference, and back out  
 1321           of the temporary storage into the actual argument storage after the procedure  
 1322           reference. This effectively results in references to and definitions of the storage  
 1323           during the procedure reference.

1324           16. An OpenMP compliant implementation must adhere to the following rule:

- 1325                   • If a variable is specified as FIRSTPRIVATE and LASTPRIVATE, the
- 1326                   implementation must ensure that the update required for LASTPRIVATE
- 1327                   occurs after all initializations for FIRSTPRIVATE.

1328           17. An implementation may generate references to any object that appears or an  
 1329           object in a common block that appears in a REDUCTION, FIRSTPRIVATE,  
 1330           LASTPRIVATE, COPYPRIVATE, or COPYIN clause, on entry to (for FIRSTPRIVATE  
 1331           and COPYIN) or exit from (for REDUCTION, LASTPRIVATE, and COPYPRIVATE) a  
 1332           construct. Except for an object with the pointer attribute in a COPYPRIVATE  
 1333           clause, if a reference to the object as the expression in an intrinsic assignment  
 1334           statement would give an exceptional value, or have undefined behavior, at that  
 1335           point in the program, then the generated reference may have the same behavior.

## 1336 2.7 Directive Binding

1337           An OpenMP compliant implementation must adhere to the following rules with  
 1338           respect to the dynamic binding of directives:

- 1339                   • A parallel region is available for binding purposes, whether it is serialized or
- 1340                   executed in parallel.



- 1341 • **The DO, SECTIONS, SINGLE, MASTER, BARRIER and WORKSHARE directives bind to**  
1342 **the dynamically enclosing PARALLEL directive, if one exists. (See Section A.19,**  
1343 **page 70 for an example.)**
- 1344 • **The ORDERED directive binds to the dynamically enclosing DO directive.**
- 1345 • **The ATOMIC directive enforces exclusive access with respect to ATOMIC directives**  
1346 **in all threads, not just the current team.**
- 1347 • **The CRITICAL directive enforces exclusive access with respect to CRITICAL**  
1348 **directives in all threads, not just the current team.**
- 1349 • **A directive can never bind to any directive outside the closest enclosing PARALLEL.**

## 1350 2.8 Directive Nesting

1351 An OpenMP compliant implementation must adhere to the following rules with  
1352 respect to the dynamic nesting of directives:

- 1353 • **A PARALLEL directive dynamically inside another PARALLEL directive logically**  
1354 **establishes a new team, which is composed of only the current thread, unless**  
1355 **nested parallelism is enabled.**
- 1356 • **DO, SECTIONS, SINGLE, and WORKSHARE directives that bind to the same**  
1357 **PARALLEL directive are not allowed to be nested one inside the other.**
- 1358 • **BLOCK WORKSHARE directives are not allowed to be nested within a DO, SECTIONS,**  
1359 **SINGLE, or WORKSHARE directive, including any WORKSHARE directives that are**  
1360 **implied by a BLOCK WORKSHARE directive, that binds to the same PARALLEL**  
1361 **directive.**
- 1362 • **DO, SECTIONS, SINGLE, and WORKSHARE directives are not permitted in the**  
1363 **dynamic extent of CRITICAL and MASTER directives.**
- 1364 • **BARRIER directives are not permitted in the dynamic extent of DO, SECTIONS,**  
1365 **SINGLE, MASTER, CRITICAL, and WORKSHARE directives.**
- 1366 • **MASTER directives are not permitted in the dynamic extent of DO, SECTIONS,**  
1367 **SINGLE, and WORKSHARE directives.**
- 1368 • **ORDERED sections are not allowed in the dynamic extent of CRITICAL sections.**
- 1369 • **Any directive set that is legal when executed dynamically inside a PARALLEL**  
1370 **region is also legal when executed outside a parallel region. When executed**  
1371 **dynamically outside a user-specified parallel region, the directive is executed with**  
1372 **respect to a team composed of only the master thread.**

1373           **See Section A.17, page 67, for legal examples of directive nesting, and Section A.18,**  
1374           **page 68, for invalid examples.**

1376 This section describes the OpenMP Fortran API run-time library routines that can be  
 1377 used to control and query the parallel execution environment. A set of general  
 1378 purpose lock routines is also provided.

1379 OpenMP Fortran API run-time library routines are external procedures. In the  
 1380 following descriptions, *scalar\_integer\_expression* is a default scalar integer expression,  
 1381 and *scalar\_logical\_expression* is a default scalar logical expression. The return values  
 1382 of these routines are also of default kind.

1383 Interface declarations for the OpenMP Fortran runtime library routines described in  
 1384 this chapter shall be provided in the form of a Fortran INCLUDE file named  
 1385 `omp_lib.h` or a Fortran 90 MODULE named `omp_lib`. This file must define the  
 1386 following:

- 1387 • The interfaces of all of the routines in this chapter.
- 1388 • The INTEGER PARAMETER `omp_lock_kind` that defines the KIND type  
 1389 parameters used for simple lock variables in the `OMP_*_LOCK` routines.
- 1390 • the INTEGER PARAMETER `omp_nest_lock_kind` that defines the KIND type  
 1391 parameters used for the nestable lock variables in the `OMP_*_NEST_LOCK` routines.
- 1392 • the INTEGER PARAMETER `openmp_version` with a value of the C preprocessor  
 1393 macro `_OPENMP` (see Section 2.1.3, page 8) that has the form `YYYYDD` where `YYYY`  
 1394 and `DD` are the year and month designations of the version of the OpenMP Fortran  
 1395 API that the implementation supports.

1396 See Appendix D, page 97, for examples of these files.

### 1397 3.1 Execution Environment Routines

1398 The following sections describe the execution environment routines:

- 1399 • Section 3.1.1, page 44, describes the `OMP_SET_NUM_THREADS` subroutine.
- 1400 • Section 3.1.2, page 44, describes the `OMP_GET_NUM_THREADS` function.
- 1401 • Section 3.1.3, page 45, describes the `OMP_GET_MAX_THREADS` function.
- 1402 • Section 3.1.4, page 45, describes the `OMP_GET_THREAD_NUM` function.
- 1403 • Section 3.1.5, page 46, describes the `OMP_GET_NUM_PROCS` function.
- 1404 • Section 3.1.6, page 46, describes the `OMP_IN_PARALLEL` function.

- 1405 • Section 3.1.7, page 46, describes the `OMP_SET_DYNAMIC` subroutine.
- 1406 • Section 3.1.8, page 47, describes the `OMP_GET_DYNAMIC` function.
- 1407 • Section 3.1.9, page 47, describes the `OMP_SET_NESTED` subroutine.
- 1408 • Section 3.1.10, page 48, describes the `OMP_GET_NESTED` function.

### 1409 3.1.1 `OMP_SET_NUM_THREADS` Subroutine

1410 The `OMP_SET_NUM_THREADS` subroutine sets the number of threads to use for  
 1411 subsequent parallel regions.

1412 The format of this subroutine is as follows:

```
1413 SUBROUTINE OMP_SET_NUM_THREADS(scalar_integer_expression)
```

1414 The value of the *scalar\_integer\_expression* must be positive. The effect of this function  
 1415 depends on whether dynamic adjustment of the number of threads is enabled. If  
 1416 dynamic adjustment is disabled, the value of the *scalar\_integer\_expression* is used as  
 1417 the number of threads for all subsequent parallel regions prior to the next call to this  
 1418 function; otherwise, the value is used as the maximum number of threads that will be  
 1419 used. This function has effect only when called from serial portions of the program. If  
 1420 it is called from a portion of the program where the `OMP_IN_PARALLEL` function  
 1421 returns `.TRUE.`, the behavior of this function is unspecified. For additional  
 1422 information on this subject, see the `OMP_SET_DYNAMIC` subroutine described in  
 1423 Section 3.1.7, page 46, and the `OMP_GET_DYNAMIC` function described in Section 3.1.8,  
 1424 page 47, and the example in Section A.11, page 62.

1425 Resource constraints on an OpenMP parallel program may change the number of  
 1426 threads that a user is allowed to create at different phases of a program's execution.  
 1427 When dynamic adjustment of the number of threads is enabled, requests for more  
 1428 threads than an implementation can support are satisfied by a smaller number of  
 1429 threads. If dynamic adjustment of the number of threads is disabled, the behavior of  
 1430 this function is implementation dependent.

1431 This call has precedence over the `OMP_NUM_THREADS` environment variable.

### 1432 3.1.2 `OMP_GET_NUM_THREADS` Function

1433 The `OMP_GET_NUM_THREADS` function returns the number of threads currently in the  
 1434 team executing the parallel region from which it is called.

1435           The format of this function is as follows:

1436           

INTEGER FUNCTION OMP\_GET\_NUM\_THREADS

1437           The `OMP_SET_NUM_THREADS` call and the `OMP_NUM_THREADS` environment variable  
1438           control the number of threads in a team. For more information on the  
1439           `OMP_SET_NUM_THREADS` call, see Section 3.1.1, page 44.

1440           If the number of threads has not been explicitly set by the user, the default is  
1441           implementation dependent. This function binds to the closest enclosing `PARALLEL`  
1442           directive. For more information on the `PARALLEL` directive, see Section 2.2, page 9.

1443           If this call is made from the serial portion of a program, or from a nested parallel  
1444           region that is serialized, this function returns 1. (See Section A.14, page 64 for an  
1445           example.)

### 1446   **3.1.3 OMP\_GET\_MAX\_THREADS Function**

1447           The `OMP_GET_MAX_THREADS` function returns the maximum value that can be  
1448           returned by calls to the `OMP_GET_NUM_THREADS` function. For more information on  
1449           `OMP_GET_NUM_THREADS`, see Section 3.1.2, page 44.

1450           The format of this function is as follows:

1451           

INTEGER FUNCTION OMP\_GET\_MAX\_THREADS

1452           If `OMP_SET_NUM_THREADS` is used to change the number of threads, subsequent calls  
1453           to `OMP_GET_MAX_THREADS` will return the new value. This function can be used to  
1454           allocate maximum sized per-thread data structures when the `OMP_SET_DYNAMIC`  
1455           subroutine is set to `.TRUE.` For more information on `OMP_SET_DYNAMIC`, see Section  
1456           3.1.7, page 46.

1457           This function has global scope and returns the maximum value whether executing  
1458           from a serial region or a parallel region.

### 1459   **3.1.4 OMP\_GET\_THREAD\_NUM Function**

1460           The `OMP_GET_THREAD_NUM` function returns the thread number, within the team,  
1461           that lies between 0 and `OMP_GET_NUM_THREADS-1`, inclusive. (See the second  
1462           example in Section A.14, page 64.) The master thread of the team is thread 0.

1463           The format of this function is as follows:

1464 INTEGER FUNCTION OMP\_GET\_THREAD\_NUM

1465 This function binds to the closest enclosing `PARALLEL` directive. For more information  
1466 on the `PARALLEL` directive, see Section 2.2, page 9.

1467 When called from a serial region, `OMP_GET_THREAD_NUM` returns 0. When called from  
1468 within a nested parallel region that is serialized, this function returns 0.

### 1469 3.1.5 OMP\_GET\_NUM\_PROCS Function

1470 The `OMP_GET_NUM_PROCS` function returns the number of processors that are  
1471 available to the program.

1472 The format of this function is as follows:

1473 INTEGER FUNCTION OMP\_GET\_NUM\_PROCS

### 1474 3.1.6 OMP\_IN\_PARALLEL Function

1475 The `OMP_IN_PARALLEL` function returns `.TRUE.` if it is called from the dynamic  
1476 extent of a region executing in parallel, and `.FALSE.` otherwise. A parallel region  
1477 that is serialized is not considered to be a region executing in parallel.

1478 The format of this function is as follows:

1479 LOGICAL FUNCTION OMP\_IN\_PARALLEL

1480 This function has global scope. As a result, it will always return `.TRUE.` within the  
1481 dynamic extent of a region executing in parallel, regardless of nested regions that are  
1482 serialized.

### 1483 3.1.7 OMP\_SET\_DYNAMIC Subroutine

1484 The `OMP_SET_DYNAMIC` subroutine enables or disables dynamic adjustment of the  
1485 number of threads available for execution of parallel regions.

1486 The format of this subroutine is as follows:

1487 SUBROUTINE OMP\_SET\_DYNAMIC(*scalar\_logical\_expression*)

1488 If *scalar\_logical\_expression* evaluates to `.TRUE.`, the number of threads that are used  
 1489 for executing subsequent parallel regions can be adjusted automatically by the  
 1490 run-time environment to obtain the best use of system resources. As a consequence,  
 1491 the number of threads specified by the user is the maximum thread count. The  
 1492 number of threads always remains fixed over the duration of each parallel region and  
 1493 is reported by the `OMP_GET_NUM_THREADS` function. For more information on the  
 1494 `OMP_GET_NUM_THREADS` function, see Section 3.1.2, page 44.

