
1
2
3

OpenMP C and C++ Application Program Interface

4

Version 2.0 March 2002

5
6
7
8
9

Copyright © 1997-2002 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.

Contents

1. Introduction	1
1.1 Scope	1
1.2 Definition of Terms	2
1.3 Execution Model	3
1.4 Compliance	4
1.5 Normative References	5
1.6 Organization	5
2. Directives	7
2.1 Directive Format	7
2.2 Conditional Compilation	8
2.3 <code>parallel</code> Construct	8
2.4 Work-sharing Constructs	11
2.4.1 <code>for</code> Construct	11
2.4.2 <code>sections</code> Construct	14
2.4.3 <code>single</code> Construct	15
2.5 Combined Parallel Work-sharing Constructs	16
2.5.1 <code>parallel for</code> Construct	16
2.5.2 <code>parallel sections</code> Construct	17
2.6 Master and Synchronization Directives	17
2.6.1 <code>master</code> Construct	18

1	2.6.2	critical Construct	18
2	2.6.3	barrier Directive	18
3	2.6.4	atomic Construct	19
4	2.6.5	flush Directive	20
5	2.6.6	ordered Construct	22
6	2.7	Data Environment	23
7	2.7.1	threadprivate Directive	23
8	2.7.2	Data-Sharing Attribute Clauses	25
9	2.7.2.1	private	25
10	2.7.2.2	firstprivate	26
11	2.7.2.3	lastprivate	27
12	2.7.2.4	shared	27
13	2.7.2.5	default	28
14	2.7.2.6	reduction	28
15	2.7.2.7	copyin	31
16	2.7.2.8	copyprivate	32
17	2.8	Directive Binding	32
18	2.9	Directive Nesting	33
19	3.	Run-time Library Functions	35
20	3.1	Execution Environment Functions	35
21	3.1.1	omp_set_num_threads Function	36
22	3.1.2	omp_get_num_threads Function	37
23	3.1.3	omp_get_max_threads Function	37
24	3.1.4	omp_get_thread_num Function	38
25	3.1.5	omp_get_num_procs Function	38
26	3.1.6	omp_in_parallel Function	38
27	3.1.7	omp_set_dynamic Function	39
28	3.1.8	omp_get_dynamic Function	40
29	3.1.9	omp_set_nested Function	40

1	3.1.10 <code>omp_get_nested</code> Function	41
2	3.2 Lock Functions	41
3	3.2.1 <code>omp_init_lock</code> and <code>omp_init_nest_lock</code> Functions	42
4	3.2.2 <code>omp_destroy_lock</code> and <code>omp_destroy_nest_lock</code>	
5	Functions	42
6	3.2.3 <code>omp_set_lock</code> and <code>omp_set_nest_lock</code> Functions	42
7	3.2.4 <code>omp_unset_lock</code> and <code>omp_unset_nest_lock</code> Functions	43
8	3.2.5 <code>omp_test_lock</code> and <code>omp_test_nest_lock</code> Functions	43
9	3.3 Timing Routines	44
10	3.3.1 <code>omp_get_wtime</code> Function	44
11	3.3.2 <code>omp_get_wtick</code> Function	45
12	4. Environment Variables	47
13	4.1 <code>OMP_SCHEDULE</code>	48
14	4.2 <code>OMP_NUM_THREADS</code>	48
15	4.3 <code>OMP_DYNAMIC</code>	49
16	4.4 <code>OMP_NESTED</code>	49
17	A. Examples	51
18	A.1 Executing a Simple Loop in Parallel	51
19	A.2 Specifying Conditional Compilation	51
20	A.3 Using Parallel Regions	52
21	A.4 Using the <code>nowait</code> Clause	52
22	A.5 Using the <code>critical</code> Directive	53
23	A.6 Using the <code>lastprivate</code> Clause	53
24	A.7 Using the <code>reduction</code> Clause	54
25	A.8 Specifying Parallel Sections	54
26	A.9 Using <code>single</code> Directives	54
27	A.10 Specifying Sequential Ordering	55
28	A.11 Specifying a Fixed Number of Threads	55
29	A.12 Using the <code>atomic</code> Directive	56

1	A.13 Using the <code>flush</code> Directive with a List	57
2	A.14 Using the <code>flush</code> Directive without a List	57
3	A.15 Determining the Number of Threads Used	59
4	A.16 Using Locks	59
5	A.17 Using Nestable Locks	61
6	A.18 Nested <code>for</code> Directives	62
7	A.19 Examples Showing Incorrect Nesting of Work-sharing Directives	63
8	A.20 Binding of <code>barrier</code> Directives	65
9	A.21 Scoping Variables with the <code>private</code> Clause	67
10	A.22 Using the <code>default(none)</code> Clause	68
11	A.23 Examples of the <code>ordered</code> Directive	68
12	A.24 Example of the <code>private</code> Clause	70
13	A.25 Examples of the <code>copyprivate</code> Data Attribute Clause	71
14	A.26 Using the <code>threadprivate</code> Directive	74
15	A.27 Use of C99 Variable Length Arrays	74
16	A.28 Use of <code>num_threads</code> Clause	75
17	A.29 Use of Work-Sharing Constructs Inside a <code>critical</code> Construct	76
18	A.30 Use of Reprivatization	77
19	A.31 Thread-Safe Lock Functions	77
20	B. Stubs for Run-time Library Functions	79
21	C. OpenMP C and C++ Grammar	85
22	C.1 Notation	85
23	C.2 Rules	86
24	D. Using the <code>schedule</code> Clause	93
25	E. Implementation-Defined Behaviors in OpenMP C/C++	97
26	F. New Features and Clarifications in Version 2.0	99

Introduction

This document specifies a collection of compiler directives, library functions, and environment variables that can be used to specify shared-memory parallelism in C and C++ programs. The functionality described in this document is collectively known as the *OpenMP C/C++ Application Program Interface (API)*. The goal of this specification is to provide a model for parallel programming that allows a program to be portable across shared-memory architectures from different vendors. The OpenMP C/C++ API will be supported by compilers from numerous vendors. More information about OpenMP, including the *OpenMP Fortran Application Program Interface*, can be found at the following web site:

<http://www.openmp.org>

The directives, library functions, and environment variables defined in this document will allow users to create and manage parallel programs while permitting portability. The directives extend the C and C++ sequential programming model with single program multiple data (SPMD) constructs, work-sharing constructs, and synchronization constructs, and they provide support for the sharing and privatization of data. Compilers that support the OpenMP C and C++ API will include a command-line option to the compiler that activates and allows interpretation of all OpenMP compiler directives.

1.1 Scope

This specification covers only user-directed parallelization, wherein the user explicitly specifies the actions to be taken by the compiler and run-time system in order to execute the program in parallel. OpenMP C and C++ implementations are not required to check for dependencies, conflicts, deadlocks, race conditions, or other problems that result in incorrect program execution. The user is responsible for ensuring that the application using the OpenMP C and C++ API constructs executes correctly. Compiler-generated automatic parallelization and directives to the compiler to assist such parallelization are not covered in this document.

1.2 Definition of Terms

The following terms are used in this document:

barrier A synchronization point that must be reached by all threads in a team. Each thread waits until all threads in the team arrive at this point. There are explicit barriers identified by directives and implicit barriers created by the implementation.

construct A construct is a statement. It consists of a directive and the subsequent structured block. Note that some directives are not part of a construct. (See *openmp-directive* in Appendix C).

directive A C or C++ **#pragma** followed by the **omp** identifier, other text, and a new line. The directive specifies program behavior.

dynamic extent All statements in the *lexical extent*, plus any statement inside a function that is executed as a result of the execution of statements within the lexical extent. A dynamic extent is also referred to as a *region*.

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

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

parallel region Statements that bind to an OpenMP parallel construct and may be executed by multiple threads.

private A private variable names a block of storage that is unique to the thread making the reference. Note that there are several ways to specify that a variable is private: a definition within a parallel region, a **threadprivate** directive, a **private**, **firstprivate**, **lastprivate**, or **reduction** clause, or use of the variable as a **for** loop control variable in a **for** loop immediately following a **for** or **parallel for** directive.

region A dynamic extent.

serial region Statements executed only by the *master thread* outside of the dynamic extent of any *parallel region*.

serialize To execute a parallel construct with a team of threads consisting of only a single thread (which is the master thread for that parallel construct), with serial order of execution for the statements within the structured block (the same order as if the block were not part of a parallel construct), and with no effect on the value returned by **omp_in_parallel()** (apart from the effects of any nested parallel constructs).

1	shared	A shared variable names a single block of storage. All threads in a team
2		that access this variable will access this single block of storage.
3	structured block	A structured block is a statement (single or compound) that has a single
4		entry and a single exit. No statement is a structured block if there is a jump
5		into or out of that statement (including a call to long jmp (3C) or the use of
6		throw , but a call to exit is permitted). A compound statement is a
7		structured block if its execution always begins at the opening { and always
8		ends at the closing }. An expression statement, selection statement,
9		iteration statement, or try block is a structured block if the corresponding
10		compound statement obtained by enclosing it in { and } would be a
11		structured block. A jump statement, labeled statement, or declaration
12		statement is not a structured block.
13	team	One or more threads cooperating in the execution of a construct.
14	thread	An execution entity having a serial flow of control, a set of private
15		variables, and access to shared variables.
16	variable	An identifier, optionally qualified by namespace names, that names an
17		object.

1.3 Execution Model

OpenMP uses the fork-join model of parallel execution. Although this fork-join model can be useful for solving a variety of problems, it is somewhat tailored for large array-based applications. OpenMP is intended to support programs that will execute correctly both as parallel programs (multiple threads of execution and a full OpenMP support library) and as sequential programs (directives ignored and a simple OpenMP stubs library). However, it is possible and permitted to develop a program that does not behave correctly when executed sequentially. Furthermore, different degrees of parallelism may result in different numeric results because of changes in the association of numeric operations. For example, a serial addition reduction may have a different pattern of addition associations than a parallel reduction. These different associations may change the results of floating-point addition.

A program written with the OpenMP C/C++ API begins execution as a single thread of execution called the *master thread*. The master thread executes in a serial region until the first parallel construct is encountered. In the OpenMP C/C++ API, the **parallel** directive constitutes a parallel construct. When a parallel construct is encountered, the master thread creates a team of threads, and the master becomes master of the team. Each thread in the team executes the statements in the dynamic extent of a parallel region, except for the work-sharing constructs. Work-sharing constructs must be encountered by all threads in the team in the same order, and the

1 statements within the associated structured block are executed by one or more of the
2 threads. The barrier implied at the end of a work-sharing construct without a
3 **nowait** clause is executed by all threads in the team.

4 If a thread modifies a shared object, it affects not only its own execution
5 environment, but also those of the other threads in the program. The modification is
6 guaranteed to be complete, from the point of view of one of the other threads, at the
7 next sequence point (as defined in the base language) only if the object is declared to
8 be volatile. Otherwise, the modification is guaranteed to be complete after first the
9 modifying thread, and then (or concurrently) the other threads, encounter a **flush**
10 directive that specifies the object (either implicitly or explicitly). Note that when the
11 **flush** directives that are implied by other OpenMP directives are not sufficient to
12 ensure the desired ordering of side effects, it is the programmer's responsibility to
13 supply additional, explicit **flush** directives.

14 Upon completion of the parallel construct, the threads in the team synchronize at an
15 implicit barrier, and only the master thread continues execution. Any number of
16 parallel constructs can be specified in a single program. As a result, a program may
17 fork and join many times during execution.

18 The OpenMP C/C++ API allows programmers to use directives in functions called
19 from within parallel constructs. Directives that do not appear in the lexical extent of
20 a parallel construct but may lie in the dynamic extent are called *orphaned* directives.
21 Orphaned directives give programmers the ability to execute major portions of their
22 program in parallel with only minimal changes to the sequential program. With this
23 functionality, users can code parallel constructs at the top levels of the program call
24 tree and use directives to control execution in any of the called functions.

25 Unsynchronized calls to C and C++ output functions that write to the same file may
26 result in output in which data written by different threads appears in
27 nondeterministic order. Similarly, unsynchronized calls to input functions that read
28 from the same file may read data in nondeterministic order. Unsynchronized use of
29 I/O, such that each thread accesses a different file, produces the same results as
30 serial execution of the I/O functions.

31 1.4 Compliance

32 An implementation of the OpenMP C/C++ API is *OpenMP-compliant* if it recognizes
33 and preserves the semantics of all the elements of this specification, as laid out in
34 Chapters 1, 2, 3, 4, and Appendix C. Appendices A, B, D, E, and F are for information
35 purposes only and are not part of the specification. Implementations that include
36 only a subset of the API are not OpenMP-compliant.

1 The OpenMP C and C++ API is an extension to the base language that is supported
2 by an implementation. If the base language does not support a language construct or
3 extension that appears in this document, the OpenMP implementation is not
4 required to support it.

5 All standard C and C++ library functions and built-in functions (that is, functions of
6 which the compiler has specific knowledge) must be thread-safe. Unsynchronized
7 use of thread-safe functions by different threads inside a parallel region does not
8 produce undefined behavior. However, the behavior might not be the same as in a
9 serial region. (A random number generation function is an example.)