1495 If *scalar\_logical\_expression* evaluates to `.FALSE.`, dynamic adjustment is disabled.  
 1496 (See Section A.11, page 62, for an example.)

1497 A call to `OMP_SET_DYNAMIC` has precedence over the `OMP_DYNAMIC` environment  
 1498 variable. For more information on the `OMP_DYNAMIC` environment variable, see  
 1499 Section 4.3, page 56.

1500 The default for dynamic thread adjustment is implementation dependent. As a result,  
 1501 user codes that depend on a specific number of threads for correct execution should  
 1502 explicitly disable dynamic threads. Implementations are not required to provide the  
 1503 ability to dynamically adjust the number of threads, but they are required to provide  
 1504 the interface in order to support portability across platforms.

### 1505 **3.1.8 `OMP_GET_DYNAMIC` Function**

1506 The `OMP_GET_DYNAMIC` function returns `.TRUE.` if dynamic thread adjustment is  
 1507 enabled and returns `.FALSE.` otherwise. For more information on dynamic thread  
 1508 adjustment, see Section 3.1.7, page 46.

1509 The format of this function is as follows:

1510 

LOGICAL FUNCTION <code>OMP_GET_DYNAMIC</code>
---

1511 If the implementation does not implement dynamic adjustment of the number of  
 1512 threads, this function always returns `.FALSE.`

### 1513 **3.1.9 `OMP_SET_NESTED` Subroutine**

1514 The `OMP_SET_NESTED` subroutine enables or disables nested parallelism.

1515 The format of this subroutine is as follows:

1516 

SUBROUTINE <code>OMP_SET_NESTED</code> ( <i>scalar_logical_expression</i> )
---

1517 If *scalar\_logical\_expression* evaluates to `.FALSE.`, nested parallelism is disabled,  
 1518 which is the default, and nested parallel regions are serialized and executed by the  
 1519 current thread. If set to `.TRUE.`, nested parallelism is enabled, and parallel regions  
 1520 that are nested can deploy additional threads to form the team.

1521 This call has precedence over the `OMP_NESTED` environment variable. For more  
 1522 information on the `OMP_NESTED` environment variable, see Section 4.4, page 56.

1523 When nested parallelism is enabled, the number of threads used to execute nested  
 1524 parallel regions is implementation dependent. As a result, OpenMP-compliant  
 1525 implementations are allowed to serialize nested parallel regions even when nested  
 1526 parallelism is enabled.

### 1527 3.1.10 `OMP_GET_NESTED` Function

1528 The `OMP_GET_NESTED` function returns `.TRUE.` if nested parallelism is enabled and  
 1529 `.FALSE.` if nested parallelism is disabled. For more information on nested  
 1530 parallelism, see Section 3.1.9, page 47.

1531 The format of this function is as follows:

1532 

LOGICAL FUNCTION <code>OMP_GET_NESTED</code>
--

1533 If an implementation does not implement nested parallelism, this function always  
 1534 returns `.FALSE.`

## 1535 3.2 Lock Routines

1536 The OpenMP run-time library includes a set of general-purpose locking routines that  
 1537 take lock variables as arguments. A lock variable must be accessed only through the  
 1538 routines described in this section. For all of these routines, a lock variable should be  
 1539 of type integer and of a `KIND` large enough to hold an address.

1540 Two types of locks are supported: simple locks and nestable locks. Nestable locks may  
 1541 be locked multiple times by the same thread before being unlocked; simple locks may  
 1542 not be locked if they are already in a locked state. Simple lock variables are  
 1543 associated with simple locks and may only be passed to simple lock routines.  
 1544 Nestable lock variables are associated with nestable locks and may only be passed to  
 1545 nestable lock routines.



1546 In the descriptions that follow, *svar* is a simple lock variable and *nvar* is a nestable  
1547 lock variable. Using the defined parameters described at the beginning of this  
1548 chapter (Chapter 3, page 43), these lock variables may be declared as the following:

```
1549 INTEGER (KIND=OMP_LOCK_KIND) :: svar
```

```
1550 INTEGER (KIND=OMP_NEST_LOCK_KIND) :: nvar
```

1551 The simple locking routines are as follows:

- 1552 • The `OMP_INIT_LOCK` subroutine initializes a simple lock (see Section 3.2.1, page  
1553 50).
- 1554 • The `OMP_DESTROY_LOCK` subroutine removes a simple lock (see Section 3.2.2, page  
1555 50).
- 1556 • The `OMP_SET_LOCK` subroutine sets a simple lock when it becomes available (see  
1557 Section 3.2.3, page 50).
- 1558 • The `OMP_UNSET_LOCK` subroutine releases a simple lock (see Section 3.2.4, page  
1559 51).
- 1560 • The `OMP_TEST_LOCK` function tests and possibly sets a simple lock (see Section  
1561 3.2.5, page 51).

1562 The nestable lock routines are as follows:

- 1563 • The `OMP_INIT_NEST_LOCK` subroutine initializes a nestable lock (see Section  
1564 3.2.1, page 50).
- 1565 • The `OMP_DESTROY_NEST_LOCK` subroutine removes a nestable lock (see Section  
1566 3.2.2, page 50).
- 1567 • The `OMP_SET_NEST_LOCK` subroutine sets a nestable lock when it becomes  
1568 available (see Section 3.2.3, page 50).
- 1569 • The `OMP_UNSET_NEST_LOCK` subroutine releases a nestable lock (see Section 3.2.4,  
1570 page 51).
- 1571 • The `OMP_TEST_NEST_LOCK` function tests and possibly sets a nestable lock (see  
1572 Section 3.2.5, page 51).

1573 See Section A.15, page 64, and Section A.16, page 65, for examples of using the  
1574 simple and the nestable lock routines.

### 1575 **3.2.1 OMP\_INIT\_LOCK and OMP\_INIT\_NEST\_LOCK Subroutines**

1576 These subroutines provide the only means of initializing a lock. Each subroutine  
1577 initializes a lock associated with the lock variable argument for use in subsequent  
1578 calls.

1579 The format of these subroutines is as follows:

```
1580 SUBROUTINE OMP_INIT_LOCK(svar)
```

```
1581 SUBROUTINE OMP_INIT_NEST_LOCK(nvar)
```

1582 The initial state is unlocked (that is, no thread owns the lock). For a nestable lock,  
1583 the initial nesting count is zero. *svar* must be an uninitialized simple lock variable.  
1584 *nvar* must be an uninitialized nestable lock variable. It is non-compliant to call this  
1585 routine with a lock variable that is already associated with a lock.

### 1586 **3.2.2 OMP\_DESTROY\_LOCK and OMP\_DESTROY\_NEST\_LOCK Subroutines**

1587 These subroutines insure that the lock variable is uninitialized and cause the lock  
1588 variable to become undefined.

1589 The format for these subroutines is as follows:

```
1590 SUBROUTINE OMP_DESTROY_LOCK(svar)
```

```
1591 SUBROUTINE OMP_DESTROY_NEST_LOCK(nvar)
```

1592 *svar* must be an initialized simple lock variable that is unlocked. *nvar* must be an  
1593 initialized nestable lock variable that is unlocked.

### 1594 **3.2.3 OMP\_SET\_LOCK and OMP\_SET\_NEST\_LOCK Subroutines**

1595 These subroutines force the thread executing the subroutine to wait until the  
1596 specified lock is available and then set the lock. A simple lock is available if it is  
1597 unlocked. A nestable lock is available if it is unlocked or if it is already owned by the  
1598 thread executing the subroutine.

1599 The format of these subroutines is as follows:

1600 SUBROUTINE OMP\_SET\_LOCK (*svar*)

1601 SUBROUTINE OMP\_SET\_NEST\_LOCK (*nvar*)

1602 *svar* must be an initialized simple lock variable. Ownership of the lock is granted to  
1603 the thread executing the subroutine.

1604 *nvar* must be an initialized nestable lock variable. The nesting count is incremented,  
1605 and the thread is granted, or retains, ownership of the lock.

### 1606 3.2.4 OMP\_UNSET\_LOCK and OMP\_UNSET\_NEST\_LOCK Subroutines

1607 These subroutines provide the means of releasing ownership of a lock.

1608 The format of these subroutines is as follows:

1609 SUBROUTINE OMP\_UNSET\_LOCK (*svar*)

1610 SUBROUTINE OMP\_UNSET\_NEST\_LOCK (*nvar*)

1611 The argument to each of these subroutines must be an initialized lock variable owned  
1612 by the thread executing the subroutine. The behavior is unspecified if the thread does  
1613 not own the lock.

1614 The OMP\_UNSET\_LOCK subroutine releases the thread executing the subroutine from  
1615 ownership of the simple lock associated with *svar*.

1616 The OMP\_UNSET\_NEST\_LOCK subroutine decrements the nesting count and releases  
1617 the thread executing the subroutine from ownership of the nestable lock associated  
1618 with *nvar* if the resulting count is zero.

### 1619 3.2.5 OMP\_TEST\_LOCK and OMP\_TEST\_NEST\_LOCK Functions

1620 These functions attempt to set a lock but do not cause the execution of the thread to  
1621 wait.

1622 The format of these functions is as follows:

1623 LOGICAL FUNCTION OMP\_TEST\_LOCK (*svar*)

1624 INTEGER FUNCTION OMP\_TEST\_NEST\_LOCK (*nvar*)

1625 The argument must be an initialized lock variable. These functions attempt to set a  
 1626 lock in the same manner as `OMP_SET_LOCK` and `OMP_SET_NEST_LOCK`, except that  
 1627 they do not cause execution of the thread to wait if the lock is already set.

1628 The `OMP_TEST_LOCK` function returns `.TRUE.` if the simple lock associated with `svar`  
 1629 is successfully set; otherwise it returns `.FALSE.` .

1630 The `OMP_TEST_NEST_LOCK` function returns the new nesting count if the nestable  
 1631 lock associated with `nvar` is successfully set; otherwise, it returns zero.  
 1632 `OMP_TEST_NEST_LOCK` returns a default integer.

### 1633 3.3 Timing Routines

1634 The OpenMP run-time library includes two routines supporting a portable wall-clock  
 1635 timer. The routines are as follows:

- 1636 • The `OMP_GET_WTIME` function, described in Section 3.3.1, page 52.
- 1637 • The `OMP_GET_WTICK` function, described in Section 3.3.2, page 53.

#### 1638 3.3.1 `OMP_GET_WTIME` Function

1639 The `OMP_GET_WTIME` function returns a double precision value equal to the elapsed  
 1640 wallclock time in seconds since some "time in the past". The actual "time in the past"  
 1641 is arbitrary, but it is guaranteed not to change during the execution of the application  
 1642 program.

1643 The format of this function is as follows:

DOUBLE PRECISION FUNCTION <code>OMP_GET_WTIME</code>
--

1644 It is anticipated that the function will be used to measure elapsed times as shown in  
 1645 the following example:

```

1647     double precision start, end
1648     start = OMP_GET_WTIME
1649     .... work to be timed
1650     end = OMP_GET_WTIME
1651     print *, 'Stuff took ', end-start, ' seconds'
  
```

1652           The times returned are "per-thread times" by which is meant they are not required to  
1653           be globally consistent across all the threads participating in an application.

### 1654   **3.3.2 OMP\_GET\_WTICK Function**

1655           The OMP\_GET\_WTICK function returns a double precision value equal to the number  
1656           of seconds between successive clock ticks.

1657           The format of this function is as follows:

1658           

DOUBLE PRECISION FUNCTION OMP\_GET\_WTICK



1660 This chapter describes the OpenMP Fortran API environment variables (or  
1661 equivalent platform-specific mechanisms) that control the execution of parallel code.  
1662 The names of environment variables must be uppercase. The values assigned to them  
1663 are case insensitive.

#### 1664 4.1 OMP\_SCHEDULE Environment Variable

1665 The OMP\_SCHEDULE environment variable applies only to DO and PARALLEL DO  
1666 directives that have the schedule type RUNTIME. For more information on the DO  
1667 directive, see Section 2.3.1, page 13. For more information on the PARALLEL DO  
1668 directive, see Section 2.4.1, page 19.

1669 The schedule type and chunk size for all such loops can be set at run time by setting  
1670 this environment variable to any of the recognized schedule types and to an optional  
1671 chunk size. If a chunk size is specified, it must be a positive scalar integer. For DO  
1672 and PARALLEL DO directives that have a schedule type other than RUNTIME, this  
1673 environment variable is ignored. The default value for this environment variable is  
1674 implementation dependent. If the optional chunk size is not set, a chunk size of 1 is  
1675 assumed, except in the case of a STATIC schedule. For a STATIC schedule, the  
1676 default chunk size is set to the loop iteration count divided by the number of threads  
1677 applied to the loop.

1678 Examples:

```
1679     setenv OMP_SCHEDULE "GUIDED,4"  
1680     setenv OMP_SCHEDULE "dynamic"
```

#### 1681 4.2 OMP\_NUM\_THREADS Environment Variable

1682 The OMP\_NUM\_THREADS environment variable sets the number of threads to use  
1683 during execution, unless that number is explicitly changed by calling the  
1684 OMP\_SET\_NUM\_THREADS subroutine. For more information on the  
1685 OMP\_SET\_NUM\_THREADS subroutine, see Section 3.1.1, page 44.

1686 When dynamic adjustment of the number of threads is enabled, the value of this  
1687 environment variable is the maximum number of threads to use. The value specified  
1688 must be a positive scalar integer. The default value is implementation dependent.

1689       **The behavior of the program is implementation dependent if the requested value of**  
1690       **OMP\_NUM\_THREADS is more than the number of threads an implementation can**  
1691       **support.**

1692       **Example:**

```
1693            setenv OMP_NUM_THREADS 16
```

### 1694 **4.3 OMP\_DYNAMIC Environment Variable**

1695       **The OMP\_DYNAMIC environment variable enables or disables dynamic adjustment of**  
1696       **the number of threads available for execution of parallel regions. For more**  
1697       **information on parallel regions, see Section 2.2, page 9.**

1698       **If set to TRUE, the number of threads that are used for executing parallel regions can**  
1699       **be adjusted by the run-time environment to best utilize system resources.**

1700       **If set to FALSE, dynamic adjustment is disabled. The default condition is**  
1701       **implementation dependent. For more information, see the OMP\_SET\_DYNAMIC**  
1702       **subroutine described in Section 3.1.7, page 46.**

1703       **Example:**

```
1704            setenv OMP_DYNAMIC TRUE
```

### 1705 **4.4 OMP\_NESTED Environment Variable**

1706       **The OMP\_NESTED environment variable enables or disables nested parallelism. If set**  
1707       **to TRUE, nested parallelism is enabled; if it is set to FALSE, it is disabled. The default**  
1708       **value is FALSE. See also Section 3.1.9, page 47.**

1709       **Example:**

```
1710            setenv OMP_NESTED TRUE
```



1711

1712

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

1713

## A.1 Executing a Simple Loop in Parallel

1714

1715

1716

The following example shows how to parallelize a simple loop using the `PARALLEL DO` directive (specified in Section 2.4.1, page 19). The loop iteration variable is private by default, so it is not necessary to declare it explicitly.

1717

1718

1719

1720

1721

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

1722

The `END PARALLEL DO` directive is optional.

1723

## A.2 Specifying Conditional Compilation

1724

1725

1726

The following example illustrates the use of the conditional compilation sentinel (specified in Section 2.1.3, page 8). Assuming Fortran fixed source form, the following statement is illegal when using OpenMP constructs:

1727

1728

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

1729

1730

1731

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:

1732

1733

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

1734

1735

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

### 1736 A.3 Using Parallel Regions

1737 The `PARALLEL` directive (specified in Section 2.2, page 9) can be used in coarse-grain  
1738 parallel programs. In the following example, each thread in the parallel region  
1739 decides what part of the global array `X` to work on based on the thread number:

```
1740 !$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,NPOINTS)
1741     IAM = OMP_GET_THREAD_NUM()
1742     NP = OMP_GET_NUM_THREADS()
1743     IPOINITS = NPOINTS/NP
1744     CALL SUBDOMAIN(X, IAM, IPOINITS)
1745 !$OMP END PARALLEL
```

### 1746 A.4 Using the `NOWAIT` Clause

1747 If there are multiple independent loops within a parallel region, you can use the  
1748 `NOWAIT` clause (specified in Section 2.3.1, page 13) to avoid the implied `BARRIER` at  
1749 the end of the `DO` directive, as follows:

```
1750 !$OMP PARALLEL
1751 !$OMP DO
1752     DO I=2,N
1753         B(I) = (A(I) + A(I-1)) / 2.0
1754     ENDDO
1755 !$OMP END DO NOWAIT
1756 !$OMP DO
1757     DO I=1,M
1758         Y(I) = SQRT(Z(I))
1759     ENDDO
1760 !$OMP END DO NOWAIT
1761 !$OMP END PARALLEL
```

### 1762 A.5 Using the `CRITICAL` Directive

1763 The following example (for Section 2.5.2, page 22) includes several `CRITICAL`  
1764 directives. The example illustrates a queuing model in which a task is dequeued and  
1765 worked on. To guard against multiple threads dequeuing the same task, the  
1766 dequeuing operation must be in a critical section. Because there are two independent

1767 **queues in this example, each queue is protected by CRITICAL directives with**  
1768 **different names, XAXIS and YAXIS, respectively.**

```
1769 !$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,Y)
1770 !$OMP CRITICAL(XAXIS)
1771     CALL DEQUEUE(IX_NEXT, X)
1772 !$OMP END CRITICAL(XAXIS)
1773     CALL WORK(IX_NEXT, X)
1774 !$OMP CRITICAL(YAXIS)
1775     CALL DEQUEUE(IY_NEXT, Y)
1776 !$OMP END CRITICAL(YAXIS)
1777     CALL WORK(IY_NEXT, Y)
1778 !$OMP END PARALLEL
```

## 1779 **A.6 Using the LASTPRIVATE Clause**

1780 **Correct execution sometimes depends on the value that the last iteration of a loop**  
1781 **assigns to a variable. Such programs must list all such variables as arguments to a**  
1782 **LASTPRIVATE clause (specified in Section 2.6.2.5, page 33) so that the values of the**  
1783 **variables are the same as when the loop is executed sequentially.**

```
1784 !$OMP PARALLEL
1785 !$OMP DO LASTPRIVATE(I)
1786     DO I=1,N
1787         A(I) = B(I) + C(I)
1788     ENDDO
1789 !$OMP END PARALLEL
1790     CALL REVERSE(I)
```

1791 **In the preceding example, the value of I at the end of the parallel region will equal**  
1792 **N+1, as in the sequential case.**

## 1793 **A.7 Using the REDUCTION Clause**

1794 **The following example (for Section 2.6.2.6, page 34) shows how to use the REDUCTION**  
1795 **clause:**

```
1796 !$OMP PARALLEL DO DEFAULT(PRIVATE) REDUCTION(+: A,B)
1797     DO I=1,N
```

```

1798         CALL WORK(ALOCAL,BLOCAL)
1799         A = A + ALOCAL
1800         B = B + BLOCAL
1801     ENDDO
1802 !$OMP END PARALLEL DO

```

1803 **The following program is not valid because the reduction is on the**  
1804 ***intrinsic\_procedure\_name* MAX but that name has been redefined to be the variable**  
1805 **named MAX.**

```

1806         MAX = HUGE(0)
1807         M = 0
1808 !$OMP PARALLEL DO REDUCTION(MAX: M) ! MAX is no longer the
1809                                     ! intrinsic so this
1810                                     ! is invalid
1811         DO I = 1, 100
1812             CALL SUB(M,I)
1813         END DO
1814     END

```

```

1815     SUBROUTINE SUB(M,I)
1816         M = MAX(M,I)
1817     END SUBROUTINE SUB

```

1818 **The following valid program performs the reduction using the**  
1819 ***intrinsic\_procedure\_name* MAX even though the intrinsic MAX has been renamed to**  
1820 **REN.**

```

1821     MODULE M
1822         INTRINSIC MAX
1823     END MODULE M
1824     PROGRAM P
1825         USE M, REN => MAX
1826         M = 0
1827 !$OMP PARALLEL DO REDUCTION(REN: M) ! still does MAX
1828         DO I = 1, 100
1829             M = MAX(M,I)
1830         END DO
1831     END PROGRAM P

```

1832 **The following valid program performs the reduction using *intrinsic\_procedure\_name***  
1833 **MAX even though the intrinsic MAX has been renamed to MIN.**

```
1834         MODULE MOD
1835             INTRINSIC MAX, MIN
1836         END MODULE MOD
1837         PROGRAM P
1838             USE MOD, MIN=>MAX, MAX=>MIN
1839             REAL :: R
1840             R = -HUGE(0.0)
1841         !$OMP PARALLEL DO REDUCTION(MIN: R) ! still does MAX
1842             DO I = 1, 1000
1843                 R = MIN(R, SIN(REAL(I)))
1844             END DO
1845             PRINT *, R
1846         END PROGRAM P
```

## 1847 A.8 Specifying Parallel Sections

1848 In the following example (for Section 2.3.2, page 15), subroutines XAXIS, YAXIS, and  
1849 ZAXIS can be executed concurrently. The first SECTION directive is optional. Note  
1850 that all SECTION directives need to appear in the lexical extent of the  
1851 PARALLEL SECTIONS/END PARALLEL SECTIONS construct.

```
1852         !$OMP PARALLEL SECTIONS
1853         !$OMP SECTION
1854             CALL XAXIS()
1855         !$OMP SECTION
1856             CALL YAXIS()
1857         !$OMP SECTION
1858             CALL ZAXIS()
1859         !$OMP END PARALLEL SECTIONS
```

## 1860 A.9 Using SINGLE Directives

1861 The first thread that encounters the SINGLE directive (specified in Section 2.3.3, page  
1862 17) executes subroutines OUTPUT and INPUT. The user must not make any  
1863 assumptions as to which thread will execute the SINGLE section. All other threads  
1864 will skip the SINGLE section and stop at the barrier at the END SINGLE construct. If  
1865 other threads can proceed without waiting for the thread executing the SINGLE  
1866 section, a NOWAIT clause can be specified on the END SINGLE directive.

```

1867      !$OMP PARALLEL DEFAULT(SHARED)
1868          CALL WORK(X)
1869      !$OMP BARRIER
1870      !$OMP SINGLE
1871          CALL OUTPUT(X)
1872          CALL INPUT(Y)
1873      !$OMP END SINGLE
1874          CALL WORK(Y)
1875      !$OMP END PARALLEL

```

## 1876 A.10 Specifying Sequential Ordering

1877 ORDERED sections (specified in Section 2.5.6, page 26) are useful for sequentially  
 1878 ordering the output from work that is done in parallel. Assuming that a reentrant I/O  
 1879 library exists, the following program prints out the indexes in sequential order:

```

1880      !$OMP DO ORDERED SCHEDULE(DYNAMIC)
1881          DO I=LB,UB,ST
1882              CALL WORK(I)
1883          END DO
1884          ...
1885      SUBROUTINE WORK(K)
1886      !$OMP ORDERED
1887          WRITE(*,*) K
1888      !$OMP END ORDERED
1889      END

```

## 1890 A.11 Specifying a Fixed Number of Threads

1891 Some programs rely on a fixed, prespecified number of threads to execute correctly.  
 1892 Because the default setting for the dynamic adjustment of the number of threads is  
 1893 implementation dependent, such programs can choose to turn off the dynamic threads  
 1894 capability and set the number of threads explicitly to ensure portability. The  
 1895 following example (for Section 3.1.1, page 44) shows how to do this:

```

1896          CALL OMP_SET_DYNAMIC(.FALSE.)
1897          CALL OMP_SET_NUM_THREADS(16)
1898      !$OMP PARALLEL DEFAULT(PRIVATE)SHARED(X,NPOINTS)
1899          IAM = OMP_GET_THREAD_NUM()

```

```

1900         IPOINTS = NPOINTS/16
1901         CALL DO_BY_16(X, IAM, IPOINTS)
1902     !$OMP END PARALLEL

```

1903 **In this example, the program executes correctly only if it is executed by 16 threads. If**  
 1904 **the implementation is not capable of supporting 16 threads, the behavior of this**  
 1905 **example is implementation dependent. Note that the number of threads executing a**  
 1906 **parallel region remains constant during a parallel region, regardless of the dynamic**  
 1907 **threads setting. The dynamic threads mechanism determines the number of threads**  
 1908 **to use at the start of the parallel region and keeps it constant for the duration of the**  
 1909 **region.**

## 1910 A.12 Using the ATOMIC Directive

1911 The following example (for Section 2.5.4, page 23) avoids race conditions by protecting  
 1912 all simultaneous updates of the location, by multiple threads, with the ATOMIC  
 1913 directive:

```

1914     !$OMP PARALLEL DO DEFAULT(PRIVATE) SHARED(X, Y, INDEX, N)
1915         DO I=1, N
1916             CALL WORK(XLOCAL, YLOCAL)
1917     !$OMP ATOMIC
1918         X(INDEX(I)) = X(INDEX(I)) + XLOCAL
1919         Y(I) = Y(I) + YLOCAL
1920     ENDDO

```

1921 Note that the ATOMIC directive applies only to the Fortran statement immediately  
 1922 following it. As a result, Y is not updated atomically in this example.