10 The OpenMP C/C++ API specifies that certain behavior is *implementation-defined*. A
11 conforming OpenMP implementation is required to define and document its
12 behavior in these cases. See Appendix E, page 97, for a list of implementation-
13 defined behaviors.

14 1.5 Normative References

- 15 ■ ISO/IEC 9899:1999, *Information Technology - Programming Languages - C*. This
16 OpenMP API specification refers to ISO/IEC 9899:1999 as C99.
- 17 ■ ISO/IEC 9899:1990, *Information Technology - Programming Languages - C*. This
18 OpenMP API specification refers to ISO/IEC 9899:1990 as C90.
- 19 ■ ISO/IEC 14882:1998, *Information Technology - Programming Languages - C++*. This
20 OpenMP API specification refers to ISO/IEC 14882:1998 as C++.

21 Where this OpenMP API specification refers to C, reference is made to the base
22 language supported by the implementation.

23 1.6 Organization

- 24 ■ Directives (see Chapter 2).
- 25 ■ Run-time library functions (see Chapter 3).
- 26 ■ Environment variables (see Chapter 4).
- 27 ■ Examples (see Appendix A).
- 28 ■ Stubs for the run-time library (see Appendix B).
- 29 ■ OpenMP Grammar for C and C++ (see Appendix C).
- 30 ■ Using the **schedule** clause (see Appendix D).
- 31 ■ Implementation-defined behaviors in OpenMP C/C++ (see Appendix E).
- 32 ■ New features in OpenMP C/C++ Version 2.0 (see Appendix F).

Directives

Directives are based on **#pragma** directives defined in the C and C++ standards. Compilers that support the OpenMP C and C++ API will include a command-line option that activates and allows interpretation of all OpenMP compiler directives.

2.1 Directive Format

The syntax of an OpenMP directive is formally specified by the grammar in Appendix C, and informally as follows:

```
#pragma omp directive-name [clause[ [,] clause]...] new-line
```

Each directive starts with **#pragma omp**, to reduce the potential for conflict with other (non-OpenMP or vendor extensions to OpenMP) pragma directives with the same names. The remainder of the directive follows the conventions of the C and C++ standards for compiler directives. In particular, white space can be used before and after the **#**, and sometimes white space must be used to separate the words in a directive. Preprocessing tokens following the **#pragma omp** are subject to macro replacement.

Directives are case-sensitive. The order in which clauses appear in directives is not significant. Clauses on directives may be repeated as needed, subject to the restrictions listed in the description of each clause. If *variable-list* appears in a clause, it must specify only variables. Only one *directive-name* can be specified per directive. For example, the following directive is not allowed:

```
/* ERROR - multiple directive names not allowed */  
#pragma omp parallel barrier
```

1 An OpenMP directive applies to at most one succeeding statement, which must be a
2 structured block.

3 2.2 Conditional Compilation

4 The `_OPENMP` macro name is defined by OpenMP-compliant implementations as the
5 decimal constant `yyymm`, which will be the year and month of the approved
6 specification. This macro must not be the subject of a `#define` or a `#undef`
7 preprocessing directive.

```
8 #ifdef _OPENMP  
9 iam = omp_get_thread_num() + index;  
10 #endif
```

11 If vendors define extensions to OpenMP, they may specify additional predefined
12 macros.

13 2.3 parallel Construct

14 The following directive defines a parallel region, which is a region of the program
15 that is to be executed by multiple threads in parallel. This is the fundamental
16 construct that starts parallel execution.

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

19 The *clause* is one of the following:

```
20 if(scalar-expression)  
21 private(variable-list)  
22 firstprivate(variable-list)  
23 default(shared | none)  
24 shared(variable-list)  
25 copyin(variable-list)  
26 reduction(operator: variable-list)  
27 num_threads(integer-expression)
```

1 When a thread encounters a parallel construct, a team of threads is created if one of
2 the following cases is true:

- 3 ■ No **if** clause is present.
- 4 ■ The **if** expression evaluates to a nonzero value.

5 This thread becomes the master thread of the team, with a thread number of 0, and
6 all threads in the team, including the master thread, execute the region in parallel. If
7 the value of the **if** expression is zero, the region is serialized.

8 To determine the number of threads that are requested, the following rules will be
9 considered in order. The first rule whose condition is met will be applied:

- 10 1. If the **num_threads** clause is present, then the value of the integer expression is
11 the number of threads requested.
- 12 2. If the **omp_set_num_threads** library function has been called, then the value
13 of the argument in the most recently executed call is the number of threads
14 requested.
- 15 3. If the environment variable **OMP_NUM_THREADS** is defined, then the value of this
16 environment variable is the number of threads requested.
- 17 4. If none of the methods above were used, then the number of threads requested is
18 implementation-defined.

19 If the **num_threads** clause is present then it supersedes the number of threads
20 requested by the **omp_set_num_threads** library function or the
21 **OMP_NUM_THREADS** environment variable only for the parallel region it is applied
22 to. Subsequent parallel regions are not affected by it.

23 The number of threads that execute the parallel region also depends upon whether
24 or not dynamic adjustment of the number of threads is enabled. If dynamic
25 adjustment is disabled, then the requested number of threads will execute the
26 parallel region. If dynamic adjustment is enabled then the requested number of
27 threads is the maximum number of threads that may execute the parallel region.

28 If a parallel region is encountered while dynamic adjustment of the number of
29 threads is disabled, and the number of threads requested for the parallel region
30 exceeds the number that the run-time system can supply, the behavior of the
31 program is implementation-defined. An implementation may, for example, interrupt
32 the execution of the program, or it may serialize the parallel region.

33 The **omp_set_dynamic** library function and the **OMP_DYNAMIC** environment
34 variable can be used to enable and disable dynamic adjustment of the number of
35 threads.

1 The number of physical processors actually hosting the threads at any given time is
2 implementation-defined. Once created, the number of threads in the team remains
3 constant for the duration of that parallel region. It can be changed either explicitly
4 by the user or automatically by the run-time system from one parallel region to
5 another.

6 The statements contained within the dynamic extent of the parallel region are
7 executed by each thread, and each thread can execute a path of statements that is
8 different from the other threads. Directives encountered outside the lexical extent of
9 a parallel region are referred to as orphaned directives.

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

12 If a thread in a team executing a parallel region encounters another parallel
13 construct, it creates a new team, and it becomes the master of that new team. Nested
14 parallel regions are serialized by default. As a result, by default, a nested parallel
15 region is executed by a team composed of one thread. The default behavior may be
16 changed by using either the runtime library function `omp_set_nested` or the
17 environment variable `OMP_NESTED`. However, the number of threads in a team that
18 execute a nested parallel region is implementation-defined.

19 Restrictions to the `parallel` directive are as follows:

- 20 ■ At most one `if` clause can appear on the directive.
- 21 ■ It is unspecified whether any side effects inside the `if` expression or
22 `num_threads` expression occur.
- 23 ■ A `throw` executed inside a parallel region must cause execution to resume within
24 the dynamic extent of the same structured block, and it must be caught by the
25 same thread that threw the exception.
- 26 ■ Only a single `num_threads` clause can appear on the directive. The
27 `num_threads` expression is evaluated outside the context of the parallel region,
28 and must evaluate to a positive integer value.
- 29 ■ The order of evaluation of the `if` and `num_threads` clauses is unspecified.

30 Cross References:

- 31 ■ `private`, `firstprivate`, `default`, `shared`, `copyin`, and `reduction`
32 clauses, see Section 2.7.2 on page 25.
- 33 ■ `OMP_NUM_THREADS` environment variable, Section 4.2 on page 48.
- 34 ■ `omp_set_dynamic` library function, see Section 3.1.7 on page 39.
- 35 ■ `OMP_DYNAMIC` environment variable, see Section 4.3 on page 49.
- 36 ■ `omp_set_nested` function, see Section 3.1.9 on page 40.
- 37 ■ `OMP_NESTED` environment variable, see Section 4.4 on page 49.
- 38 ■ `omp_set_num_threads` library function, see Section 3.1.1 on page 36.

2.4 Work-sharing Constructs

A work-sharing construct distributes the execution of the associated statement among the members of the team that encounter it. The work-sharing directives do not launch new threads, and there is no implied barrier on entry to a work-sharing construct.

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

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

- **for** directive
- **sections** directive
- **single** directive

2.4.1 for Construct

The **for** directive identifies an iterative work-sharing construct that specifies that the iterations of the associated loop will be executed in parallel. The iterations of the **for** loop are distributed across threads that already exist in the team executing the parallel construct to which it binds. The syntax of the **for** construct is as follows:

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

The clause is one of the following:

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

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

for (<i>init-expr</i> ; <i>var</i> <i>logical-op</i> <i>b</i> ; <i>incr-expr</i>)	
<i>init-expr</i>	One of the following: <i>var</i> = <i>lb</i> <i>integer-type var</i> = <i>lb</i>
<i>incr-expr</i>	One of the following: ++ <i>var</i> <i>var</i> ++ -- <i>var</i> <i>var</i> -- <i>var</i> += <i>incr</i> <i>var</i> -= <i>incr</i> <i>var</i> = <i>var</i> + <i>incr</i> <i>var</i> = <i>incr</i> + <i>var</i> <i>var</i> = <i>var</i> - <i>incr</i>
<i>var</i>	A signed integer variable. If this variable would otherwise be shared, it is implicitly made private for the duration of the for . This variable must not be modified within the body of the for statement. Unless the variable is specified lastprivate , its value after the loop is indeterminate.
<i>logical-op</i>	One of the following: < <= > >=
<i>lb</i> , <i>b</i> , and <i>incr</i>	Loop invariant integer expressions. There is no synchronization during the evaluation of these expressions. Thus, any evaluated side effects produce indeterminate results.

30 Note that the canonical form allows the number of loop iterations to be computed on
31 entry to the loop. This computation is performed with values in the type of *var*, after
32 integral promotions. In particular, if value of $b - lb + incr$ cannot be represented in
33 that type, the result is indeterminate. Further, if *logical-op* is < or <= then *incr-expr*
34 must cause *var* to increase on each iteration of the loop. If *logical-op* is > or >= then
35 *incr-expr* must cause *var* to decrease on each iteration of the loop.

36 The **schedule** clause specifies how iterations of the **for** loop are divided among
37 threads of the team. The correctness of a program must not depend on which thread
38 executes a particular iteration. The value of *chunk_size*, if specified, must be a loop
39 invariant integer expression with a positive value. There is no synchronization
40 during the evaluation of this expression. Thus, any evaluated side effects produce
41 indeterminate results. The schedule *kind* can be one of the following:

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

static	When schedule(static, chunk_size) is specified, iterations are divided into chunks of a size specified by <i>chunk_size</i> . The chunks are statically assigned to threads in the team in a round-robin fashion in the order of the thread number. When no <i>chunk_size</i> is specified, the iteration space is divided into chunks that are approximately equal in size, with one chunk assigned to each thread.
dynamic	When schedule(dynamic, chunk_size) is specified, the iterations are divided into a series of chunks, each containing <i>chunk_size</i> iterations. Each chunk is assigned to a thread that is waiting for an assignment. The thread executes the chunk of iterations and then waits for its next assignment, until no chunks remain to be assigned. Note that the last chunk to be assigned may have a smaller number of iterations. When no <i>chunk_size</i> is specified, it defaults to 1.
guided	When schedule(guided, chunk_size) is specified, the iterations are assigned to threads in chunks with decreasing sizes. When a thread finishes its assigned chunk of iterations, it is dynamically assigned another chunk, until none remain. For a <i>chunk_size</i> of 1, the size of each chunk is approximately the number of unassigned iterations divided by the number of threads. These sizes decrease approximately exponentially to 1. For a <i>chunk_size</i> with value <i>k</i> greater than 1, the sizes decrease approximately exponentially to <i>k</i> , except that the last chunk may have fewer than <i>k</i> iterations. When no <i>chunk_size</i> is specified, it defaults to 1.
runtime	When schedule(runtime) is specified, the decision regarding scheduling is deferred until runtime. The schedule <i>kind</i> and size of the chunks can be chosen at run time by setting the environment variable OMP_SCHEDULE . If this environment variable is not set, the resulting schedule is implementation-defined. When schedule(runtime) is specified, <i>chunk_size</i> must not be specified.

In the absence of an explicitly defined **schedule** clause, the default **schedule** is implementation-defined.

An OpenMP-compliant program should not rely on a particular schedule for correct execution. A program should not rely on a schedule *kind* conforming precisely to the description given above, because it is possible to have variations in the implementations of the same schedule *kind* across different compilers. The descriptions can be used to select the schedule that is appropriate for a particular situation.

The **ordered** clause must be present when **ordered** directives bind to the **for** construct.