## 1923 A.13 Using the FLUSH Directive

1924 The following example (for Section 2.5.5, page 25) uses the FLUSH directive for  
 1925 point-to-point synchronization between pairs of threads:

```

1926     !$OMP PARALLEL DEFAULT(PRIVATE) SHARED(ISYNC)
1927         IAM = OMP_GET_THREAD_NUM()
1928         ISYNC(IAM) = 0
1929         NEIGH = GET_NEIGHBOR (IAM)
1930     !$OMP BARRIER
1931         CALL WORK()

```

```
1932      C      I AM DONE WITH MY WORK, SYNCHRONIZE WITH MY NEIGHBOR
1933          ISYNC(IAM) = 1
1934      !$OMP FLUSH(ISYNC)
1935      C      WAIT TILL NEIGHBOR IS DONE
1936          DO WHILE (ISYNC(NEIGH) .EQ. 0)
1937      !$OMP FLUSH(ISYNC)
1938          END DO
1939      !$OMP END PARALLEL
```

## 1940 **A.14 Determining the Number of Threads Used**

1941 Consider the following incorrect example:

```
1942          NP = OMP_GET_NUM_THREADS()
1943      !$OMP PARALLEL DO SCHEDULE(STATIC)
1944          DO I = 0, NP-1
1945              CALL WORK(I)
1946          ENDDO
1947      !$OMP END PARALLEL DO
```

1948 The `OMP_GET_NUM_THREADS` call (specified in Section 3.1.2, page 44) returns 1 in the  
1949 serial section of the code, so `NP` will always be equal to 1 in the preceding example. To  
1950 determine the number of threads that will be deployed for the parallel region, the call  
1951 should be inside the parallel region.

1952 The following example shows how to rewrite this program without including a query  
1953 for the number of threads:

```
1954      !$OMP PARALLEL PRIVATE(I)
1955          I = OMP_GET_THREAD_NUM()
1956          CALL WORK(I)
1957      !$OMP END PARALLEL
```

## 1958 **A.15 Using Locks**

1959 This is an example of the use of the simple lock routines (specified in Section 3.2,  
1960 page 48).



1961                   **In the following program, note that the argument to the lock routines should be of**  
 1962                   **type INTEGER and of a KIND large enough to hold an address:**

```

1963                   PROGRAM LOCK_USAGE
1964                   EXTERNAL OMP_TEST_LOCK
1965                   LOGICAL OMP_TEST_LOCK

1966                   INTEGER LCK               ! THIS VARIABLE SHOULD BE POINTER SIZED

1967                   CALL OMP_INIT_LOCK(LCK)
1968   !$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
1969                   ID = OMP_GET_THREAD_NUM( )
1970                   CALL OMP_SET_LOCK(LCK)
1971                   PRINT *, 'MY THREAD ID IS ', ID
1972                   CALL OMP_UNSET_LOCK(LCK)

1973                   DO WHILE ( .NOT. OMP_TEST_LOCK(LCK) )
1974                    CALL SKIP(ID)           ! WE DO NOT YET HAVE THE LOCK
1975                                           ! SO WE MUST DO SOMETHING ELSE
1976                   END DO

1977                   CALL WORK(ID)           ! WE NOW HAVE THE LOCK
1978                                           ! AND CAN DO THE WORK
1979                   CALL OMP_UNSET_LOCK( LCK )
1980   !$OMP END PARALLEL

1981                   CALL OMP_DESTROY_LOCK( LCK )

1982                   END

```

## 1983                   **A.16 Using Nestable Locks**

1984                   **The following example shows how a nestable lock (specified in Section 3.2, page 48)**  
 1985                   **can be used to synchronize updates both to a structure and to one of its components.**

```

1986                   MODULE DATA
1987                    USE OMP_LIB, ONLY OMP_NEXT_LOCK_KIND

1988                    TYPE LOCKED_PAIR
1989                    INTEGER A
1990                    INTEGER B
1991                    INTEGER (OMP_NEST_LOCK_KIND) LCK

```

```
1992         END TYPE
1993     END MODULE DATA

1994     SUBROUTINE INCR_A(P, A)
1995         ! called only from INCR_PAIR, no need to lock
1995         USE DATA
1997         TYPE(LOCKED_PAIR) :: P
1998         INTEGER A

1999         P%A = P%A + A
2000     END SUBROUTINE INCR_A

2001     SUBROUTINE INCR_B(P, B)
2002         ! called from both INCR_PAIR and elsewhere,
2003         ! so we need a nestable lock
2004         USE OMP_LIB
2005         USE DATA
2006         TYPE(LOCKED_PAIR) :: P
2007         INTEGER B

2008         CALL OMP_SET_NEST_LOCK(P%LCK)
2009         P%B = P%B + B
2010         CALL OMP_UNSET_NEST_LOCK(P%LCK)
2011     END SUBROUTINE INCR_B

2012     SUBROUTINE INCR_PAIR(P, A, B)
2013         USE OMP_LIB
2014         USE DATA
2015         TYPE(LOCKED_PAIR) :: P
2016         INTEGER A
2017         INTEGER B

2018         CALL OMP_SET_NEST_LOCK(P%LCK)
2019         CALL INCR_A(P, A)
2020         CALL INCR_B(P, B)
2021         CALL OMP_UNSET_NEST_LOCK(P%LCK)
2022     END SUBROUTINE INCR_PAIR

2023     SUBROUTINE F(P)
2024         USE OMP_LIB
2025         USE DATA
2026         TYPE(LOCKED_PAIR) :: P
2027         INTEGER WORK1, WORK2, WORK3
2028         EXTERNAL WORK1, WORK2, WORK3
```

```

2029      !$OMP PARALLEL SECTIONS
2030      !$OMP SECTION
2031          CALL INCR_PAIR(P, WORK1, WORK2)
2032      !$OMP SECTION
2033          CALL INCR_B(P, WORK3)
2034      !$OMP END PARALLEL SECTIONS
2035      END SUBROUTINE F

```

## 2036 A.17 Nested DO Directives

2037 The following example of directive nesting (specified in Section 2.8, page 41) is legal  
 2038 because the inner and outer DO directives bind to different PARALLEL regions:

```

2039      !$OMP PARALLEL DEFAULT(SHARED)
2040      !$OMP DO
2041          DO I = 1, N
2042      !$OMP PARALLEL SHARED(I,N)
2043      !$OMP DO
2044          DO J = 1, N
2045              CALL WORK(I,J)
2046          END DO
2047      !$OMP END PARALLEL
2048          END DO
2049      !$OMP END PARALLEL

```

2050 The following variation of the preceding example is also legal:

```

2051      !$OMP PARALLEL DEFAULT(SHARED)
2052      !$OMP DO
2053          DO I = 1, N
2054              CALL SOME_WORK(I,N)
2055          END DO
2056      !$OMP END PARALLEL
2057      SUBROUTINE SOME_WORK(I,N)
2058      !$OMP PARALLEL DEFAULT(SHARED)
2059      !$OMP DO
2060          DO J = 1, N
2061              CALL WORK(I,J)
2062          END DO
2063      !$OMP END PARALLEL
2064      RETURN
2065      END

```

## 2066 A.18 Examples Showing Incorrect Nesting of Work-sharing Directives

2067 The examples in this section illustrate the directive nesting rules (specified in Section  
2068 2.8, page 41).

2069 The following example is non-compliant because the inner and outer DO directives are  
2070 nested and bind to the same PARALLEL directive:

### 2071 Example 1: Invalid Example

```
2072 !$OMP PARALLEL DEFAULT(SHARED)
2073 !$OMP DO
2074     DO I = 1, N
2075 !$OMP DO
2076     DO J = 1, N
2077         CALL WORK(I,J)
2078     END DO
2079 END DO
2080 !$OMP END PARALLEL
2081 END
```

2082 The following dynamically nested version of the preceding example is also  
2083 non-compliant:

### 2084 Example 2: Invalid Example

```
2085 !$OMP PARALLEL DEFAULT(SHARED)
2086 !$OMP DO
2087     DO I = 1, N
2088         CALL SOME_WORK(I,N)
2089     END DO
2090 !$OMP END PARALLEL
2091 END
2092 SUBROUTINE SOME_WORK(I,N)
2093 !$OMP DO
2094     DO J = 1, N
2095         CALL WORK(I,J)
2096     END DO
2097     RETURN
2098 END
```

2099 The following example is non-compliant because the DO and SINGLE directives are  
2100 nested, and they bind to the same PARALLEL region:

### 2101 Example 3: Invalid Example

```
2102 !$OMP PARALLEL DEFAULT(SHARED)
2103 !$OMP DO
```

```
2104         DO I = 1, N
2105     !$OMP SINGLE
2106         CALL WORK(I)
2107     !$OMP END SINGLE
2108     END DO
2109 !$OMP END PARALLEL
2110     END
```

2111 **The following example is non-compliant because a BARRIER directive inside a SINGLE**  
2112 **or a DO can result in deadlock:**

#### 2113 **Example 4: Invalid Example**

```
2114     !$OMP PARALLEL DEFAULT(SHARED)
2115     !$OMP DO
2116         DO I = 1, N
2117             CALL WORK(I)
2118     !$OMP BARRIER
2119             CALL MORE_WORK(I)
2120         END DO
2121     !$OMP END PARALLEL
2122     END
```

2123 **The following example is non-compliant because the BARRIER results in deadlock due**  
2124 **to the fact that only one thread at a time can enter the critical section:**

#### 2125 **Example 5: Invalid Example**

```
2126     !$OMP PARALLEL DEFAULT(SHARED)
2127     !$OMP CRITICAL
2128         CALL WORK(N,1)
2129     !$OMP BARRIER
2130         CALL MORE_WORK(N,2)
2131     !$OMP END CRITICAL
2132     !$OMP END PARALLEL
2133     END
```

2134 **The following example is non-compliant because the BARRIER results in deadlock due**  
2135 **to the fact that only one thread executes the SINGLE section:**

#### 2136 **Example 6: Invalid Example**

```
2137     !$OMP PARALLEL DEFAULT(SHARED)
2138         CALL SETUP(N)
2139     !$OMP SINGLE
2140         CALL WORK(N,1)
2141     !$OMP BARRIER
```

```

2142             CALL MORE_WORK(N, 2)
2143     !$OMP END SINGLE
2144             CALL FINISH(N)
2145     !$OMP END PARALLEL
2146             END

```

## 2147 A.19 Binding of BARRIER Directives

2148       The directive binding rules call for a BARRIER directive to bind to the closest  
2149       enclosing PARALLEL directive. (For more information, see Section 2.7, page 40.)

2150       In the following example, the call from MAIN to SUB2 is OpenMP compliant because  
2151       the BARRIER (in SUB3) binds to the PARALLEL region in SUB2. The call from MAIN to  
2152       SUB1 is OpenMP compliant because the BARRIER binds to the PARALLEL region in  
2153       subroutine SUB2.

```

2154             PROGRAM MAIN
2155             CALL SUB1(2)
2156             CALL SUB2(2)
2157             END

2158             SUBROUTINE SUB1(N)
2159     !$OMP PARALLEL PRIVATE(I) SHARED(N)
2160     !$OMP DO
2161             DO I = 1, N
2162             CALL SUB2(I)
2163             END DO
2164     !$OMP END PARALLEL
2165             END

2166             SUBROUTINE SUB2(K)
2167     !$OMP PARALLEL SHARED(K)
2168             CALL SUB3(K)
2169     !$OMP END PARALLEL
2170             END

2171             SUBROUTINE SUB3(N)
2172             CALL WORK(N)
2173     !$OMP BARRIER
2174             CALL WORK(N)
2175             END

```

## 2176 **A.20 Scoping Variables with the PRIVATE Clause**

2177           The values of I and J in the following example are undefined on exit from the  
2178           parallel region:

```
2179           INTEGER I,J
2180           I = 1
2181           J = 2
2182       !$OMP PARALLEL PRIVATE(I) FIRSTPRIVATE(J)
2183           I = 3
2184           J = J+ 2
2185       !$OMP END PARALLEL
2186           PRINT *, I, J
```

2187           (For more information, see Section 2.6.2.1, page 31.)

## 2188 **A.21 Examples of Invalid Storage Association**

2189           The following examples illustrate the implications of the PRIVATE clause rules (see  
2190           Section 2.6.2.1, page 31, rule 4) with regard to storage association:

2191           Example 1: Invalid Example

```
2192           COMMON /BLOCK/ X
2193           X = 1.0
2194       !$OMP PARALLEL PRIVATE (X)
2195           X = 2.0
2196           CALL SUB()
2197           ...
2198       !$OMP END PARALLEL
2199           ...
2200           SUBROUTINE SUB()
2201           COMMON /BLOCK/ X
2202           ...
2203           PRINT *,X                   ! X is undefined. The result of the
2204                                       ! print is undefined.
2205           ...
2206           END SUBROUTINE SUB
2207           END PROGRAM
```

2208 **Example 2: Invalid Example**

```

2209         COMMON /BLOCK/ X
2210         X = 1.0
2211     !$OMP PARALLEL PRIVATE (X)
2212         X = 2.0
2213         CALL SUB()
2214         ...
2215     !$OMP END PARALLEL
2216         ...
2217     CONTAINS
2218         SUBROUTINE SUB()
2219             COMMON /BLOCK/ Y
2220             ...
2221             PRINT *,X                ! X is undefined.
2222             PRINT *,Y                ! Y is undefined.
2223             ...
2224         END SUBROUTINE SUB
2225     END PROGRAM

```

2226 **Example 3: Invalid Example**

```

2227         EQUIVALENCE (X,Y)
2228         X = 1.0
2229     !$OMP PARALLEL PRIVATE(X)
2230         ...
2231         PRINT *,Y                    ! Y is undefined.
2232         Y = 10
2233         PRINT *,X                    ! X is undefined.
2234     !$OMP END PARALLEL

```

2235 **Example 4: Invalid Example**

```

2236         INTEGER A(100), B(100)
2237         EQUIVALENCE (A(51), B(1))

2238     !$OMP PARALLEL DO DEFAULT(PRIVATE) PRIVATE(I,J) LASTPRIVATE(A)
2239         DO I=1,100
2240             DO J=1,100
2241                 B(J) = J - 1
2242             ENDDO

2243             DO J=1,100
2244                 A(J) = J                ! B becomes undefined at this point

```



```

2245         ENDDO
2246         DO J=1,50
2247             B(J) = B(J) + 1 ! Reference to B is not defined. A
2248                               ! becomes undefined at this point.
2249         ENDDO
2250     ENDDO
2251 !$OMP END PARALLEL DO      ! The LASTPRIVATE write for A has
2252                               ! undefined results.

2253         PRINT *, B         ! B is undefined since the LASTPRIVATE
2254                               ! write of A was not defined.
2255     END

```

### 2256 Example 5: Invalid Example

```

2257         COMMON /FOO/ A
2258         DIMENSION B(10)
2259         EQUIVALENCE (A,B(1))
2260         ! the common block has to be at least 10 words
2261         A = 0
2262 !$OMP PARALLEL PRIVATE(/FOO/)
2263         !
2264         ! Without the private clause,
2265         ! we would be passing a member of a sequence
2266         ! that is at least ten elements long. With the private
2267         ! clause, A may no longer be sequence-associated.
2268         !
2269         CALL BAR(A)
2270 !$OMP MASTER
2271         PRINT *, A
2272 !$OMP END MASTER
2273 !$OMP END PARALLEL
2274     END

2275     SUBROUTINE BAR(X)
2276     DIMENSION X(10)
2277     !
2278     ! This use of X does not conform to the specification.
2279     ! It would be legal Fortran 90, but the OpenMP private
2280     ! directive allows the compiler to break the sequence
2281     ! association that A had with the rest of the common block.
2282     !
2283     FORALL (I = 1:10) X(I) = I
2284     END

```

## 2285 A.22 Examples of Syntax of Parallel DO Loops

2286 Both block-do and non-block-do are permitted with PARALLEL DO and work-sharing  
2287 DO directives. However, if a user specifies an ENDDO directive for a non-block-do  
2288 construct with shared termination, then the matching DO directive must precede the  
2289 outermost DO. (For more information, see Section 2.3.1, page 13 and Section 2.4.1,  
2290 page 19.)

2291 The following are some examples:

### 2292 Example 1:

```
2293         DO 100 I = 1,10
2294     !$OMP DO
2295         DO 100 J = 1,10
2296             ...
2297     100 CONTINUE
```

### 2298 Example 2:

```
2299     !$OMP DO
2300         DO 100 J = 1,10
2301             ...
2302     100     A(I) = I + 1
2303     !$OMP ENDDO
```

### 2304 Example 3:

```
2305     !$OMP DO
2306         DO 100 I = 1,10
2307             DO 100 J = 1,10
2308                 ...
2309     100     CONTINUE
2310     !$OMP ENDDO
```

### 2311 Example 4: Invalid Example

```
2312         DO 100 I = 1,10
2313     !$OMP DO
2314         DO 100 J = 1,10
2315             ...
2316     100 CONTINUE
2317     !$OMP ENDDO
```

2318 **A.23 Examples of the ATOMIC Directive**

2319 All atomic references to the storage location of a variable that appears on the  
2320 left-hand side of an ATOMIC assignment statement throughout the program are  
2321 required to have the same type and type parameters. (For more information, see  
2322 Section 2.5.4, page 23.)

2323 The following are some examples:

2324 **Example 1: Invalid Example**

```
2325             INTEGER:: I
2326             REAL:: R
2327             EQUIVALENCE(I,R)
2328     !$OMP PARALLEL
2329             ...
2330     !$OMP ATOMIC
2331             I = I + 1
2332             ...
2333     !$OMP ATOMIC
2334             R = R + 1.0
2335     !$OMP END PARALLEL
```

2336 **Example 2: Invalid Example**

```
2337             SUBROUTINE FRED()
2338             COMMON /BLK/ I
2339             INTEGER I
2340     !$OMP PARALLEL
2341             ...
2342     !$OMP ATOMIC
2343             I = I + 1
2344             ...
2345             CALL SUB()
2346     !$OMP END PARALLEL
2347             END

2348             SUBROUTINE SUB()
2349             COMMON /BLK/ R
2350             REAL R
2351             ...
2352     !$OMP ATOMIC
2353             R = R + 1
2354             END
```

2355 **Example 3: Invalid Example**

2356 **Although the following example might work on some implementation, this is**  
2357 **considered a non-compliant example.**

```
2358         INTEGER :: I
2359         REAL :: R
2360         EQUIVALENCE(I,R)
2361     !OMP PARALLEL
2362         ...
2363     !OMP ATOMIC
2364         I = I + 1
2365     !OMP END PARALLEL
2366         ...
2367     !OMP PARALLEL
2368         ...
2369     !OMP ATOMIC
2370         R = R + 1.0
2371     !OMP END PARALLEL
```

2372 **A.24 Examples of the ORDERED Directive**

2373 **It is possible to have multiple ORDERED sections within a DO specified with the**  
2374 **ORDERED clause. However, the following example is invalid, because the API states**  
2375 **the following:**

2376 **An iteration of a loop with a DO directive must not execute the same**  
2377 **ORDERED directive more than once, and it must not execute more than one**  
2378 **ORDERED directive.**

2379 **For more information, see Section 2.5.6, page 26.**

2380

**Example 1: Invalid Example**

2381

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

```

2382     !$OMP DO
2383         DO I = 1, N
2384             ...
2385     !$OMP ORDERED
2386         ...
2387     !$OMP END ORDERED
2388         ...
2389     !$OMP ORDERED
2390         ...
2391     !$OMP END ORDERED
2392         ...
2393     END DO

```

2394

**Example 2:**

2395

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

```

2396     !$OMP DO ORDERED
2397         DO I = 1,N
2398             ...
2399             IF ( I <= 10) THEN
2400                 ...
2401             !$OMP ORDERED
2402                 WRITE(4,*) I
2403             !$OMP END ORDERED
2404             ENDIF
2405             ...
2406             IF ( I > 10) THEN
2407                 ...
2408             !$OMP ORDERED
2409                 WRITE(3,*) I
2410             !$OMP END ORDERED
2411             ENDIF
2412         ENDDO

```

2413

**A.25 Examples of THREADPRIVATE Data**

2414

2415

2416

The following examples show two invalid uses and two correct uses of the `THREADPRIVATE` directive. For more information, see Section 2.6.1, page 27, item 9 of Section 2.6.3, page 38, and Section 2.6.2.7, page 36.

2417 **Example 1: Invalid Example**

```
2418         MODULE FOO
2419         COMMON /T/ A
2420         END MODULE FOO

2421         SUBROUTINE BAR()
2422         USE FOO
2423         !$OMP THREADPRIVATE(/T/)
2424         !$OMP PARALLEL
2425         ...
2426         !$OMP END PARALLEL
2427         END SUBROUTINE BAR
```

2428 **Example 2: Invalid Example**

```
2429         COMMON /T/ A
2430         !$OMP THREADPRIVATE(/T/)
2431         ...
2432         CONTAINS
2433         SUBROUTINE BAR()
2434         !$OMP PARALLEL COPYIN(/T/)
2435         ...
2436         !$OMP END PARALLEL
2437         END SUBROUTINE BAR
2438         END PROGRAM
```

2439 **Example 3: Correct Rewrite of the Previous Example**

```
2440         COMMON /T/ A
2441         !$OMP THREADPRIVATE(/T/)
2442         ...
2443         CONTAINS
2444         SUBROUTINE BAR()
2445         COMMON /T/ A
2446         !$OMP THREADPRIVATE(/T/)
2447         !$OMP PARALLEL COPYIN(/T/)
2448         ...
2449         !$OMP END PARALLEL
2450         END SUBROUTINE BAR
2451         END PROGRAM
```

```
2452      Example 4: An example of THREADPRIVATE for local variables
2453      PROGRAM P
2454      INTEGER, ALLOCATABLE, SAVE :: A(:)
2455      INTEGER, POINTER, SAVE :: PTR
2456      INTEGER, SAVE :: I
2457      INTEGER, TARGET :: TARG
2458      LOGICAL :: FIRSTIN = .TRUE.
2459      !$OMP THREADPRIVATE(A, B, I, PTR)
2460
2461      ALLOCATE (A(3))
2462      A = (/1,2,3/)
2463      PTR => TARG
2464      I = 5
2465
2466      !$OMP PARALLEL COPYIN(I, PTR)
2467      !$OMP CRITICAL
2468      IF (FIRSTIN) THEN
2469          TARG = 4           ! Update target of ptr
2470          I = I + 10
2471          IF (ALLOCATED(A)) A = A + 10
2472          FIRSTIN = .FALSE.
2473      END IF
2474      IF (ALLOCATED(A)) THEN
2475          PRINT *, 'a = ', A
2476      ELSE
2477          PRINT *, 'A is not allocated'
2478      END IF
2479      PRINT *, 'ptr = ', PTR
2480      PRINT *, 'i = ', I
2481      PRINT *
2482      !$OMP END CRITICAL
2483      !$OMP END PARALLEL
2484      END PROGRAM P
```

2483           **This program, if executed by two threads, will print the following.**

```

2484           a = 11 12 13
2485           ptr = 4
2486           i = 15

2487           A is not allocated
2488           ptr = 4
2489           i = 5

2490           or

2491           A is not allocated
2492           ptr = 4
2493           i = 15

2494           a = 1 2 3
2495           ptr = 4
2496           i = 5

```

## 2497 **A.26 Examples of the Data Attribute Clauses: SHARED and PRIVATE**

2498           **When a named common block is specified in a PRIVATE, FIRSTPRIVATE or**  
2499           **LASTPRIVATE clause of a directive, none of its constituent elements may be declared**  
2500           **in another scope attribute clause in that directive. The following examples, both valid**  
2501           **and invalid, illustrate this point. (For more information, see item 8 of Section 2.6.3,**  
2502           **page 38.)**

2503           **Example 1:**

```

2504           COMMON /C/ X,Y
2505           !$OMP PARALLEL PRIVATE (/C/)
2506           ...
2507           !$OMP END PARALLEL
2508           ...
2509           !$OMP PARALLEL SHARED (X,Y)
2510           ...
2511           !$OMP END PARALLEL

```



2512

**Example 2:**

```
2513             COMMON /C/ X,Y
2514             !$OMP PARALLEL
2515             ...
2516             !$OMP DO PRIVATE(/C/)
2517             ...
2518             !$OMP END DO
2519             !
2520             !$OMP DO PRIVATE(X)
2521             ...
2522             !$OMP END DO
2523             ...
2524             !$OMP END PARALLEL
```