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

1 Restrictions to the **for** directive are as follows:

- 2 ■ The **for** loop must be a structured block, and, in addition, its execution must not
3 be terminated by a **break** statement.
- 4 ■ The values of the loop control expressions of the **for** loop associated with a **for**
5 directive must be the same for all the threads in the team.
- 6 ■ The **for** loop iteration variable must have a signed integer type.
- 7 ■ Only a single **schedule** clause can appear on a **for** directive.
- 8 ■ Only a single **ordered** clause can appear on a **for** directive.
- 9 ■ Only a single **nowait** clause can appear on a **for** directive.
- 10 ■ It is unspecified if or how often any side effects within the *chunk_size*, *lb*, *b*, or *incr*
11 expressions occur.
- 12 ■ The value of the *chunk_size* expression must be the same for all threads in the
13 team.

14 **Cross References:**

- 15 ■ **private**, **firstprivate**, **lastprivate**, and **reduction** clauses, see
16 Section 2.7.2 on page 25.
- 17 ■ **OMP_SCHEDULE** environment variable, see Section 4.1 on page 48.
- 18 ■ **ordered** construct, see Section 2.6.6 on page 22.
- 19 ■ Appendix D, page 93, gives more information on using the **schedule** clause.

20 **2.4.2 sections Construct**

21 The **sections** directive identifies a noniterative work-sharing construct that
22 specifies a set of constructs that are to be divided among threads in a team. Each
23 section is executed once by a thread in the team. The syntax of the **sections**
24 directive is as follows:

```
25 #pragma omp sections [clause[, clause] ...] new-line  
26 {  
27   [#pragma omp section new-line]  
28   structured-block  
29   [#pragma omp section new-line]  
30   structured-block ]  
31   ...  
32 }
```

1 The clause is one of the following:

```
2     private(variable-list)  
3     firstprivate(variable-list)  
4     lastprivate(variable-list)  
5     reduction(operator: variable-list)  
6     nowait
```

7 Each section is preceded by a **section** directive, although the **section** directive is
8 optional for the first section. The **section** directives must appear within the lexical
9 extent of the **sections** directive. There is an implicit barrier at the end of a
10 **sections** construct, unless a **nowait** is specified.

11 Restrictions to the **sections** directive are as follows:

- 12 ■ A **section** directive must not appear outside the lexical extent of the **sections**
13 directive.
- 14 ■ Only a single **nowait** clause can appear on a **sections** directive.

15 **Cross References:**

- 16 ■ **private**, **firstprivate**, **lastprivate**, and **reduction** clauses, see
17 Section 2.7.2 on page 25.

18 **2.4.3 single Construct**

19 The **single** directive identifies a construct that specifies that the associated
20 structured block is executed by only one thread in the team (not necessarily the
21 master thread). The syntax of the **single** directive is as follows:

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

24 The clause is one of the following:

```
25     private(variable-list)  
26     firstprivate(variable-list)  
27     copyprivate(variable-list)  
28     nowait
```

1 There is an implicit barrier after the **single** construct unless a **nowait** clause is
2 specified.

3 Restrictions to the **single** directive are as follows:

- 4 ■ Only a single **nowait** clause can appear on a **single** directive.
- 5 ■ The **copyprivate** clause must not be used with the **nowait** clause.

6 **Cross References:**

- 7 ■ **private**, **firstprivate**, and **copyprivate** clauses, see Section 2.7.2 on
8 page 25.

9 **2.5 Combined Parallel Work-sharing** 10 **Constructs**

11 Combined parallel work-sharing constructs are shortcuts for specifying a parallel
12 region that contains only one work-sharing construct. The semantics of these
13 directives are identical to that of explicitly specifying a **parallel** directive
14 followed by a single work-sharing construct.

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

- 16 ■ the **parallel for** directive.
- 17 ■ the **parallel sections** directive.

18 **2.5.1 parallel for Construct**

19 The **parallel for** directive is a shortcut for a **parallel** region that contains
20 only a single **for** directive. The syntax of the **parallel for** directive is as
21 follows:

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

24 This directive allows all the clauses of the **parallel** directive and the **for**
25 directive, except the **nowait** clause, with identical meanings and restrictions. The
26 semantics are identical to explicitly specifying a **parallel** directive immediately
27 followed by a **for** directive.

Cross References:

- **parallel** directive, see Section 2.3 on page 8.
- **for** directive, see Section 2.4.1 on page 11.
- Data attribute clauses, see Section 2.7.2 on page 25.

2.5.2 parallel sections Construct

The **parallel sections** directive provides a shortcut form for specifying a **parallel** region containing only a single **sections** directive. The semantics are identical to explicitly specifying a **parallel** directive immediately followed by a **sections** directive. The syntax of the **parallel sections** directive is as follows:

```
#pragma omp parallel sections [clause[,] clause] ...] new-line
{
  [#pragma omp section new-line]
  structured-block
  [#pragma omp section new-line]
  structured-block ]
  ...
}
```

The *clause* can be one of the clauses accepted by the **parallel** and **sections** directives, except the **nowait** clause.

Cross References:

- **parallel** directive, see Section 2.3 on page 8.
- **sections** directive, see Section 2.4.2 on page 14.

2.6 Master and Synchronization Directives

The following sections describe :

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

2.6.1 master Construct

The **master** directive identifies a construct that specifies a structured block that is executed by the master thread of the team. The syntax of the **master** directive is as follows:

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

Other threads in the team do not execute the associated structured block. There is no implied barrier either on entry to or exit from the master construct.

2.6.2 critical Construct

The **critical** directive identifies a construct that restricts execution of the associated structured block to a single thread at a time. The syntax of the **critical** directive is as follows:

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

An optional *name* may be used to identify the critical region. Identifiers used to identify a critical region have external linkage and are in a name space which is separate from the name spaces used by labels, tags, members, and ordinary identifiers.

A thread waits at the beginning of a critical region until no other thread is executing a critical region (anywhere in the program) with the same name. All unnamed **critical** directives map to the same unspecified name.

2.6.3 barrier Directive

The **barrier** directive synchronizes all the threads in a team. When encountered, each thread in the team waits until all of the others have reached this point. The syntax of the **barrier** directive is as follows:

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

After all threads in the team have encountered the barrier, each thread in the team begins executing the statements after the barrier directive in parallel.

1 Note that because the **barrier** directive does not have a C language statement as
2 part of its syntax, there are some restrictions on its placement within a program. See
3 Appendix C for the formal grammar. The example below illustrates these
4 restrictions.

```
5 /* ERROR - The barrier directive cannot be the immediate  
6 *           substatement of an if statement  
7 */  
8 if (x!=0)  
9     #pragma omp barrier  
10 ...  
11  
12 /* OK - The barrier directive is enclosed in a  
13 *       compound statement.  
14 */  
15 if (x!=0) {  
16     #pragma omp barrier  
17 }
```

17 2.6.4 atomic Construct

18 The **atomic** directive ensures that a specific memory location is updated atomically,
19 rather than exposing it to the possibility of multiple, simultaneous writing threads.
20 The syntax of the **atomic** directive is as follows:

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

23 The expression statement must have one of the following forms:

```
24     x binop= expr  
25     x++  
26     ++x  
27     x--  
28     --x
```

29 In the preceding expressions:

- 30 ■ *x* is an lvalue expression with scalar type.
- 31 ■ *expr* is an expression with scalar type, and it does not reference the object
32 designated by *x*.

- 1 ■ *binop* is not an overloaded operator and is one of `+`, `*`, `-`, `/`, `&`, `^`, `|`,
2 `<<`, or `>>`.

3 Although it is implementation-defined whether an implementation replaces all
4 **atomic** directives with **critical** directives that have the same unique *name*, the
5 **atomic** directive permits better optimization. Often hardware instructions are
6 available that can perform the atomic update with the least overhead.

7 Only the load and store of the object designated by *x* are atomic; the evaluation of
8 *expr* is not atomic. To avoid race conditions, all updates of the location in parallel
9 should be protected with the **atomic** directive, except those that are known to be
10 free of race conditions.

11 Restrictions to the **atomic** directive are as follows:

- 12 ■ All atomic references to the storage location *x* throughout the program are
13 required to have a compatible type.

14 Examples:

```
15 extern float a[], *p = a, b;  
16 /* Protect against races among multiple updates. */  
17 #pragma omp atomic  
18 a[index[i]] += b;  
19 /* Protect against races with updates through a. */  
20 #pragma omp atomic  
21 p[i] -= 1.0f;  
  
22 extern union {int n; float x;} u;  
23 /* ERROR - References through incompatible types. */  
24 #pragma omp atomic  
25 u.n++;  
26 #pragma omp atomic  
27 u.x -= 1.0f;
```

28 2.6.5 flush Directive

29 The **flush** directive, whether explicit or implied, specifies a “cross-thread”
30 sequence point at which the implementation is required to ensure that all threads in
31 a team have a consistent view of certain objects (specified below) in memory. This
32 means that previous evaluations of expressions that reference those objects are
33 complete and subsequent evaluations have not yet begun. For example, compilers
34 must restore the values of the objects from registers to memory, and hardware may
35 need to flush write buffers to memory and reload the values of the objects from
36 memory.

1 The syntax of the **flush** directive is as follows:

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

3 If the objects that require synchronization can all be designated by variables, then
4 those variables can be specified in the optional *variable-list*. If a pointer is present in
5 the *variable-list*, the pointer itself is flushed, not the object the pointer refers to.

6 A **flush** directive without a *variable-list* synchronizes all shared objects except
7 inaccessible objects with automatic storage duration. (This is likely to have more
8 overhead than a **flush** with a *variable-list*.) A **flush** directive without a *variable-list*
9 is implied for the following directives:

- 10 ■ **barrier**
- 11 ■ At entry to and exit from **critical**
- 12 ■ At entry to and exit from **ordered**
- 13 ■ At entry to and exit from **parallel**
- 14 ■ At exit from **for**
- 15 ■ At exit from **sections**
- 16 ■ At exit from **single**
- 17 ■ At entry to and exit from **parallel for**
- 18 ■ At entry to and exit from **parallel sections**

19 The directive is not implied if a **nowait** clause is present. It should be noted that the
20 **flush** directive is not implied for any of the following:

- 21 ■ At entry to **for**
- 22 ■ At entry to or exit from **master**
- 23 ■ At entry to **sections**
- 24 ■ At entry to **single**

25 A reference that accesses the value of an object with a volatile-qualified type behaves
26 as if there were a **flush** directive specifying that object at the previous sequence
27 point. A reference that modifies the value of an object with a volatile-qualified type
28 behaves as if there were a **flush** directive specifying that object at the subsequent
29 sequence point.

1 Note that because the **flush** directive does not have a C language statement as part
2 of its syntax, there are some restrictions on its placement within a program. See
3 Appendix C for the formal grammar. The example below illustrates these
4 restrictions.

```
5 /* ERROR - The flush directive cannot be the immediate  
6 *      substatement of an if statement.  
7 */  
8 if (x!=0)  
9     #pragma omp flush (x)  
10 ...  
  
11 /* OK - The flush directive is enclosed in a  
12 *      compound statement  
13 */  
14 if (x!=0) {  
15     #pragma omp flush (x)  
16 }
```

17 Restrictions to the **flush** directive are as follows:

- 18 ■ A variable specified in a **flush** directive must not have a reference type.

19 2.6.6 ordered Construct

20 The structured block following an **ordered** directive is executed in the order in
21 which iterations would be executed in a sequential loop. The syntax of the **ordered**
22 directive is as follows:

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

25 An **ordered** directive must be within the dynamic extent of a **for** or **parallel**
26 **for** construct. The **for** or **parallel for** directive to which the **ordered**
27 construct binds must have an **ordered** clause specified as described in Section 2.4.1
28 on page 11. In the execution of a **for** or **parallel for** construct with an
29 **ordered** clause, **ordered** constructs are executed strictly in the order in which
30 they would be executed in a sequential execution of the loop.

31 Restrictions to the **ordered** directive are as follows:

- 32 ■ An iteration of a loop with a **for** construct must not execute the same **ordered**
33 directive more than once, and it must not execute more than one **ordered**
34 directive.

2.7 Data Environment

This section presents a directive and several clauses for controlling the data environment during the execution of parallel regions, as follows:

- A **threadprivate** directive (see the following section) is provided to make file-scope, namespace-scope, or static block-scope variables local to a thread.
- Clauses that may be specified on the directives to control the sharing attributes of variables for the duration of the parallel or work-sharing constructs are described in Section 2.7.2 on page 25.

2.7.1 threadprivate Directive

The **threadprivate** directive makes the named file-scope, namespace-scope, or static block-scope variables specified in the *variable-list* private to a thread. *variable-list* is a comma-separated list of variables that do not have an incomplete type. The syntax of the **threadprivate** directive is as follows:

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

Each copy of a **threadprivate** variable is initialized once, at an unspecified point in the program prior to the first reference to that copy, and in the usual manner (i.e., as the master copy would be initialized in a serial execution of the program). Note that if an object is referenced in an explicit initializer of a **threadprivate** variable, and the value of the object is modified prior to the first reference to a copy of the variable, then the behavior is unspecified.

As with any private variable, a thread must not reference another thread's copy of a **threadprivate** object. During serial regions and master regions of the program, references will be to the master thread's copy of the object.