2525

**Example 3: Invalid Example**

```
2526             COMMON /C/ X,Y
2527             !$OMP PARALLEL PRIVATE(/C/), SHARED(X)
2528             ...
2529             !$OMP END PARALLEL
```

2530

**Example 4:**

```
2531             COMMON /C/ X,Y
2532             !$OMP PARALLEL PRIVATE (/C/)
2533             ...
2534             !$OMP END PARALLEL
2535             ...
2536             !$OMP PARALLEL SHARED (/C/)
2537             ...
2538             !$OMP END PARALLEL
```

2539

**Example 5: Invalid Example**

```
2540             COMMON /C/ X,Y
2541             !$OMP PARALLEL PRIVATE(/C/), SHARED(/C/)
2542             ...
2543             !$OMP END PARALLEL
```

2544

**Example 6:**

```
2545             MODULE M
2546             REAL A
2547             CONTAINS
2548             SUBROUTINE SUB
2549             !$OMP PARALLEL PRIVATE(A)
```

```

2550         CALL SUB1()
2551     !$OMP END PARALLEL
2552         END SUBROUTINE SUB
2553     SUBROUTINE SUB1()
2554         A = 5    ! This is A in module M, not the PRIVATE
2555                ! A in SUB
2556     END SUBROUTINE SUB1
2557 END MODULE M

```

## 2558 A.27 Examples of the Data Attribute Clause: COPYPRIVATE

2559 **Example 1.** The COPYPRIVATE clause (specified in Section 2.6.2.8, page 37) can be  
 2560 used to broadcast the value resulting from a read statement directly to all instances  
 2561 of a private variable.

```

2562         SUBROUTINE INIT(A,B)
2563             COMMON /XY/ X,Y
2564     !$OMP THREADPRIVATE (/XY/)
2565     !$OMP SINGLE
2566         READ (11) A,B,X,Y
2567     !$OMP END SINGLE COPYPRIVATE (A,B,/XY/)
2568     END

```

2569 If subroutine `init` is called from a serial region, its behavior is not affected by the  
 2570 presence of the directives. If it is called from a parallel region, then the actual  
 2571 arguments with which `a` and `b` are associated must be private. After the read  
 2572 statement has been executed by one thread, no thread leaves the construct until the  
 2573 private objects designated by `a`, `b`, `x`, and `y` in all threads have become defined with  
 2574 the values read.

2575 **Example 2.** In contrast to the previous example, suppose the read must be performed  
 2576 by a particular thread, say the master thread. In this case, the COPYPRIVATE clause  
 2577 cannot be used to do the broadcast directly, but it can be used to provide access to a  
 2578 temporary shared object.

```

2579         REAL FUNCTION READ_NEXT()
2580         REAL, POINTER :: TMP
2581     !$OMP SINGLE
2582         ALLOCATE (TMP)

```

```

2583         !$OMP END SINGLE COPYPRIVATE (TMP)

2584         !$OMP MASTER
2585             READ (11) TMP
2586         !$OMP END MASTER

2587         !$OMP BARRIER
2588             READ_NEXT = TMP
2589         !$OMP BARRIER

2590         !$OMP SINGLE
2591             DEALLOCATE (TMP)
2592         !$OMP END SINGLE NOWAIT
2593     END FUNCTION READ_NEXT

```

2594 **Example 3.** Suppose that the number of lock objects required within a parallel region  
2595 cannot easily be determined prior to entering it. The copyprivate clause can be used  
2596 to provide access to shared lock objects that are allocated within that parallel region.

```

2597     FUNCTION NEW_LOCK()
2598         INTEGER(OMP_LOCK_KIND), POINTER :: NEW_LOCK
2599         !$OMP SINGLE
2600             ALLOCATE(NEW_LOCK)
2601             CALL OMP_INIT_LOCK(NEW_LOCK)
2602         !$OMP END SINGLE COPYPRIVATE(NEW_LOCK)
2603     END FUNCTION NEW_LOCK

```

2604 **Example 4.** Note that the effect of the copyprivate clause on a variable with the  
2605 allocatable attribute is different than on a variable with the pointer attribute.

```

2606     SUBROUTINE S(N)
2607         REAL, DIMENSION(:), ALLOCATABLE :: A
2608         REAL, DIMENSION(:), POINTER :: B
2609         ALLOCATE (A(N))
2610     !$OMP SINGLE
2611         ALLOCATE (B(N))
2612         READ (11) A,B
2613     !$OMP END SINGLE COPYPRIVATE(A,B)
2614         ! Variable A designates a private object
2615         !   which has the same values in each thread.
2616         ! Variable B designates a shared object.
2617         ...
2618     !$OMP BARRIER
2619     !$OMP SINGLE

```

```

2620          DEALLOCATE (B)
2621      !$OMP END SINGLE NOWAIT
2622          END SUBROUTINE S

```

## 2623 A.28 Examples of WORKSHARE Directive

2624 In the following examples, assume that all 2 letter variable names (e.g., AA, BB) are  
 2625 conformable arrays and single letter names (e.g., I, X) are scalars; implicit typing  
 2626 rules hold. Each of the examples is enclosed in a parallel region. All of the examples  
 2627 are fixed source form so the directives start in column 1.

2628 **Example 1.** WORKSHARE spreads work across some number of threads and there is a  
 2629 barrier after the last statement. Implementations must enforce Fortran execution  
 2630 rules inside of the WORKSHARE block.

```

2631      !$OMP WORKSHARE
2632          AA = BB
2633          CC = DD
2634          EE = FF
2635      !$OMP END WORKSHARE

```

2636 **Example 2.** The final barrier can be eliminated with NOWAIT:

```

2637      !$OMP WORKSHARE
2638          AA = BB
2639          CC = DD
2640      !$OMP END WORKSHARE NOWAIT

2641      !$OMP WORKSHARE
2642          EE = FF
2643      !$OMP END WORKSHARE

```

2644 **Threads doing CC = DD immediately begin work on EE = FF when they are done**  
 2645 **with CC = DD.**

2646 **Example 3.** ATOMIC can be used with WORKSHARE:

```

2647      !$OMP WORKSHARE
2648          AA = BB
2649      !$OMP ATOMIC
2650          I = I + SUM(AA)
2651          CC = DD

```

```
2652         !$OMP END WORKSHARE
```

2653 **The computation of `SUM(AA)` is workshared, but the update to `I` is `ATOMIC`.**

2654 **Example 4. Fortran `WHERE` and `FORALL` statements are *compound statements* of the**  
 2655 **form:**

```
2656         WHERE (EE .ne. 0) FF = 1 / EE
2657         FORALL (I=1:N, XX(I) .ne. 0) YY(I) = 1 / XX(I)
```

2658 **They are made up of a *control* part and a *statement* part. When `WORKSHARE` is applied**  
 2659 **to one of these, both the *control* and the *statement* parts are workshared.**

```
2660         !$OMP WORKSHARE
2661             AA = BB
2662             CC = DD
2663             WHERE (EE .ne. 0) FF = 1 / EE
2664             GG = HH
2665         !$OMP END WORKSHARE
```

2666 **Each task gets worked on in order by the threads:**

```
2667             AA = BB      then
2668             CC = DD      then
2669             EE .ne. 0    then
2670             FF = 1 / EE then
2671             GG = HH
```

2672 **Example 5. An assignment to a shared scalar variable is performed by one thread in**  
 2673 **a `WORKSHARE` while all other threads in the team wait. `SHR` is a shared scalar**  
 2674 **variable in this example.**

```
2675         !$OMP WORKSHARE
2676             AA = BB
2677             SHR = 1
2678             CC = DD
2679         !$OMP END WORKSHARE
```

2680 **Invalid Example 6. An assignment to a private scalar variable is performed by one**  
 2681 **thread while all other threads wait. The private scalar variable is undefined after the**  
 2682 **assignment statement. `PRI` is a private scalar variable in this example.**

```
2683         !$OMP WORKSHARE
2684             AA = BB
2685             PRI = 1
2686             CC = DD
2687         !$OMP END WORKSHARE
```



# Stubs for Run-time Library Routines [B]

---

2688

2689 This section provides stubs for the runtime library routines defined in the OpenMP  
2690 Fortran API. The stubs are provided to enable portability to platforms that do not  
2691 support the OpenMP Fortran API. On such platforms, OpenMP programs must be  
2692 linked with a library containing these stub routines. The stub routines assume that  
2693 the directives in the OpenMP program are ignored. As such, they emulate serial  
2694 semantics.

2695 **Note:** The lock variable that appears in the lock routines must be accessed  
2696 exclusively through these routines. It should not be initialized or otherwise  
2697 modified in the user program. It is declared as a `POINTER` to guarantee that it is  
2698 capable of holding an address. Alternatively, for Fortran 90 implementations, it  
2699 could be declared as an `INTEGER(OMP_LOCK_KIND)` or  
2700 `INTEGER(OMP_NEST_LOCK_KIND)`, as appropriate. In an actual implementation  
2701 the lock variable might be used to hold the address of an allocated object, but  
2702 here it is used to hold an integer value. Users should not make assumptions  
2703 about mechanisms used by OpenMP Fortran implementations to implement  
2704 locks based on the scheme used by the stub routines.

```
2705 SUBROUTINE OMP_SET_NUM_THREADS(NP)
2706   INTEGER NP
2707   RETURN
2708 END
```

```
2709 INTEGER FUNCTION OMP_GET_NUM_THREADS()
2710   OMP_GET_NUM_THREADS = 1
2711   RETURN
2712 END
```

```
2713 INTEGER FUNCTION OMP_GET_MAX_THREADS()
2714   OMP_GET_MAX_THREADS = 1
2715   RETURN
2716 END
```

```
2717 INTEGER FUNCTION OMP_GET_THREAD_NUM()
2718   OMP_GET_THREAD_NUM = 0
2719   RETURN
2720 END
```

```
2721 INTEGER FUNCTION OMP_GET_NUM_PROCS()
2722   OMP_GET_NUM_PROCS = 1
2723   RETURN
2724 END
```

```
2725     LOGICAL FUNCTION OMP_IN_PARALLEL()  
2726     OMP_IN_PARALLEL = .FALSE.  
2727     RETURN  
2728     END  
  
2729     SUBROUTINE OMP_SET_DYNAMIC(FLAG)  
2730     LOGICAL FLAG  
2731     RETURN  
2732     END  
  
2733     LOGICAL FUNCTION OMP_GET_DYNAMIC()  
2734     OMP_GET_DYNAMIC = .FALSE.  
2735     RETURN  
2736     END  
  
2737     SUBROUTINE OMP_SET_NESTED(FLAG)  
2738     LOGICAL FLAG  
2739     RETURN  
2740     END  
  
2741     LOGICAL FUNCTION OMP_GET_NESTED()  
2742     OMP_GET_NESTED = .FALSE.  
2743     RETURN  
2744     END  
  
2745     SUBROUTINE OMP_INIT_LOCK(LOCK)  
2746     ! LOCK is 0 if the simple lock is not initialized  
2747     !       -1 if the simple lock is initialized but not set  
2748     !       1 if the simple lock is set  
2749     POINTER (LOCK,IL)  
2750     INTEGER IL  
2751     LOCK = -1  
2752     RETURN  
2753     END  
  
2754     SUBROUTINE OMP_INIT_NEST_LOCK(NLOCK)  
2755     ! NLOCK is 0 if the nestable lock is not initialized  
2756     !       -1 if the nestable lock is initialized but not set  
2757     !       1 if the nestable lock is set  
2758     ! no use count is maintained  
2759     POINTER (NLOCK,NIL)  
2760     INTEGER NIL  
2761     NLOCK = -1  
2762     RETURN  
2763     END
```



```
2764      SUBROUTINE OMP_DESTROY_LOCK(LOCK)
2765      POINTER (LOCK,IL)
2766      INTEGER IL
2767      LOCK = 0
2768      RETURN
2769      END

2770      SUBROUTINE OMP_DESTROY_NEST_LOCK(NLOCK)
2771      POINTER (NLOCK,NIL)
2772      INTEGER NIL
2773      NLOCK = 0
2774      RETURN
2775      END

2776      SUBROUTINE OMP_SET_LOCK(LOCK)
2777      POINTER (LOCK,IL)
2778      INTEGER IL

2779      IF (LOCK .EQ. 0) THEN
2780          PRINT *, 'ERROR: LOCK NOT INITIALIZED'
2781          STOP
2782      ELSEIF (LOCK .EQ. 1) THEN
2783          PRINT *, 'ERROR: DEADLOCK IN USING LOCK VARIABLE'
2784          STOP
2785      ELSE
2786          LOCK = 1
2787      ENDIF
2788      RETURN
2789      END

2790      SUBROUTINE OMP_SET_NEST_LOCK(NLOCK)
2791      POINTER (NLOCK,NIL)
2792      INTEGER NIL

2793      IF (NLOCK .EQ. 0) THEN
2794          PRINT *, 'ERROR: NESTED LOCK NOT INITIALIZED'
2795          STOP
2796      ELSEIF (NLOCK .EQ. 1) THEN
2797          PRINT *, 'ERROR: DEADLOCK USING NESTED LOCK VARIABLE'
2798          STOP
2799      ELSE
2800          NLOCK = 1
```

```
2801         ENDIF
2802         RETURN
2803         END

2804         SUBROUTINE OMP_UNSET_LOCK(LOCK)
2805         POINTER (LOCK,IL)
2806         INTEGER IL
2807         IF (LOCK .EQ. 0) THEN
2808             PRINT *, 'ERROR: LOCK NOT INITIALIZED'
2809             STOP
2810         ELSEIF (LOCK .EQ. 1) THEN
2811             LOCK = -1
2812         ELSE
2813             PRINT *, 'ERROR: LOCK NOT SET'
2814             STOP
2815         ENDIF
2816         RETURN
2817         END

2818         SUBROUTINE OMP_UNSET_NEST_LOCK(NLOCK)
2819         POINTER (NLOCK,NIL)
2820         INTEGER NIL
2821         IF (NLOCK .EQ. 0) THEN
2822             PRINT *, 'ERROR: NESTED LOCK NOT INITIALIZED'
2823             STOP
2824         ELSEIF (NLOCK .EQ. 1) THEN
2825             NLOCK = -1
2826         ELSE
2827             PRINT *, 'ERROR: NESTED LOCK NOT SET'
2828             STOP
2829         ENDIF
2830         RETURN
2831         END

2832         LOGICAL FUNCTION OMP_TEST_LOCK(LOCK)
2833         POINTER (LOCK,IL)
2834         INTEGER IL
2835         IF (LOCK .EQ. -1) THEN
2836             LOCK = 1
2837             OMP_TEST_LOCK = .TRUE.
```

```
2838         ELSEIF (LOCK .EQ. 1) THEN
2839             OMP_TEST_LOCK = .FALSE.
2840         ELSE
2841             PRINT *, 'ERROR: LOCK NOT INITIALIZED'
2842             STOP
2843         ENDIF
2844         RETURN
2845     END

2846     INTEGER FUNCTION OMP_TEST_NEST_LOCK(NLOCK)
2847     POINTER (NLOCK,NIL)
2848     INTEGER NIL

2849     IF (NLOCK .EQ. -1) THEN
2850         NLOCK = 1
2851         OMP_TEST_NEST_LOCK = 1
2852     ELSEIF (NLOCK .EQ. 1) THEN
2853         OMP_TEST_NEST_LOCK = 0
2854     ELSE
2855         PRINT *, 'ERROR: NESTED LOCK NOT INITIALIZED'
2856         STOP
2857     ENDIF

2858     RETURN
2859     END

2860     DOUBLE PRECISION OMP_WTIME()
2861     OMP_WTIME = 0
2862     RETURN
2863     END

2864     DOUBLE PRECISION OMP_WTICK()
2865     OMP_WTICK = 1.0
2866     RETURN
2867     END
```



2869 A parallel region has at least one barrier, at its end, and may have additional barriers  
 2870 within it. At each barrier, the other members of the team must wait for the last  
 2871 thread to arrive. To minimize this wait time, shared work should be distributed so  
 2872 that all threads arrive at the barrier at about the same time. If some of that shared  
 2873 work is contained in DO constructs, the SCHEDULE clause can be used for this purpose.

2874 When there are repeated references to the same objects, the choice of schedule for a  
 2875 DO construct may be determined primarily by characteristics of the memory system,  
 2876 such as the presence and size of caches and whether memory access times are  
 2877 uniform or nonuniform. Such considerations may make it preferable to have each  
 2878 thread consistently refer to the same set of elements of an array in a series of loops,  
 2879 even if some threads are assigned relatively less work in some of the loops. This can  
 2880 be done by using the STATIC schedule with the same bounds for all the loops. In the  
 2881 following example, note that 1 is used as the lower bound in the second loop, even  
 2882 though K would be more natural if the schedule were not important.

```
2883 !$OMP PARALLEL
2884 !$OMP DO SCHEDULE(STATIC)
2885     DO I=1,N
2886         A(I) = WORK1(I)
2887     ENDDO
2888 !$OMP DO SCHEDULE(STATIC)
2889     DO I=1,N
2890         IF(I .GE. K) A(I) = A(I) + WORK2(I)
2891     ENDDO
2892 !$OMP END PARALLEL
2893     ENDDO
```

2894 In the remaining examples, it is assumed that memory access is not the dominant  
 2895 consideration, and, unless otherwise stated, that all threads receive comparable  
 2896 computational resources. In these cases, the choice of schedule for a DO construct  
 2897 depends on all the shared work that is to be performed between the nearest preceding  
 2898 barrier and either the implied closing barrier or the nearest subsequent barrier, if  
 2899 there is a NOWAIT clause. For each kind of schedule, a short example shows how that  
 2900 schedule kind is likely to be the best choice. A brief discussion follows each example.

2901 The STATIC schedule is also appropriate for the simplest case, a parallel region  
 2902 containing a single DO construct, with each iteration requiring the same amount of  
 2903 work.

```
2904 !$OMP PARALLEL DO SCHEDULE(STATIC)
2905     DO I=1,N
2906         CALL INVARIANT_AMOUNT_OF_WORK(I)
```

2907                   ENDDO

2908                   The **STATIC** schedule is characterized by the properties that each thread gets  
 2909                   approximately the same number of iterations as any other thread, and each thread  
 2910                   can independently determine the iterations assigned to it. Thus no synchronization is  
 2911                   required to distribute the work, and, under the assumption that each iteration  
 2912                   requires the same amount of work, all threads should finish at about the same time.

2913                   For a team of  $P$  threads, let  $\text{CEILING}(N/P)$  be the integer  $Q$ , which satisfies  $N = P \cdot Q$   
 2914                   –  $R$  with  $0 \leq R < P$ . One implementation of the **STATIC** schedule for this example  
 2915                   would assign  $Q$  iterations to the first  $P-1$  threads, and  $Q-R$  iterations to the last  
 2916                   thread. Another acceptable implementation would assign  $Q$  iterations to the first  $P-R$   
 2917                   threads, and  $Q-1$  iterations to the remaining  $R$  threads. This illustrates why a  
 2918                   program should not rely on the details of a particular implementation.

2919                   The **DYNAMIC** schedule is appropriate for the case of a **DO** construct with the  
 2920                   iterations requiring varying, or even unpredictable, amounts of work.

```
2921                   !$OMP PARALLEL DO SCHEDULE(DYNAMIC)
2922                   DO I=1,N
2923                   CALL UNPREDICTABLE_AMOUNT_OF_WORK(I)
2924                   ENDDO
```

2925                   The **DYNAMIC** schedule is characterized by the property that no thread waits at the  
 2926                   barrier for longer than it takes another thread to execute its final iteration. This  
 2927                   requires that iterations be assigned one at a time to threads as they become  
 2928                   available, with synchronization for each assignment. The synchronization overhead  
 2929                   can be reduced by specifying a minimum chunk size  $K$  greater than 1, so that each  
 2930                   thread is assigned  $K$  iterations at a time until fewer than  $K$  iterations remain. This  
 2931                   guarantees that no thread waits at the barrier longer than it takes another thread to  
 2932                   execute its final chunk of (at most)  $K$  iterations.

2933                   The **DYNAMIC** schedule can be useful if the threads receive varying computational  
 2934                   resources, which has much the same effect as varying amounts of work for each  
 2935                   iteration. Similarly, the **DYNAMIC** schedule can also be useful if the threads arrive at  
 2936                   the **DO** construct at varying times, though in some of these cases the **GUIDED** schedule  
 2937                   may be preferable.

2938                   The **GUIDED** schedule is appropriate for the case in which the threads may arrive at  
 2939                   varying times at a **DO** construct with each iteration requiring about the same amount  
 2940                   of work. This can happen if, for example, the **DO** construct is preceded by one or more  
 2941                   **SECTIONS** or **DO** constructs with **NOWAIT** clauses.

```
2942                   !$OMP PARALLEL
2943                   !$OMP SECTIONS
2944                   .....
2945                   !$OMP END SECTIONS NOWAIT
```

```
2946      !$OMP DO SCHEDULE(GUIDED)
2947          DO I=1,N
2948              CALL INVARIANT_AMOUNT_OF_WORK(I)
2949          ENDDO
```

2950 Like DYNAMIC, the GUIDED schedule guarantees that no thread waits at the barrier  
2951 longer than it takes another thread to execute its final iteration, or final  $K$  iterations  
2952 if a chunk size of  $K$  is specified. Among such schedules, the GUIDED schedule is  
2953 characterized by the property that it requires the fewest synchronizations. For chunk  
2954 size  $K$ , a typical implementation will assign  $Q = \text{CEILING}(N/P)$  iterations to the first  
2955 available thread, set  $N$  to the larger of  $N-Q$  and  $P*K$ , and repeat until all iterations  
2956 are assigned.

2957 When the choice of the optimum schedule is not as clear as it is for these examples,  
2958 the RUNTIME schedule is convenient for experimenting with different schedules and  
2959 chunk sizes without having to modify and recompile the program. It can also be  
2960 useful when the optimum schedule depends (in some predictable way) on the input  
2961 data to which the program is applied.

2962 To see an example of the trade-offs between different schedules, consider sharing  
2963 1000 iterations among 8 threads. Suppose there is an invariant amount of work in  
2964 each iteration, and use that as the unit of time.

2965 If all threads start at the same time, the STATIC schedule will cause the construct to  
2966 execute in 125 units, with no synchronization. But suppose that one thread is 100  
2967 units late in arriving. Then the remaining seven threads wait for 100 units at the  
2968 barrier, and the execution time for the whole construct increases to 225.

2969 Because both the DYNAMIC and GUIDED schedules ensure that no thread waits for  
2970 more than one unit at the barrier, the delayed thread causes their execution times for  
2971 the construct to increase only to 138 units, possibly increased by delays from  
2972 synchronization. If such delays are not negligible, it becomes important that the  
2973 number of synchronizations is 1000 for DYNAMIC but only 41 for GUIDED, assuming  
2974 the default chunk size of one. With a chunk size of 25, DYNAMIC and GUIDED both  
2975 finish in 150 units, plus any delays from the required synchronizations, which now  
2976 number only 40 and 20, respectively.





2978 This appendix gives examples of the Fortran `INCLUDE` file and Fortran 90 module  
 2979 that shall be provided by implementations as specified in Chapter 3, page 43.

2980 It has three sections:

- 2981 • Section D.1, page 97, contains an example of a FORTRAN 77 interface declaration  
 2982 `INCLUDE` file
- 2983 • Section D.2, page 99, contains an example of a Fortran 90 interface declaration  
 2984 `MODULE`
- 2985 • Section D.3, page 103, contains an example of a Fortran 90 generic interface for a  
 2986 library routine

## 2987 D.1 Example of an Interface Declaration `INCLUDE` File

```

2988 C the "C" of this comment starts in column 1
2989 integer omp_lock_kind
2990 parameter ( omp_lock_kind = 8 )

2991 integer omp_nest_lock_kind
2992 parameter ( omp_nest_lock_kind = 8 )

2993 C default integer type assumed below
2994 C default logical type assumed below
2995 C OpenMP Fortran API v1.1
2996 integer openmp_version
2997 parameter ( openmp_version = 199910 )

2998 external omp_destroy_lock

2999 external omp_destroy_nest_lock

3000 external omp_get_dynamic
3001 logical omp_get_dynamic

3002 external omp_get_max_threads
3003 integer omp_get_max_threads

3004 external omp_get_nested
  
```

```
3005      logical  omp_get_nested
3006      external omp_get_num_procs
3007      integer  omp_get_num_procs
3008      external omp_get_num_threads
3009      integer  omp_get_num_threads
3010      external omp_get_thread_num
3011      integer  omp_get_thread_num
3012      external omp_get_wtick
3013      double precision  omp_get_wtick
3014      external omp_get_wtime
3015      double precision  omp_get_wtime
3016      external omp_init_lock
3017      external omp_init_nest_lock
3018      external omp_in_parallel
3019      logical  omp_in_parallel
3020      external omp_set_dynamic
3021      external omp_set_lock
3022      external omp_set_nest_lock
3023      external omp_set_nested
3024      external omp_set_num_threads
3025      external omp_test_lock
3026      logical  omp_test_lock
3027      external omp_test_nest_lock
3028      integer  omp_test_nest_lock
3029      external omp_unset_lock
3030      external omp_unset_nest_lock
```