After the first parallel region executes, the data in the **threadprivate** objects is guaranteed to persist only if the dynamic threads mechanism has been disabled and if the number of threads remains unchanged for all parallel regions.

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

- A **threadprivate** directive for file-scope or namespace-scope variables must appear outside any definition or declaration, and must lexically precede all references to any of the variables in its list.
- Each variable in the *variable-list* of a **threadprivate** directive at file or namespace scope must refer to a variable declaration at file or namespace scope that lexically precedes the directive.

- 1 ■ A **threadprivate** directive for static block-scope variables must appear in the
2 scope of the variable and not in a nested scope. The directive must lexically
3 precede all references to any of the variables in its list.
- 4 ■ Each variable in the *variable-list* of a **threadprivate** directive in block scope
5 must refer to a variable declaration in the same scope that lexically precedes the
6 directive. The variable declaration must use the static storage-class specifier.
- 7 ■ If a variable is specified in a **threadprivate** directive in one translation unit, it
8 must be specified in a **threadprivate** directive in every translation unit in
9 which it is declared.
- 10 ■ A **threadprivate** variable must not appear in any clause except the **copyin**,
11 **copyprivate**, **schedule**, **num_threads**, or the **if** clause.
- 12 ■ The address of a **threadprivate** variable is not an address constant.
- 13 ■ A **threadprivate** variable must not have an incomplete type or a reference
14 type.
- 15 ■ A **threadprivate** variable with non-POD class type must have an accessible,
16 unambiguous copy constructor if it is declared with an explicit initializer.

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

```

20 int x = 1;
21 T a(x);
22 const T b_aux(x); /* Capture value of x = 1 */
23 T b(b_aux);
24 #pragma omp threadprivate(a, b)

25 void f(int n) {
26     x++;
27     #pragma omp parallel for
28     /* In each thread:
29      * Object a is constructed from x (with value 1 or ??)
30      * Object b is copy-constructed from b_aux
31      */
32     for (int i=0; i<n; i++) {
33         g(a, b); /* Value of a is unspecified. */
34     }
35 }

```

36 Cross References:

- 37 ■ Dynamic threads, see Section 3.1.7 on page 39.
- 38 ■ **OMP_DYNAMIC** environment variable, see Section 4.3 on page 49.

2.7.2 Data-Sharing Attribute Clauses

Several directives accept clauses that allow a user to control the sharing attributes of variables for the duration of the region. Sharing attribute clauses apply only to variables in the lexical extent of the directive on which the clause appears. Not all of the following clauses are allowed on all directives. The list of clauses that are valid on a particular directive are described with the directive.

If a variable is visible when a parallel or work-sharing construct is encountered, and the variable is not specified in a sharing attribute clause or **threadprivate** directive, then the variable is shared. Static variables declared within the dynamic extent of a parallel region are shared. Heap allocated memory (for example, using **malloc()** in C or C++ or the **new** operator in C++) is shared. (The pointer to this memory, however, can be either private or shared.) Variables with automatic storage duration declared within the dynamic extent of a parallel region are private.

Most of the clauses accept a *variable-list* argument, which is a comma-separated list of variables that are visible. If a variable referenced in a data-sharing attribute clause has a type derived from a template, and there are no other references to that variable in the program, the behavior is undefined.

All variables that appear within directive clauses must be visible. Clauses may be repeated as needed, but no variable may be specified in more than one clause, except that a variable can be specified in both a **firstprivate** and a **lastprivate** clause.

The following sections describe the data-sharing attribute clauses:

- **private**, Section 2.7.2.1 on page 25.
- **firstprivate**, Section 2.7.2.2 on page 26.
- **lastprivate**, Section 2.7.2.3 on page 27.
- **shared**, Section 2.7.2.4 on page 27.
- **default**, Section 2.7.2.5 on page 28.
- **reduction**, Section 2.7.2.6 on page 28.
- **copyin**, Section 2.7.2.7 on page 31.
- **copyprivate**, Section 2.7.2.8 on page 32.

2.7.2.1 private

The **private** clause declares the variables in *variable-list* to be private to each thread in a team. The syntax of the **private** clause is as follows:

```
private(variable-list)
```

1 The behavior of a variable specified in a **private** clause is as follows. A new object
2 with automatic storage duration is allocated for the construct. The size and
3 alignment of the new object are determined by the type of the variable. This
4 allocation occurs once for each thread in the team, and a default constructor is
5 invoked for a class object if necessary; otherwise the initial value is indeterminate.
6 The original object referenced by the variable has an indeterminate value upon entry
7 to the construct, must not be modified within the dynamic extent of the construct,
8 and has an indeterminate value upon exit from the construct.

9 In the lexical extent of the directive construct, the variable references the new private
10 object allocated by the thread.

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

- 12 ■ A variable with a class type that is specified in a **private** clause must have an
13 accessible, unambiguous default constructor.
- 14 ■ A variable specified in a **private** clause must not have a **const**-qualified type
15 unless it has a class type with a **mutable** member.
- 16 ■ A variable specified in a **private** clause must not have an incomplete type or a
17 reference type.
- 18 ■ Variables that appear in the **reduction** clause of a **parallel** directive cannot
19 be specified in a **private** clause on a work-sharing directive that binds to the
20 parallel construct.

21 2.7.2.2 **firstprivate**

22 The **firstprivate** clause provides a superset of the functionality provided by the
23 **private** clause. The syntax of the **firstprivate** clause is as follows:

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

25 Variables specified in *variable-list* have **private** clause semantics, as described in
26 Section 2.7.2.1 on page 25. The initialization or construction happens as if it were
27 done once per thread, prior to the thread's execution of the construct. For a
28 **firstprivate** clause on a parallel construct, the initial value of the new private
29 object is the value of the original object that exists immediately prior to the parallel
30 construct for the thread that encounters it. For a **firstprivate** clause on a work-
31 sharing construct, the initial value of the new private object for each thread that
32 executes the work-sharing construct is the value of the original object that exists
33 prior to the point in time that the same thread encounters the work-sharing
34 construct. In addition, for C++ objects, the new private object for each thread is copy
35 constructed from the original object.

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

- 37 ■ A variable specified in a **firstprivate** clause must not have an incomplete
38 type or a reference type.

- 1 ■ A variable with a class type that is specified as **firstprivate** must have an
2 accessible, unambiguous copy constructor.
- 3 ■ Variables that are private within a parallel region or that appear in the
4 **reduction** clause of a **parallel** directive cannot be specified in a
5 **firstprivate** clause on a work-sharing directive that binds to the parallel
6 construct.

7 2.7.2.3 lastprivate

8 The **lastprivate** clause provides a superset of the functionality provided by the
9 **private** clause. The syntax of the **lastprivate** clause is as follows:

```
10 lastprivate(variable-list)
```

11 Variables specified in the *variable-list* have **private** clause semantics. When a
12 **lastprivate** clause appears on the directive that identifies a work-sharing
13 construct, the value of each **lastprivate** variable from the sequentially last
14 iteration of the associated loop, or the lexically last section directive, is assigned to
15 the variable's original object. Variables that are not assigned a value by the last
16 iteration of the **for** or **parallel for**, or by the lexically last section of the
17 **sections** or **parallel sections** directive, have indeterminate values after the
18 construct. Unassigned subobjects also have an indeterminate value after the
19 construct.

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

- 21 ■ All restrictions for **private** apply.
- 22 ■ A variable with a class type that is specified as **lastprivate** must have an
23 accessible, unambiguous copy assignment operator.
- 24 ■ Variables that are private within a parallel region or that appear in the
25 **reduction** clause of a **parallel** directive cannot be specified in a
26 **lastprivate** clause on a work-sharing directive that binds to the parallel
27 construct.

28 2.7.2.4 shared

29 This clause shares variables that appear in the *variable-list* among all the threads in a
30 team. All threads within a team access the same storage area for **shared** variables.
31 The syntax of the **shared** clause is as follows:

```
32 shared(variable-list)
```

2.7.2.5 default

The **default** clause allows the user to affect the data-sharing attributes of variables. The syntax of the **default** clause is as follows:

```
default(shared | none)
```

Specifying **default(shared)** is equivalent to explicitly listing each currently visible variable in a **shared** clause, unless it is **threadprivate** or **const-qualified**. In the absence of an explicit **default** clause, the default behavior is the same as if **default(shared)** were specified.

Specifying **default(none)** requires that at least one of the following must be true for every reference to a variable in the lexical extent of the parallel construct:

- The variable is explicitly listed in a data-sharing attribute clause of a construct that contains the reference.
- The variable is declared within the parallel construct.
- The variable is **threadprivate**.
- The variable has a **const**-qualified type.
- The variable is the loop control variable for a **for** loop that immediately follows a **for** or **parallel for** directive, and the variable reference appears inside the loop.

Specifying a variable on a **firstprivate**, **lastprivate**, or **reduction** clause of an enclosed directive causes an implicit reference to the variable in the enclosing context. Such implicit references are also subject to the requirements listed above.

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

A variable's default data-sharing attribute can be overridden by using the **private**, **firstprivate**, **lastprivate**, **reduction**, and **shared** clauses, as demonstrated by the following example:

```
#pragma omp parallel for default(shared) firstprivate(i)\
private(x) private(r) lastprivate(i)
```

2.7.2.6 reduction

This clause performs a reduction on the scalar variables that appear in *variable-list*, with the operator *op*. The syntax of the **reduction** clause is as follows:

```
reduction(op:variable-list)
```

1 A reduction is typically specified for a statement with one of the following forms:

```
2 x = x op expr
3 x binop= expr
4 x = expr op x      (except for subtraction)
5 x++
6 ++x
7 x--
8 --x
```

9 where:

10 <i>x</i>	One of the reduction variables specified in
11	the <i>list</i> .
12 <i>variable-list</i>	A comma-separated list of scalar reduction
13	variables.
14 <i>expr</i>	An expression with scalar type that does
15	not reference <i>x</i> .
16 <i>op</i>	Not an overloaded operator but one of + ,
17	* , - , & , ^ , , && , or .
18 <i>binop</i>	Not an overloaded operator but one of + ,
19	* , - , & , ^ , or .

20 The following is an example of the **reduction** clause:

```
21 #pragma omp parallel for reduction(+: a, y) reduction(||: am)
22 for (i=0; i<n; i++) {
23     a += b[i];
24     y = sum(y, c[i]);
25     am = am || b[i] == c[i];
26 }
```

27 As shown in the example, an operator may be hidden inside a function call. The user
28 should be careful that the operator specified in the **reduction** clause matches the
29 reduction operation.

30 Although the right operand of the **||** operator has no side effects in this example,
31 they are permitted, but should be used with care. In this context, a side effect that is
32 guaranteed not to occur during sequential execution of the loop may occur during
33 parallel execution. This difference can occur because the order of execution of the
34 iterations is indeterminate.

1 The operator is used to determine the initial value of any private variables used by
2 the compiler for the reduction and to determine the finalization operator. Specifying
3 the operator explicitly allows the reduction statement to be outside the lexical extent
4 of the construct. Any number of **reduction** clauses may be specified on the
5 directive, but a variable may appear in at most one **reduction** clause for that
6 directive.

7 A private copy of each variable in *variable-list* is created, one for each thread, as if the
8 **private** clause had been used. The private copy is initialized according to the
9 operator (see the following table).

10 At the end of the region for which the **reduction** clause was specified, the original
11 object is updated to reflect the result of combining its original value with the final
12 value of each of the private copies using the operator specified. The reduction
13 operators are all associative (except for subtraction), and the compiler may freely
14 reassociate the computation of the final value. (The partial results of a subtraction
15 reduction are added to form the final value.)

16 The value of the original object becomes indeterminate when the first thread reaches
17 the containing clause and remains so until the reduction computation is complete.
18 Normally, the computation will be complete at the end of the construct; however, if
19 the **reduction** clause is used on a construct to which **nowait** is also applied, the
20 value of the original object remains indeterminate until a barrier synchronization has
21 been performed to ensure that all threads have completed the **reduction** clause.

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

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

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

- 35 ■ The type of the variables in the **reduction** clause must be valid for the
36 reduction operator except that pointer types and reference types are never
37 permitted.

- A variable that is specified in the **reduction** clause must not be **const**-qualified.
- Variables that are private within a parallel region or that appear in the **reduction** clause of a **parallel** directive cannot be specified in a **reduction** clause on a work-sharing directive that binds to the parallel construct.

```
#pragma omp parallel private(y)
{ /* ERROR - private variable y cannot be specified
   in a reduction clause */
  #pragma omp for reduction(+: y)
  for (i=0; i<n; i++)
    y += b[i];
}

/* ERROR - variable x cannot be specified in both
   a shared and a reduction clause */
#pragma omp parallel for shared(x) reduction(+: x)
```

2.7.2.7 **copyin**

The **copyin** clause provides a mechanism to assign the same value to **threadprivate** variables for each thread in the team executing the parallel region. For each variable specified in a **copyin** clause, the value of the variable in the master thread of the team is copied, as if by assignment, to the thread-private copies at the beginning of the parallel region. The syntax of the **copyin** clause is as follows:

```
copyin(variable-list)
```

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

- A variable that is specified in the **copyin** clause must have an accessible, unambiguous copy assignment operator.
- A variable that is specified in the **copyin** clause must be a **threadprivate** variable.

2.7.2.8 `copyprivate`

The `copyprivate` clause provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members. It is an alternative to using a shared variable for the value when providing such a shared variable would be difficult (for example, in a recursion requiring a different variable at each level). The `copyprivate` clause can only appear on the `single` directive.

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

```
copyprivate(variable-list)
```

The effect of the `copyprivate` clause on the variables in its *variable-list* occurs after the execution of the structured block associated with the `single` construct, and before any of the threads in the team have left the barrier at the end of the construct. Then, in all other threads in the team, for each variable in the *variable-list*, that variable becomes defined (as if by assignment) with the value of the corresponding variable in the thread that executed the construct's structured block.

Restrictions to the `copyprivate` clause are as follows:

- A variable that is specified in the `copyprivate` clause must not appear in a `private` or `firstprivate` clause for the same `single` directive.
- If a `single` directive with a `copyprivate` clause is encountered in the dynamic extent of a parallel region, all variables specified in the `copyprivate` clause must be private in the enclosing context.
- A variable that is specified in the `copyprivate` clause must have an accessible unambiguous copy assignment operator.

2.8 Directive Binding

Dynamic binding of directives must adhere to the following rules:

- The `for`, `sections`, `single`, `master`, and `barrier` directives bind to the dynamically enclosing `parallel`, if one exists, regardless of the value of any `if` clause that may be present on that directive. If no parallel region is currently being executed, the directives are executed by a team composed of only the master thread.
- The `ordered` directive binds to the dynamically enclosing `for`.
- The `atomic` directive enforces exclusive access with respect to `atomic` directives in all threads, not just the current team.
- The `critical` directive enforces exclusive access with respect to `critical` directives in all threads, not just the current team.

- A directive can never bind to any directive outside the closest dynamically enclosing **parallel**.

2.9 Directive Nesting

Dynamic nesting of directives must adhere to the following rules:

- A **parallel** directive dynamically inside another **parallel** logically establishes a new team, which is composed of only the current thread, unless nested parallelism is enabled.
- **for**, **sections**, and **single** directives that bind to the same **parallel** are not allowed to be nested inside each other.
- **critical** directives with the same *name* are not allowed to be nested inside each other. Note this restriction is not sufficient to prevent deadlock.
- **for**, **sections**, and **single** directives are not permitted in the dynamic extent of **critical**, **ordered**, and **master** regions if the directives bind to the same **parallel** as the regions.
- **barrier** directives are not permitted in the dynamic extent of **for**, **ordered**, **sections**, **single**, **master**, and **critical** regions if the directives bind to the same **parallel** as the regions.
- **master** directives are not permitted in the dynamic extent of **for**, **sections**, and **single** directives if the **master** directives bind to the same **parallel** as the work-sharing directives.
- **ordered** directives are not allowed in the dynamic extent of **critical** regions if the directives bind to the same **parallel** as the regions.
- Any directive that is permitted when executed dynamically inside a parallel region is also permitted when executed outside a parallel region. When executed dynamically outside a user-specified parallel region, the directive is executed by a team composed of only the master thread.

Run-time Library Functions

This section describes the OpenMP C and C++ run-time library functions. The header `<omp.h>` declares two types, several functions that can be used to control and query the parallel execution environment, and lock functions that can be used to synchronize access to data.

The type `omp_lock_t` is an object type capable of representing that a lock is available, or that a thread owns a lock. These locks are referred to as *simple locks*.

The type `omp_nest_lock_t` is an object type capable of representing either that a lock is available, or both the identity of the thread that owns the lock and a *nesting count* (described below). These locks are referred to as *nestable locks*.

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

The descriptions in this chapter are divided into the following topics:

- Execution environment functions (see Section 3.1 on page 35).
- Lock functions (see Section 3.2 on page 41).

3.1 Execution Environment Functions

The functions described in this section affect and monitor threads, processors, and the parallel environment:

- the `omp_set_num_threads` function.
- the `omp_get_num_threads` function.
- the `omp_get_max_threads` function.
- the `omp_get_thread_num` function.
- the `omp_get_num_procs` function.
- the `omp_in_parallel` function.

- 1 ■ the `omp_set_dynamic` function.
- 2 ■ the `omp_get_dynamic` function.
- 3 ■ the `omp_set_nested` function.
- 4 ■ the `omp_get_nested` function.

5 3.1.1 `omp_set_num_threads` Function

6 The `omp_set_num_threads` function sets the default number of threads to use
7 for subsequent parallel regions that do not specify a `num_threads` clause. The
8 format is as follows:

```
9 #include <omp.h>  
10 void omp_set_num_threads(int num_threads);
```

11 The value of the parameter `num_threads` must be a positive integer. Its effect depends
12 upon whether dynamic adjustment of the number of threads is enabled. For a
13 comprehensive set of rules about the interaction between the
14 `omp_set_num_threads` function and dynamic adjustment of threads, see
15 Section 2.3 on page 8.

16 This function has the effects described above when called from a portion of the
17 program where the `omp_in_parallel` function returns zero. If it is called from a
18 portion of the program where the `omp_in_parallel` function returns a nonzero
19 value, the behavior of this function is undefined.

20 This call has precedence over the `OMP_NUM_THREADS` environment variable. The
21 default value for the number of threads, which may be established by calling
22 `omp_set_num_threads` or by setting the `OMP_NUM_THREADS` environment
23 variable, can be explicitly overridden on a single `parallel` directive by specifying
24 the `num_threads` clause.

25 Cross References:

- 26 ■ `omp_set_dynamic` function, see Section 3.1.7 on page 39.
- 27 ■ `omp_get_dynamic` function, see Section 3.1.8 on page 40.
- 28 ■ `OMP_NUM_THREADS` environment variable, see Section 4.2 on page 48, and
29 Section 2.3 on page 8.
- 30 ■ `num_threads` clause, see Section 2.3 on page 8

3.1.2 `omp_get_num_threads` Function

The `omp_get_num_threads` function returns the number of threads currently in the team executing the parallel region from which it is called. The format is as follows:

```
#include <omp.h>
int omp_get_num_threads(void);
```

The `num_threads` clause, the `omp_set_num_threads` function, and the `OMP_NUM_THREADS` environment variable control the number of threads in a team.

If the number of threads has not been explicitly set by the user, the default is implementation-defined. This function binds to the closest enclosing `parallel` directive. If called from a serial portion of a program, or from a nested parallel region that is serialized, this function returns 1.

Cross References:

- `OMP_NUM_THREADS` environment variable, see Section 4.2 on page 48.
- `num_threads` clause, see Section 2.3 on page 8.
- `parallel` construct, see Section 2.3 on page 8.

3.1.3 `omp_get_max_threads` Function

The `omp_get_max_threads` function returns an integer that is guaranteed to be at least as large as the number of threads that would be used to form a team if a parallel region without a `num_threads` clause were to be encountered at that point in the code. The format is as follows:

```
#include <omp.h>
int omp_get_max_threads(void);
```

The following expresses a lower bound on the value of `omp_get_max_threads`:

threads-used-for-next-team \leq `omp_get_max_threads`

Note that if a subsequent parallel region uses the `num_threads` clause to request a specific number of threads, the guarantee on the lower bound of the result of `omp_get_max_threads` no longer holds.

The `omp_get_max_threads` function's return value can be used to dynamically allocate sufficient storage for all threads in the team formed at the subsequent parallel region.

Cross References:

- `omp_get_num_threads` function, see Section 3.1.2 on page 37.
- `omp_set_num_threads` function, see Section 3.1.1 on page 36.
- `omp_set_dynamic` function, see Section 3.1.7 on page 39.
- `num_threads` clause, see Section 2.3 on page 8.

3.1.4 `omp_get_thread_num` Function

The `omp_get_thread_num` function returns the thread number, within its team, of the thread executing the function. The thread number lies between 0 and `omp_get_num_threads() - 1`, inclusive. The master thread of the team is thread 0. The format is as follows:

```
#include <omp.h>
int omp_get_thread_num(void);
```

If called from a serial region, `omp_get_thread_num` returns 0. If called from within a nested parallel region that is serialized, this function returns 0.

Cross References:

- `omp_get_num_threads` function, see Section 3.1.2 on page 37.

3.1.5 `omp_get_num_procs` Function

The `omp_get_num_procs` function returns the number of processors that are available to the program at the time the function is called. The format is as follows:

```
#include <omp.h>
int omp_get_num_procs(void);
```

3.1.6 `omp_in_parallel` Function

The `omp_in_parallel` function returns a nonzero value if it is called within the dynamic extent of a parallel region executing in parallel; otherwise, it returns 0. The format is as follows:

```
#include <omp.h>
int omp_in_parallel(void);
```

1 This function returns a nonzero value when called from within a region executing in
2 parallel, including nested regions that are serialized.

3 3.1.7 **omp_set_dynamic** Function

4 The **omp_set_dynamic** function enables or disables dynamic adjustment of the
5 number of threads available for execution of parallel regions. The format is as
6 follows:

```
7 #include <omp.h>  
8 void omp_set_dynamic(int dynamic_threads);
```

9 If *dynamic_threads* evaluates to a nonzero value, the number of threads that are used
10 for executing subsequent parallel regions may be adjusted automatically by the run-
11 time environment to best utilize system resources. As a consequence, the number of
12 threads specified by the user is the maximum thread count. The number of threads
13 in the team executing a parallel region remains fixed for the duration of that parallel
14 region and is reported by the **omp_get_num_threads** function.

15 If *dynamic_threads* evaluates to 0, dynamic adjustment is disabled.

16 This function has the effects described above when called from a portion of the
17 program where the **omp_in_parallel** function returns zero. If it is called from a
18 portion of the program where the **omp_in_parallel** function returns a nonzero
19 value, the behavior of this function is undefined.

20 A call to **omp_set_dynamic** has precedence over the **OMP_DYNAMIC** environment
21 variable.

22 The default for the dynamic adjustment of threads is implementation-defined. As a
23 result, user codes that depend on a specific number of threads for correct execution
24 should explicitly disable dynamic threads. Implementations are not required to
25 provide the ability to dynamically adjust the number of threads, but they are
26 required to provide the interface in order to support portability across all platforms.

27 **Cross References:**

- 28 ■ **omp_get_num_threads** function, see Section 3.1.2 on page 37.
- 29 ■ **OMP_DYNAMIC** environment variable, see Section 4.3 on page 49.
- 30 ■ **omp_in_parallel** function, see Section 3.1.6 on page 38.

3.1.8 `omp_get_dynamic` Function

The `omp_get_dynamic` function returns a nonzero value if dynamic adjustment of threads is enabled, and returns 0 otherwise. The format is as follows:

```
#include <omp.h>
int omp_get_dynamic(void);
```

If the implementation does not implement dynamic adjustment of the number of threads, this function always returns 0.

Cross References:

- For a description of dynamic thread adjustment, see Section 3.1.7 on page 39.

3.1.9 `omp_set_nested` Function

The `omp_set_nested` function enables or disables nested parallelism. The format is as follows:

```
#include <omp.h>
void omp_set_nested(int nested);
```

If *nested* evaluates to 0, nested parallelism is disabled, which is the default, and nested parallel regions are serialized and executed by the current thread. If *nested* evaluates to a nonzero value, nested parallelism is enabled, and parallel regions that are nested may deploy additional threads to form nested teams.

This function has the effects described above when called from a portion of the program where the `omp_in_parallel` function returns zero. If it is called from a portion of the program where the `omp_in_parallel` function returns a nonzero value, the behavior of this function is undefined.

This call has precedence over the `OMP_NESTED` environment variable.

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

Cross References:

- `OMP_NESTED` environment variable, see Section 4.4 on page 49.
- `omp_in_parallel` function, see Section 3.1.6 on page 38.

3.1.10 `omp_get_nested` Function

The `omp_get_nested` function returns a nonzero value if nested parallelism is enabled and 0 if it is disabled. For more information on nested parallelism, see Section 3.1.9 on page 40. The format is as follows:

```
#include <omp.h>
int omp_get_nested(void);
```

If an implementation does not implement nested parallelism, this function always returns 0.

3.2 Lock Functions

The functions described in this section manipulate locks used for synchronization.

For the following functions, the lock variable must have type `omp_lock_t`. This variable must only be accessed through these functions. All lock functions require an argument that has a pointer to `omp_lock_t` type.

- The `omp_init_lock` function initializes a simple lock.
- The `omp_destroy_lock` function removes a simple lock.
- The `omp_set_lock` function waits until a simple lock is available.
- The `omp_unset_lock` function releases a simple lock.
- The `omp_test_lock` function tests a simple lock.

For the following functions, the lock variable must have type `omp_nest_lock_t`. This variable must only be accessed through these functions. All nestable lock functions require an argument that has a pointer to `omp_nest_lock_t` type.