## D.2 Example of a Fortran 90 Interface Declaration MODULE

```
3031
3032     !      the "!" of this comment starts in column 1
3033
3034     module omp_lib_kinds
3035
3036         integer, parameter :: omp_integer_kind      = 4
3037         integer, parameter :: omp_logical_kind     = 4
3038         integer, parameter :: omp_lock_kind       = 8
3039         integer, parameter :: omp_nest_lock_kind  = 8
3040
3041     end module omp_lib_kinds
3042
3043     module omp_lib
3044
3045         use omp_lib_kinds
3046
3047         !
3048         OpenMP Fortran API v1.1
3049         integer, parameter :: openmp_version = 199910
3050
3051         interface
3052             subroutine omp_destroy_lock ( var )
3053             use omp_lib_kinds
3054             integer ( kind=omp_lock_kind ), intent(inout) :: var
3055             end subroutine omp_destroy_lock
3056         end interface
3057
3058         interface
3059             subroutine omp_destroy_nest_lock ( var )
3060             use omp_lib_kinds
3061             integer ( kind=omp_nest_lock_kind ), intent(inout) :: var
3062             end subroutine omp_destroy_nest_lock
3063         end interface
3064
3065         interface
3066             function omp_get_dynamic ( )
3067             use omp_lib_kinds
3068             logical ( kind=omp_logical_kind ) :: omp_get_dynamic
3069             end function omp_get_dynamic
3070         end interface
3071
3072         interface
3073             function omp_get_max_threads ( )
3074             use omp_lib_kinds
```

```
3064         integer ( kind=omp_integer_kind ) :: omp_get_max_threads
3065         end function omp_get_max_threads
3066     end interface

3067     interface
3068         function omp_get_nested ()
3069         use omp_lib_kinds
3070         logical ( kind=omp_logical_kind ) :: omp_get_nested
3071         end function omp_get_nested
3072     end interface

3073     interface
3074         function omp_get_num_procs ()
3075         use omp_lib_kinds
3076         integer ( kind=omp_integer_kind ) :: omp_get_num_procs
3077         end function omp_get_num_procs
3078     end interface

3079     interface
3080         function omp_get_num_threads ()
3081         use omp_lib_kinds
3082         integer ( kind=omp_integer_kind ) :: omp_get_num_threads
3083         end function omp_get_num_threads
3084     end interface

3085     interface
3086         function omp_get_thread_num ()
3087         use omp_lib_kinds
3088         integer ( kind=omp_integer_kind ) :: omp_get_thread_num
3089         end function omp_get_thread_num
3090     end interface

3091     interface
3092         function omp_get_wtick ()
3093         double precision :: omp_get_wtick
3094         end function omp_get_wtick
3095     end interface

3096     interface
3097         function omp_get_wtime ()
3098         double precision :: omp_get_wtime
3099         end function omp_get_wtime
3100     end interface

3101     interface
```

```
3102         subroutine omp_init_lock ( var )
3103         use omp_lib_kinds
3104         integer ( kind=omp_lock_kind ), intent(out) :: var
3105         end subroutine omp_init_lock
3106     end interface

3107     interface
3108         subroutine omp_init_nest_lock ( var )
3109         use omp_lib_kinds
3110         integer ( kind=omp_nest_lock_kind ), intent(out) :: var
3111         end subroutine omp_init_nest_lock
3112     end interface

3113     interface
3114         function omp_in_parallel ()
3115         use omp_lib_kinds
3116         logical ( kind=omp_logical_kind ) :: omp_in_parallel
3117         end function omp_in_parallel
3118     end interface

3119     interface
3120         subroutine omp_set_dynamic ( enable_expr )
3121         use omp_lib_kinds
3122         logical ( kind=omp_logical_kind ), intent(in) :: enable_expr
3123         end subroutine omp_set_dynamic
3124     end interface

3125     interface
3126         subroutine omp_set_lock ( var )
3127         use omp_lib_kinds
3128         integer ( kind=omp_lock_kind ), intent(inout) :: var
3129         end subroutine omp_set_lock
3130     end interface

3131     interface
3132         subroutine omp_set_nest_lock ( var )
3133         use omp_lib_kinds
3134         integer ( kind=omp_nest_lock_kind ), intent(inout) :: var
3135         end subroutine omp_set_nest_lock
3136     end interface

3137     interface
3138         subroutine omp_set_nested ( enable_expr )
3139         use omp_lib_kinds
3140         logical ( kind=omp_logical_kind ), intent(in) :: &
```

```
3141      &                                     enable_expr
3142      end subroutine omp_set_nested
3143 end interface

3144 interface
3145   subroutine omp_set_num_threads ( number_of_threads_expr )
3146   use omp_lib_kinds
3147   integer ( kind=omp_integer_kind ), intent(in) :: &
3148   &                                     number_of_threads_expr
3149   end subroutine omp_set_num_threads
3150 end interface

3151 interface
3152   function omp_test_lock ( var )
3153   use omp_lib_kinds
3154   logical ( kind=omp_logical_kind ) :: omp_test_lock
3155   integer ( kind=omp_lock_kind ), intent(inout) :: var
3156   end function omp_test_lock
3157 end interface

3158 interface
3159   function omp_test_nest_lock ( var )
3160   use omp_lib_kinds
3161   integer ( kind=omp_integer_kind ) :: omp_test_nest_lock
3162   integer ( kind=omp_nest_lock_kind ), intent(inout) :: var
3163   end function omp_test_nest_lock
3164 end interface

3165 interface
3166   subroutine omp_unset_lock ( var )
3167   use omp_lib_kinds
3168   integer ( kind=omp_lock_kind ), intent(inout) :: var
3169   end subroutine omp_unset_lock
3170 end interface

3171 interface
3172   subroutine omp_unset_nest_lock ( var )
3173   use omp_lib_kinds
3174   integer ( kind=omp_nest_lock_kind ), intent(inout) :: var
3175   end subroutine omp_unset_nest_lock
3176 end interface
3177 end module omp_lib
```

### 3178 **D.3 Example of a Generic Interface for a Library Routine**

3179 Any of the OMP runtime library routines that take an argument may be implemented  
3180 with a generic interface so arguments of different KIND type can be accomodated.

3181 **Assume an implementation supports both default INTEGER as KIND =**  
3182 **OMP\_INTEGER\_KIND and another INTEGER KIND, KIND = SHORT\_INT. Then**  
3183 **OMP\_SET\_NUM\_THREADS could be specified in the omp\_lib module as the following:**

```
3184 ! the "!" of this comment starts in column 1
3185 interface omp_set_num_threads
3186   subroutine omp_set_num_threads_1 ( number_of_threads_expr )
3187     use omp_lib_kinds
3188     integer ( kind=omp_integer_kind ), intent(in) :: &
3189 &                                     number_of_threads_expr
3190   end subroutine omp_set_num_threads_1
3191   subroutine omp_set_num_threads_2 ( number_of_threads_expr )
3192     use omp_lib_kinds
3193     integer ( kind=short_int ), intent(in) :: &
3194 &                                     number_of_threads_expr
3195   end subroutine omp_set_num_threads_2
3196 end interface omp_set_num_threads
```





# Implementation Dependent Behaviors in OpenMP Fortran [E]

---

3197

3198

3199

3200

3201

3202

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.

3203

3204

3205

- `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 14).

3206

3207

3208

3209

3210

- 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 14).

3211

3212

- In the absence of the `SCHEDULE` clause, the default schedule is implementation dependent.

3213

3214

- `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 44).

3215

3216

- `OMP_SET_DYNAMIC`: The default for dynamic thread adjustment is implementation dependent (Section 3.1.7, page 46).

3217

3218

3219

- `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 47).

3220

3221

- `OMP_SCHEDULE` environment variable: The default value for this environment variable is implementation dependent (Section 4.1, page 55).

3222

3223

- `OMP_NUM_THREADS` environment variable: The default value is implementation dependent (Section 4.2, page 55).

3224

3225

- `OMP_DYNAMIC` environment variable: The default condition is implementation dependent (Section 4.3, page 56).

3226

3227

- An implementation can replace all `ATOMIC` directives by enclosing the statement in a critical section (Section 2.5.4, page 23).

3228

3229

3230

- 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.

3231  
3232  
3233  
3234  
3235  
3236  
3237  
3238  
3239

- Due to resource constraints, it is not possible for an implementation to document the maximum number of threads that can be created successfully during a program's execution. This number is dependent upon the load on the system, the amount of memory allocated by the program, and the amount of implementation dependent stack space allocated to each thread. If the dynamic threads mechanism is disabled, the behavior of the program is implementation dependent when more threads are requested than can be successfully created. If the dynamic threads mechanism is enabled, requests for more threads than an implementation can support are satisfied by a smaller number of threads.

# New Features in OpenMP Fortran version 2.0 [F]

---

3240

3241

3242

3243

3244

This appendix summarizes the new features in the OpenMP Fortran API version 2.0. Each feature is cross referenced back to the section in the specification where it is described.

3245

3246

3247

- 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 1.3, page 2.

3248

3249

- An OpenMP compliant implementation must document its implementation-defined behaviors. See Appendix E, page 105.

3250

3251

- Directives may contain end-of-line comments starting with an exclamation point. See Section 2.1.2, page 8.

3252

3253

3254

- 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 spec supported by the implementation. See Section 2.1.3, page 8.

3255

3256

- Under the right circumstances, subsequent parallel regions use the same threads with the same thread numbers as previous regions. See Section 2.2, page 9.

3257

- `COPYPRIVATE` is a new modifier on `END SINGLE`. See Section 2.6.2.8, page 37.

3258

3259

- `THREADPRIVATE` may now be applied to variables as well as `COMMON` blocks. See Section 2.6.1, page 27.

3260

3261

3262

- It is implementation-defined whether global variable references in statement functions refer to `SHARED` or `PRIVATE` copies of those variables. See Section 2.6.2, page 30

3263

- `REDUCTION` is now allowed on an array name. See Section 2.6.2.6, page 34.

3264

3265

- `REDUCTION` variables should only be used in reduction computations. See Section 2.6.2.6, page 34

3266

3267

- `COPYIN` now works on variables as well as `COMMON` blocks. See Section 2.6.2.7, page 36.

3268

- Reprivatization of variables is now allowed. See Section 2.6.3, page 38.

3269

- Exceptional values in `REDUCTIONS` may affect the computation.

3270

- Section 3.2, page 48, now defines nested lock routines.

3271

- Section 3.3, page 52, now defines some timing routines.

- 3272 • **Appendix A, page 57, contains more examples.**
- 3273 • **Appendix D, page 97, contains example `INTERFACE` definitions for all of the OMP**  
3274 **run-time routines.**
- 3275 • **The `NUM_THREADS` clause on parallel regions defines the number of threads to be**  
3276 **used to execute that region. See Section 2.2, page 9.**
- 3277 • **New `WORKSHARE`, `BLOCK WORKSHARE`, and `NOWORKSHARE` directives allow**  
3278 **parallelization of array expressions in Fortran statements. See Section 2.3.4, page**  
3279 **17, to .**

3281 *defined* - For the contents of a data object, the property of having or being given a  
3282 valid value. For the allocation status or association status of a data object, the  
3283 property of having or being given a valid status.

3284 *implementation dependent* - A behavior or value that is implementation dependent is  
3285 permitted to vary among different OpenMP compliant implementations (possibly in  
3286 response to limitations of hardware or operating system). Implementation dependent  
3287 items are listed in Appendix E, page 105, and OpenMP compliant implementations  
3288 are required to document how these items are handled.

3289 *non-compliant* - Code structures or arrangements described as non-compliant are not  
3290 required to be supported by OpenMP compliant implementations. Upon encountering  
3291 such structures, an OpenMP compliant implementation may produce a compiler  
3292 error. Even if an implementation produces an executable for a program containing  
3293 such structures, its execution may terminate prematurely or have unpredictable  
3294 behavior.

3295 *undefined* - For the contents of a data object, the property of not having a determinate  
3296 value. The result of a reference to a data object with undefined contents is  
3297 unspecified. For the allocation status or association status of a data object, the  
3298 property of not having a valid status. The behavior of an operation which relies upon  
3299 an undefined allocation status or association status is unspecified.

3300 *unspecified* - A behavior or result that is unspecified is not constrained by  
3301 requirements in the OpenMP Fortran API. Possibly resulting from the misuse of a  
3302 language construct or other error, such a behavior or result may not be knowable  
3303 prior to the execution of a program, and may lead to premature termination of the  
3304 program.