- The `omp_init_nest_lock` function initializes a nestable lock.
- The `omp_destroy_nest_lock` function removes a nestable lock.
- The `omp_set_nest_lock` function waits until a nestable lock is available.
- The `omp_unset_nest_lock` function releases a nestable lock.
- The `omp_test_nest_lock` function tests a nestable lock.

The OpenMP lock functions access the lock variable in such a way that they always read and update the most current value of the lock variable. Therefore, it is not necessary for an OpenMP program to include explicit `flush` directives to ensure that the lock variable's value is consistent among different threads. (There may be a need for `flush` directives to make the values of other variables consistent.)

3.2.1 `omp_init_lock` and `omp_init_nest_lock` Functions

These functions provide the only means of initializing a lock. Each function initializes the lock associated with the parameter *lock* for use in subsequent calls. The format is as follows:

```
#include <omp.h>
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

The initial state is unlocked (that is, no thread owns the lock). For a nestable lock, the initial nesting count is zero. It is noncompliant to call either of these routines with a lock variable that has already been initialized.

3.2.2 `omp_destroy_lock` and `omp_destroy_nest_lock` Functions

These functions ensure that the pointed to lock variable *lock* is uninitialized. The format is as follows:

```
#include <omp.h>
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

It is noncompliant to call either of these routines with a lock variable that is uninitialized or unlocked.

3.2.3 `omp_set_lock` and `omp_set_nest_lock` Functions

Each of these functions blocks the thread executing the function until the specified lock is available and then sets the lock. A simple lock is available if it is unlocked. A nestable lock is available if it is unlocked or if it is already owned by the thread executing the function. The format is as follows:

```
#include <omp.h>
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
```


1 For a simple lock, the argument to the `omp_set_lock` function must point to an
2 initialized lock variable. Ownership of the lock is granted to the thread executing the
3 function.

4 For a nestable lock, the argument to the `omp_set_nest_lock` function must point
5 to an initialized lock variable. The nesting count is incremented, and the thread is
6 granted, or retains, ownership of the lock.

7 3.2.4 `omp_unset_lock` and `omp_unset_nest_lock` 8 Functions

9 These functions provide the means of releasing ownership of a lock. The format is as
10 follows:

```
11 #include <omp.h>  
12 void omp_unset_lock(omp_lock_t *lock);  
13 void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

14 The argument to each of these functions must point to an initialized lock variable
15 owned by the thread executing the function. The behavior is undefined if the thread
16 does not own that lock.

17 For a simple lock, the `omp_unset_lock` function releases the thread executing the
18 function from ownership of the lock.

19 For a nestable lock, the `omp_unset_nest_lock` function decrements the nesting
20 count, and releases the thread executing the function from ownership of the lock if
21 the resulting count is zero.

22 3.2.5 `omp_test_lock` and `omp_test_nest_lock` 23 Functions

24 These functions attempt to set a lock but do not block execution of the thread. The
25 format is as follows:

```
26 #include <omp.h>  
27 int omp_test_lock(omp_lock_t *lock);  
28 int omp_test_nest_lock(omp_nest_lock_t *lock);
```

29 The argument must point to an initialized lock variable. These functions attempt to
30 set a lock in the same manner as `omp_set_lock` and `omp_set_nest_lock`,
31 except that they do not block execution of the thread.

1 For a simple lock, the `omp_test_lock` function returns a nonzero value if the lock
2 is successfully set; otherwise, it returns zero.

3 For a nestable lock, the `omp_test_nest_lock` function returns the new nesting
4 count if the lock is successfully set; otherwise, it returns zero.

5 3.3 Timing Routines

6 The functions described in this section support a portable wall-clock timer:

- 7 ■ The `omp_get_wtime` function returns elapsed wall-clock time.
- 8 ■ The `omp_get_wtick` function returns seconds between successive clock ticks.

9 3.3.1 `omp_get_wtime` Function

10 The `omp_get_wtime` function returns a double-precision floating point value
11 equal to the elapsed wall clock time in seconds since some “time in the past”. The
12 actual “time in the past” is arbitrary, but it is guaranteed not to change during the
13 execution of the application program. The format is as follows:

```
14 #include <omp.h>  
15 double omp_get_wtime(void);
```

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

```
18 double start;  
19 double end;  
20 start = omp_get_wtime();  
21 ... work to be timed ...  
22 end = omp_get_wtime();  
23 printf("Work took %f sec. time.\n", end-start);
```

24 The times returned are “per-thread times” by which is meant they are not required
25 to be globally consistent across all the threads participating in an application.

3.3.2 `omp_get_wtick` Function

The `omp_get_wtick` function returns a double-precision floating point value equal to the number of seconds between successive clock ticks. The format is as follows:

```
#include <omp.h>
double omp_get_wtick(void);
```


Environment Variables

This chapter describes the OpenMP C and C++ API environment variables (or equivalent platform-specific mechanisms) that control the execution of parallel code. The names of environment variables must be uppercase. The values assigned to them are case insensitive and may have leading and trailing white space. Modifications to the values after the program has started are ignored.

The environment variables are as follows:

- **OMP_SCHEDULE** sets the run-time schedule type and chunk size.
- **OMP_NUM_THREADS** sets the number of threads to use during execution.
- **OMP_DYNAMIC** enables or disables dynamic adjustment of the number of threads.
- **OMP_NESTED** enables or disables nested parallelism.

The examples in this chapter only demonstrate how these variables might be set in Unix C shell (csh) environments. In Korn shell and DOS environments the actions are similar, as follows:

- csh:

```
setenv OMP_SCHEDULE "dynamic"
```

- ksh:

```
export OMP_SCHEDULE="dynamic"
```

- DOS:

```
set OMP_SCHEDULE="dynamic"
```

4.1 OMP_SCHEDULE

`OMP_SCHEDULE` applies only to **for** and **parallel for** directives that have the schedule type **runtime**. The schedule type and chunk size for all such loops can be set at run time by setting this environment variable to any of the recognized schedule types and to an optional *chunk_size*.

For **for** and **parallel for** directives that have a schedule type other than **runtime**, `OMP_SCHEDULE` is ignored. The default value for this environment variable is implementation-defined. If the optional *chunk_size* is set, the value must be positive. If *chunk_size* is not set, a value of 1 is assumed, except in the case of a **static** schedule. For a **static** schedule, the default chunk size is set to the loop iteration space divided by the number of threads applied to the loop.

Example:

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

Cross References:

- **for** directive, see Section 2.4.1 on page 11.
- **parallel for** directive, see Section 2.5.1 on page 16.

4.2 OMP_NUM_THREADS

The `OMP_NUM_THREADS` environment variable sets the default number of threads to use during execution, unless that number is explicitly changed by calling the `omp_set_num_threads` library routine or by an explicit `num_threads` clause on a **parallel** directive.

The value of the `OMP_NUM_THREADS` environment variable must be a positive integer. Its effect depends upon whether dynamic adjustment of the number of threads is enabled. For a comprehensive set of rules about the interaction between the `OMP_NUM_THREADS` environment variable and dynamic adjustment of threads, see Section 2.3 on page 8.

If no value is specified for the `OMP_NUM_THREADS` environment variable, or if the value specified is not a positive integer, or if the value is greater than the maximum number of threads the system can support, the number of threads to use is implementation-defined.

1 Example:

```
2 setenv OMP_NUM_THREADS 16
```

3 Cross References:

- 4 ■ `num_threads` clause, see Section 2.3 on page 8.
- 5 ■ `omp_set_num_threads` function, see Section 3.1.1 on page 36.
- 6 ■ `omp_set_dynamic` function, see Section 3.1.7 on page 39.

7 4.3 OMP_DYNAMIC

8 The `OMP_DYNAMIC` environment variable enables or disables dynamic adjustment
9 of the number of threads available for execution of parallel regions unless dynamic
10 adjustment is explicitly enabled or disabled by calling the `omp_set_dynamic`
11 library routine. Its value must be `TRUE` or `FALSE`.

12 If set to `TRUE`, the number of threads that are used for executing parallel regions
13 may be adjusted by the runtime environment to best utilize system resources.

14 If set to `FALSE`, dynamic adjustment is disabled. The default condition is
15 implementation-defined.

16 Example:

```
17 setenv OMP_DYNAMIC TRUE
```

18 Cross References:

- 19 ■ For more information on parallel regions, see Section 2.3 on page 8.
- 20 ■ `omp_set_dynamic` function, see Section 3.1.7 on page 39.

21 4.4 OMP_NESTED

22 The `OMP_NESTED` environment variable enables or disables nested parallelism
23 unless nested parallelism is enabled or disabled by calling the `omp_set_nested`
24 library routine. If set to `TRUE`, nested parallelism is enabled; if it is set to `FALSE`,
25 nested parallelism is disabled. The default value is `FALSE`.

1 Example:

```
2 setenv OMP_NESTED TRUE
```

3 **Cross Reference:**

- 4 ■ `omp_set_nested` function, see Section 3.1.9 on page 40.

2

Examples

3 The following are examples of the constructs defined in this document. Note that a
4 statement following a directive is compound only when necessary, and a non-
5 compound statement is indented with respect to a directive preceding it.

6 A.1 Executing a Simple Loop in Parallel

7 The following example demonstrates how to parallelize a simple loop using the
8 **parallel for** directive (Section 2.5.1 on page 16). The loop iteration variable is
9 private by default, so it is not necessary to specify it explicitly in a private clause.

```
10 #pragma omp parallel for  
11   for (i=1; i<n; i++)  
12     b[i] = (a[i] + a[i-1]) / 2.0;
```

13 A.2 Specifying Conditional Compilation

14 The following examples illustrate the use of conditional compilation using the
15 OpenMP macro `_OPENMP` (Section 2.2 on page 8). With OpenMP compilation, the
16 `_OPENMP` macro becomes defined.

```
17 # ifdef _OPENMP  
18   printf("Compiled by an OpenMP-compliant implementation.\n");  
19 # endif
```

1 The defined preprocessor operator allows more than one macro to be tested in a
2 single directive.

```
3 # if defined(_OPENMP) && defined(VERBOSE)  
4     printf("Compiled by an OpenMP-compliant implementation.\n");  
5 # endif
```

6 A.3 Using Parallel Regions

7 The `parallel` directive (Section 2.3 on page 8) can be used in coarse-grain parallel
8 programs. In the following example, each thread in the parallel region decides what
9 part of the global array `x` to work on, based on the thread number:

```
10 #pragma omp parallel shared(x, npoints) private(iam, np, ipoints)  
11 {  
12     iam = omp_get_thread_num();  
13     np = omp_get_num_threads();  
14     ipoints = npoints / np;  
15     subdomain(x, iam, ipoints);  
16 }
```

17 A.4 Using the `nowait` Clause

18 If there are multiple independent loops within a parallel region, you can use the
19 `nowait` clause (Section 2.4.1 on page 11) to avoid the implied barrier at the end of
20 the `for` directive, as follows:

```
21 #pragma omp parallel  
22 {  
23     #pragma omp for nowait  
24     for (i=1; i<n; i++)  
25         b[i] = (a[i] + a[i-1]) / 2.0;  
26     #pragma omp for nowait  
27     for (i=0; i<m; i++)  
28         y[i] = sqrt(z[i]);  
29 }
```

1 A.5 Using the `critical` Directive

2 The following example includes several `critical` directives (Section 2.6.2 on page
3 18). The example illustrates a queuing model in which a task is dequeued and
4 worked on. To guard against multiple threads dequeuing the same task, the
5 dequeuing operation must be in a `critical` section. Because the two queues in
6 this example are independent, they are protected by `critical` directives with
7 different names, *xaxis* and *yaxis*.

```
8 #pragma omp parallel shared(x, y) private(x_next, y_next)  
9 {  
10     #pragma omp critical ( xaxis )  
11         x_next = dequeue(x);  
12     work(x_next);  
13     #pragma omp critical ( yaxis )  
14         y_next = dequeue(y);  
15     work(y_next);  
16 }
```

17 A.6 Using the `lastprivate` Clause

18 Correct execution sometimes depends on the value that the last iteration of a loop
19 assigns to a variable. Such programs must list all such variables as arguments to a
20 `lastprivate` clause (Section 2.7.2.3 on page 27) so that the values of the variables
21 are the same as when the loop is executed sequentially.

```
22 #pragma omp parallel  
23 {  
24     #pragma omp for lastprivate(i)  
25         for (i=0; i<n-1; i++)  
26             a[i] = b[i] + b[i+1];  
27 }  
28 a[i]=b[i];
```

29 In the preceding example, the value of *i* at the end of the parallel region will equal
30 *n-1*, as in the sequential case.

1 A.7 Using the reduction Clause

2 The following example demonstrates the **reduction** clause (Section 2.7.2.6 on page
3 28):

```
4 #pragma omp parallel for private(i) shared(x, y, n) \  
5 reduction(+: a, b)  
6 for (i=0; i<n; i++) {  
7     a = a + x[i];  
8     b = b + y[i];  
9 }
```

10 A.8 Specifying Parallel Sections

11 In the following example, (for Section 2.4.2 on page 14) functions *xaxis*, *yaxis*, and
12 *zaxis* can be executed concurrently. The first **section** directive is optional. Note
13 that all **section** directives need to appear in the lexical extent of the
14 **parallel sections** construct.

```
15 #pragma omp parallel sections  
16 {  
17     #pragma omp section  
18     xaxis();  
19     #pragma omp section  
20     yaxis();  
21     #pragma omp section  
22     zaxis();  
23 }
```

24 A.9 Using single Directives

25 The following example demonstrates the **single** directive (Section 2.4.3 on page
26 15). In the example, only one thread (usually the first thread that encounters the
27 **single** directive) prints the progress message. The user must not make any
28 assumptions as to which thread will execute the **single** section. All other threads

1 will skip the **single** section and stop at the barrier at the end of the **single**
2 construct. If other threads can proceed without waiting for the thread executing the
3 **single** section, a **nowait** clause can be specified on the **single** directive.

```
4 #pragma omp parallel  
5 {  
6     #pragma omp single  
7     printf("Beginning work1.\n");  
8     work1();  
9     #pragma omp single  
10    printf("Finishing work1.\n");  
11    #pragma omp single nowait  
12    printf("Finished work1 and beginning work2.\n");  
13    work2();  
14 }
```

15 A.10 Specifying Sequential Ordering

16 Ordered sections (Section 2.6.6 on page 22) are useful for sequentially ordering the
17 output from work that is done in parallel. The following program prints out the
18 indexes in sequential order:

```
19 #pragma omp for ordered schedule(dynamic)  
20 for (i=lb; i<ub; i+=st)  
21     work(i);  
  
22 void work(int k)  
23 {  
24     #pragma omp ordered  
25     printf(" %d", k);  
26 }
```

27 A.11 Specifying a Fixed Number of Threads

28 Some programs rely on a fixed, prespecified number of threads to execute correctly.
29 Because the default setting for the dynamic adjustment of the number of threads is
30 implementation-defined, such programs can choose to turn off the dynamic threads

1 capability and set the number of threads explicitly to ensure portability. The
2 following example shows how to do this using `omp_set_dynamic` (Section 3.1.7
3 on page 39), and `omp_set_num_threads` (Section 3.1.1 on page 36):

```
4  omp_set_dynamic(0);  
5  omp_set_num_threads(16);  
6  #pragma omp parallel shared(x, npoints) private(iam, ipoints)  
7  {  
8      if (omp_get_num_threads() != 16) abort();  
9      iam = omp_get_thread_num();  
10     ipoints = npoints/16;  
11     do_by_16(x, iam, ipoints);  
12 }
```

13 In this example, the program executes correctly only if it is executed by 16 threads. If
14 the implementation is not capable of supporting 16 threads, the behavior of this
15 example is implementation-defined.

16 Note that the number of threads executing a parallel region remains constant during
17 a parallel region, regardless of the dynamic threads setting. The dynamic threads
18 mechanism determines the number of threads to use at the start of the parallel
19 region and keeps it constant for the duration of the region.

20 A.12 Using the `atomic` Directive

21 The following example avoids race conditions (simultaneous updates of an element
22 of x by multiple threads) by using the `atomic` directive (Section 2.6.4 on page 19):

```
23 #pragma omp parallel for shared(x, y, index, n)  
24     for (i=0; i<n; i++) {  
25         #pragma omp atomic  
26         x[index[i]] += work1(i);  
27         y[i] += work2(i);  
28     }
```

29 The advantage of using the `atomic` directive in this example is that it allows
30 updates of two different elements of x to occur in parallel. If a `critical` directive
31 (Section 2.6.2 on page 18) were used instead, then all updates to elements of x would
32 be executed serially (though not in any guaranteed order).

33 Note that the `atomic` directive applies only to the C or C++ statement immediately
34 following it. As a result, elements of y are not updated atomically in this example.

A.13 Using the `flush` Directive with a List

The following example uses the `flush` directive for point-to-point synchronization of specific objects between pairs of threads:

```
1
2
3
4     int    sync[NUMBER_OF_THREADS];
5     float work[NUMBER_OF_THREADS];
6     #pragma omp parallel private(iam,neighbor) shared(work, sync)
7     {
8
9         iam = omp_get_thread_num();
10        sync[iam] = 0;
11        #pragma omp barrier
12
13        /*Do computation into my portion of work array */
14        work[iam] = ...;
15
16        /* Announce that I am done with my work
17         * The first flush ensures that my work is
18         * made visible before sync.
19         * The second flush ensures that sync is made visible.
20         */
21        #pragma omp flush(work)
22        sync[iam] = 1;
23        #pragma omp flush(sync)
24
25        /*Wait for neighbor*/
26        neighbor = (iam>0 ? iam : omp_get_num_threads()) - 1;
27        while (sync[neighbor]==0) {
28            #pragma omp flush(sync)
29        }
30
31        /*Read neighbor's values of work array */
32        ... = work[neighbor];
33    }
```

A.14 Using the `flush` Directive without a List

The following example (for Section 2.6.5 on page 20) distinguishes the shared objects affected by a `flush` directive with no list from the shared objects that are not affected:

```

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

```

1 A.15 Determining the Number of Threads Used

2 Consider the following incorrect example (for Section 3.1.2 on page 37):

```
3 np = omp_get_num_threads(); /* misplaced */  
4 #pragma omp parallel for schedule(static)  
5   for (i=0; i<np; i++)  
6     work(i);
```

7 The `omp_get_num_threads()` call returns 1 in the serial section of the code, so
8 *np* will always be equal to 1 in the preceding example. To determine the number of
9 threads that will be deployed for the parallel region, the call should be inside the
10 parallel region.

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

```
13 #pragma omp parallel private(i)  
14 {  
15   i = omp_get_thread_num();  
16   work(i);  
17 }
```

18 A.16 Using Locks

19 In the following example, (for Section 3.2 on page 41) note that the argument to the
20 lock functions should have type `omp_lock_t`, and that there is no need to flush it.
21 The lock functions cause the threads to be idle while waiting for entry to the first

1 critical section, but to do other work while waiting for entry to the second. The
2 `omp_set_lock` function blocks, but the `omp_test_lock` function does not,
3 allowing the work in `skip()` to be done.

```
4 #include <omp.h>
5 int main()
6 {
7     omp_lock_t lck;
8     int id;
9
10    omp_init_lock(&lck);
11    #pragma omp parallel shared(lck) private(id)
12    {
13        id = omp_get_thread_num();
14
15        omp_set_lock(&lck);
16        printf("My thread id is %d.\n", id);
17        // only one thread at a time can execute this printf
18        omp_unset_lock(&lck);
19
20        while (! omp_test_lock(&lck)) {
21            skip(id); /* we do not yet have the lock,
22                       so we must do something else */
23        }
24        work(id); /* we now have the lock
25                  and can do the work */
26        omp_unset_lock(&lck);
27    }
28    omp_destroy_lock(&lck);
29 }
```

A.17 Using Nestable Locks

The following example (for Section 3.2 on page 41) demonstrates how a nestable lock can be used to synchronize updates both to a whole structure and to one of its members.

```
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38

#include <omp.h>
typedef struct {int a,b; omp_nest_lock_t lck;} pair;

void incr_a(pair *p, int a)
{
    // Called only from incr_pair, no need to lock.
    p->a += a;
}

void incr_b(pair *p, int b)
{
    // Called both from incr_pair and elsewhere,
    // so need a nestable lock.

    omp_set_nest_lock(&p->lck);
    p->b += b;
    omp_unset_nest_lock(&p->lck);
}

void incr_pair(pair *p, int a, int b)
{
    omp_set_nest_lock(&p->lck);
    incr_a(p, a);
    incr_b(p, b);
    omp_unset_nest_lock(&p->lck);
}

void f(pair *p)
{
    extern int work1(), work2(), work3();
    #pragma omp parallel sections
    {
        #pragma omp section
        incr_pair(p, work1(), work2());
        #pragma omp section
        incr_b(p, work3());
    }
}
```

A.18 Nested for Directives

The following example of `for` directive nesting (Section 2.9 on page 33) is compliant because the inner and outer `for` directives bind to different parallel regions:

```
#pragma omp parallel default(shared)
{
    #pragma omp for
    for (i=0; i<n; i++) {
        #pragma omp parallel shared(i, n)
        {
            #pragma omp for
            for (j=0; j<n; j++)
                work(i, j);
        }
    }
}
```

A following variation of the preceding example is also compliant:

```
#pragma omp parallel default(shared)
{
    #pragma omp for
    for (i=0; i<n; i++)
        work1(i, n);
}

void work1(int i, int n)
{
    int j;
    #pragma omp parallel default(shared)
    {
        #pragma omp for
        for (j=0; j<n; j++)
            work2(i, j);
    }
    return;
}
```

A.19 Examples Showing Incorrect Nesting of Work-sharing Directives

The examples in this section illustrate the directive nesting rules. For more information on directive nesting, see Section 2.9 on page 33.

The following example is noncompliant because the inner and outer **for** directives are nested and bind to the same **parallel** directive:

```
void wrong1(int n)
{
    #pragma omp parallel default(shared)
    {
        int i, j;
        #pragma omp for
        for (i=0; i<n; i++) {
            #pragma omp for
            for (j=0; j<n; j++)
                work(i, j);
        }
    }
}
```

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

```
void wrong2(int n)
{
    #pragma omp parallel default(shared)
    {
        int i;
        #pragma omp for
        for (i=0; i<n; i++)
            work1(i, n);
    }
}

void work1(int i, int n)
{
    int j;
    #pragma omp for
    for (j=0; j<n; j++)
        work2(i, j);
}
```

1 The following example is noncompliant because the **for** and **single** directives are
2 nested, and they bind to the same parallel region:

```
3 void wrong3(int n)
4 {
5     #pragma omp parallel default(shared)
6     {
7         int i;
8         #pragma omp for
9         for (i=0; i<n; i++) {
10            #pragma omp single
11            work(i);
12        }
13    }
14 }
```

15 The following example is noncompliant because a **barrier** directive inside a **for**
16 can result in deadlock:

```
17 void wrong4(int n)
18 {
19     #pragma omp parallel default(shared)
20     {
21         int i;
22         #pragma omp for
23         for (i=0; i<n; i++) {
24            work1(i);
25            #pragma omp barrier
26            work2(i);
27        }
28    }
29 }
```

1 The following example is noncompliant because the **barrier** results in deadlock
2 due to the fact that only one thread at a time can enter the critical section:

```
3 void wrong5()  
4 {  
5     #pragma omp parallel  
6     {  
7         #pragma omp critical  
8         {  
9             work1();  
10            #pragma omp barrier  
11            work2();  
12        }  
13    }  
14 }
```

15 The following example is noncompliant because the **barrier** results in deadlock
16 due to the fact that only one thread executes the **single** section:

```
17 void wrong6()  
18 {  
19     #pragma omp parallel  
20     {  
21         setup();  
22         #pragma omp single  
23         {  
24             work1();  
25             #pragma omp barrier  
26             work2();  
27         }  
28         finish();  
29     }  
30 }
```

31 A.20 Binding of barrier Directives

32 The directive binding rules call for a **barrier** directive to bind to the closest
33 enclosing **parallel** directive. For more information on directive binding, see
34 Section 2.8 on page 32.

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

1 The call from *main* to *sub3* is compliant because the **barrier** does not bind to any
2 parallel region and is ignored. Also note that the **barrier** only synchronizes the
3 team of threads in the enclosing parallel region and not all the threads created in
4 *sub1*.

```
5 int main()  
6 {  
7     sub1(2);  
8     sub2(2);  
9     sub3(2);  
10 }  
  
11 void sub1(int n)  
12 {  
13     int i;  
14     #pragma omp parallel private(i) shared(n)  
15     {  
16         #pragma omp for  
17         for (i=0; i<n; i++)  
18             sub2(i);  
19     }  
20 }  
  
21 void sub2(int k)  
22 {  
23     #pragma omp parallel shared(k)  
24         sub3(k);  
25 }  
  
26 void sub3(int n)  
27 {  
28     work(n);  
29     #pragma omp barrier  
30     work(n);  
31 }
```

1 **A.21 Scoping Variables with the `private`** 2 ****Clause****

3 The values of *i* and *j* in the following example are undefined on exit from the parallel
4 region:

```
5     int i, j;  
6     i = 1;  
7     j = 2;  
8     #pragma omp parallel private(i) firstprivate(j)  
9     {  
10        i = 3;  
11        j = j + 2;  
12     }  
13     printf("%d %d\n", i, j);
```

14 For more information on the `private` clause, see Section 2.7.2.1 on page 25.

A.22 Using the `default(none)` Clause

The following example distinguishes the variables that are affected by the `default(none)` clause from those that are not:

```
int x, y, z[1000];
#pragma omp threadprivate(x)

void fun(int a) {
    const int c = 1;
    int i = 0;

    #pragma omp parallel default(none) private(a) shared(z)
    {
        int j = omp_get_num_thread();
        //O.K. - j is declared within parallel region
        a = z[j]; // O.K. - a is listed in private clause
                // - z is listed in shared clause
        x = c; // O.K. - x is threadprivate
                // - c has const-qualified type
        z[i] = y; // Error - cannot reference i or y here

        #pragma omp for firstprivate(y)
        for (i=0; i<10 ; i++) {
            z[i] = y; // O.K. - i is the loop control variable
                    // - y is listed in firstprivate clause
        }
        z[i] = y; // Error - cannot reference i or y here
    }
}
```

For more information on the `default` clause, see Section 2.7.2.5 on page 28.

A.23 Examples of the `ordered` Directive

It is possible to have multiple ordered sections with a `for` specified with the `ordered` clause. The first example is noncompliant because the API specifies the following:

“An iteration of a loop with a `for` construct must not execute the same `ordered` directive more than once, and it must not execute more than one `ordered` directive.” (See Section 2.6.6 on page 22)

1

In this noncompliant example, all iterations execute 2 ordered sections:

2
3
4
5
6
7
8
9
10
11

```
#pragma omp for ordered
for (i=0; i<n; i++) {
    ...
    #pragma omp ordered
    { ... }
    ...
    #pragma omp ordered
    { ... }
    ...
}
```

12

The following compliant example shows a **for** with more than one ordered section:

13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28

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

A.24 Example of the `private` Clause

The `private` clause (Section 2.7.2.1 on page 25) of a parallel region is only in effect for the lexical extent of the region, not for the dynamic extent of the region. Therefore, in the example that follows, any uses of the variable `a` within the `for` loop in the routine `f` refers to a private copy of `a`, while a usage in routine `g` refers to the global `a`.

```
int a;

void f(int n) {
    a = 0;

    #pragma omp parallel for private(a)
    for (int i=1; i<n; i++) {

        a = i;
        g(i, n);
        d(a);    // Private copy of "a"
        ...
    }
    ...
}

void g(int k, int n) {

    h(k,a); //The global "a", not the private "a" in f
}
```

A.25 Examples of the `copyprivate` Data Attribute Clause

Example 1: The `copyprivate` clause (Section 2.7.2.8 on page 32) can be used to broadcast values acquired by a single thread directly to all instances of the private variables in the other threads.

```
float x, y;
#pragma omp threadprivate(x, y)

void init( ) {
float a;
float b;

    #pragma omp single copyprivate(a,b,x,y)
    {
        get_values(a,b,x,y);
    }

    use_values(a, b, x, y);
}
```

If routine *init* is called from a serial region, its behavior is not affected by the presence of the directives. After the call to the *get_values* routine has been executed by one thread, no thread leaves the construct until the private objects designated by *a*, *b*, *x*, and *y* in all threads have become defined with the values read.

1 **Example 2:** In contrast to the previous example, suppose the read must be
2 performed by a particular thread, say the master thread. In this case, the
3 **copyprivate** clause cannot be used to do the broadcast directly, but it can be used
4 to provide access to a temporary shared object.

```
5           float read_next( ) {  
6           float * tmp;  
7           float return_val;  
  
8                 #pragma omp single copyprivate(tmp)  
9                 {  
10                    tmp = (float *) malloc(sizeof(float));  
11                 }  
  
12                 #pragma omp master  
13                 {  
14                    get_float( tmp );  
15                 }  
  
16                 #pragma omp barrier  
17                 return_val = *tmp;  
18                 #pragma omp barrier  
  
19                 #pragma omp single  
20                 {  
21                    free(tmp);  
22                 }  
  
23           return return_val;  
24           }
```

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

```
5           #include <omp.h>
6
7           omp_lock_t *new_lock()
8           {
9                 #pragma omp single copyprivate(lock_ptr)
10                 {
11                     lock_ptr = (omp_lock_t *) malloc(sizeof(omp_lock_t));
12                     omp_init_lock( lock_ptr );
13                 }
14           return lock_ptr;
15           }
```

1 A.26 Using the `threadprivate` Directive

2 The following examples demonstrate how to use the `threadprivate` directive
3 (Section 2.7.1 on page 23) to give each thread a separate counter.

4 **Example 1:**

```
5 int counter = 0;  
6 #pragma omp threadprivate(counter)  
  
7 int sub()  
8 {  
9     counter++;  
10    return(counter);  
11 }
```

12 **Example 2:**

```
13 int sub()  
14 {  
15     static int counter = 0;  
16     #pragma omp threadprivate(counter)  
17     counter++;  
18     return(counter);  
19 }
```

20 A.27 Use of C99 Variable Length Arrays

21 The following example demonstrates how to use C99 Variable Length Arrays (VLAs)
22 in a `firstprivate` directive (Section 2.7.2.2 on page 26).

```
23 void f(int m, int C[m][m])  
24 {  
25     double v1[m];  
26     ...  
27     #pragma omp parallel firstprivate(C, v1)  
28     ...  
29 }
```

1 **A.28 Use of num_threads Clause**

2 The following example demonstrates the **num_threads** clause (Section 2.3 on page
3 8). The parallel region is executed with a maximum of 10 threads.

```
4                    #include <omp.h>  
5                    main()  
6                    {  
7                        omp_set_dynamic(1);  
8                        ...  
9                        #pragma omp parallel num_threads(10)  
10                        {  
11                            ... parallel region ...  
12                        }  
13                    }
```

1 A.29 Use of Work-Sharing Constructs Inside a 2 **critical Construct**

3 The following example demonstrates using a work-sharing construct inside a
4 **critical** construct. This example is compliant because the work-sharing construct
5 and the **critical** construct do not bind to the same parallel region.

```
6    void f()  
7    {  
8        int i = 1;  
9        #pragma omp parallel sections  
10        {  
11            #pragma omp section  
12            {  
13                #pragma omp critical (name)  
14                {  
15                    #pragma omp parallel  
16                    {  
17                        #pragma omp single  
18                        {  
19                            i++;  
20                        }  
21                    }  
22                }  
23            }  
24        }  
25    }
```

A.30 Use of Reprivatization

The following example demonstrates the reprivatization of variables. Private variables can be marked `private` again in a nested directive. They do not have to be shared in the enclosing parallel region.

```
int i, a;
...
#pragma omp parallel private(a)
{
    ...
    #pragma omp parallel for private(a)
    for (i=0; i<10; i++)
        {
            ...
        }
}
```

A.31 Thread-Safe Lock Functions

The following C++ example demonstrates how to initialize an array of locks in a parallel region by using `omp_init_lock` (Section 3.2.1 on page 42).

```
#include <omp.h>

omp_lock_t *new_locks()
{
    int i;
    omp_lock_t *lock = new omp_lock_t[1000];
    #pragma omp parallel for private(i)
    for (i=0; i<1000; i++)
        {
            omp_init_lock(&lock[i]);
        }
    return lock;
}
```


2 Stubs for Run-time Library 3 Functions

4 This section provides stubs for the run-time library functions defined in the OpenMP
5 C and C++ API. The stubs are provided to enable portability to platforms that do not
6 support the OpenMP C and C++ API. On these platforms, OpenMP programs must
7 be linked with a library containing these stub functions. The stub functions assume
8 that the directives in the OpenMP program are ignored. As such, they emulate serial
9 semantics.

10 **Note** – The lock variable that appears in the lock functions must be accessed
11 exclusively through these functions. It should not be initialized or otherwise
12 modified in the user program. Users should not make assumptions about
13 mechanisms used by OpenMP C and C++ implementations to implement locks
14 based on the scheme used by the stub functions.

```
1      #include <stdio.h>
2      #include <stdlib.h>
3      #include "omp.h"
4      #ifdef __cplusplus
5      extern "C" {
6      #endif

7      void omp_set_num_threads(int num_threads)
8      {
9      }

10     int omp_get_num_threads(void)
11     {
12         return 1;
13     }

14     int omp_get_max_threads(void)
15     {
16         return 1;
17     }

18     int omp_get_thread_num(void)
19     {
20         return 0;
21     }

22     int omp_get_num_procs(void)
23     {
24         return 1;
25     }

26     void omp_set_dynamic(int dynamic_threads)
27     {
28     }

29     int omp_get_dynamic(void)
30     {
31         return 0;
32     }

33     int omp_in_parallel(void)
34     {
35         return 0;
36     }

37     void omp_set_nested(int nested)
38     {
39     }
```

```

1      int omp_get_nested(void)
2      {
3          return 0;
4      }
5
6      enum {UNLOCKED = -1, INIT, LOCKED};
7
8      void omp_init_lock(omp_lock_t *lock)
9      {
10         *lock = UNLOCKED;
11     }
12
13     void omp_destroy_lock(omp_lock_t *lock)
14     {
15         *lock = INIT;
16     }
17
18     void omp_set_lock(omp_lock_t *lock)
19     {
20         if (*lock == UNLOCKED) {
21             *lock = LOCKED;
22         } else if (*lock == LOCKED) {
23             fprintf(stderr, "error: deadlock in using lock variable\n");
24             exit(1);
25         } else {
26             fprintf(stderr, "error: lock not initialized\n");
27             exit(1);
28         }
29     }
30
31     void omp_unset_lock(omp_lock_t *lock)
32     {
33         if (*lock == LOCKED) {
34             *lock = UNLOCKED;
35         } else if (*lock == UNLOCKED) {
36             fprintf(stderr, "error: lock not set\n");
37             exit(1);
38         } else {
39             fprintf(stderr, "error: lock not initialized\n");
40             exit(1);
41         }
42     }

```

```

1  int omp_test_lock(omp_lock_t *lock)
2  {
3      if (*lock == UNLOCKED) {
4          *lock = LOCKED;
5          return 1;
6      } else if (*lock == LOCKED) {
7          return 0;
8      } else {
9          fprintf(stderr, "error: lock not initialized\n");
10         exit(1);
11     }
12 }

13 #ifndef OMP_NEST_LOCK_T
14 typedef struct { /* This really belongs in omp.h */
15     int owner;
16     int count;
17 } omp_nest_lock_t;
18 #endif

19 enum {MASTER = 0};

20 void omp_init_nest_lock(omp_nest_lock_t *lock)
21 {
22     lock->owner = UNLOCKED;
23     lock->count = 0;
24 }

25 void omp_destroy_nest_lock(omp_nest_lock_t *lock)
26 {
27     lock->owner = UNLOCKED;
28     lock->count = UNLOCKED;
29 }

30 void omp_set_nest_lock(omp_nest_lock_t *lock)
31 {
32     if (lock->owner == MASTER && lock->count >= 1) {
33         lock->count++;
34     } else if (lock->owner == UNLOCKED && lock->count == 0) {
35         lock->owner = MASTER;
36         lock->count = 1;
37     } else {
38         fprintf(stderr, "error: lock corrupted or not initialized\n");
39         exit(1);
40     }
41 }

```

```

1 void omp_unset_nest_lock(omp_nest_lock_t *lock)
2 {
3     if (lock->owner == MASTER && lock->count >= 1) {
4         lock->count--;
5         if (lock->count == 0) {
6             lock->owner = UNLOCKED;
7         }
8     } else if (lock->owner == UNLOCKED && lock->count == 0) {
9         fprintf(stderr, "error: lock not set\n");
10        exit(1);
11    } else {
12        fprintf(stderr, "error: lock corrupted or not initialized\n");
13        exit(1);
14    }
15 }

16 int omp_test_nest_lock(omp_nest_lock_t *lock)
17 {
18     omp_set_nest_lock(lock);
19     return lock->count;
20 }

21 double omp_get_wtime(void)
22 {
23     /* This function does not provide a working
24        wallclock timer. Replace it with a version
25        customized for the target machine.
26     */
27     return 0.0;
28 }

29 double omp_get_wtick(void)
30 {
31     /* This function does not provide a working
32        clock tick function. Replace it with
33        a version customized for the target machine.
34     */
35     return 365. * 86400.;
36 }

37 #ifdef __cplusplus
38 }
39 #endif

```

2

OpenMP C and C++ Grammar

3

C.1 Notation

4 The grammar rules consist of the name for a non-terminal, followed by a colon,
5 followed by replacement alternatives on separate lines.6 The syntactic expression $term_{opt}$ indicates that the term is optional within the
7 replacement.8 The syntactic expression $term_{optseq}$ is equivalent to $term-seq_{opt}$ with the following
9 additional rules:10 *term-seq* :11 *term*12 *term-seq term*13 *term-seq , term*

1 C.2 Rules

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

5 */* in C++ (ISO/IEC 14882:1998) */*

6 *statement-seq:*

7 *statement*

8 *openmp-directive*

9 *statement-seq statement*

10 *statement-seq openmp-directive*

11 */* in C90 (ISO/IEC 9899:1990) */*

12 *statement-list:*

13 *statement*

14 *openmp-directive*

15 *statement-list statement*

16 *statement-list openmp-directive*

17 */* in C99 (ISO/IEC 9899:1999) */*

18 *block-item:*

19 *declaration*

20 *statement*

21 *openmp-directive*

```

1      statement:
2          /* standard statements */
3          openmp-construct
4      openmp-construct:
5          parallel-construct
6          for-construct
7          sections-construct
8          single-construct
9          parallel-for-construct
10         parallel-sections-construct
11         master-construct
12         critical-construct
13         atomic-construct
14         ordered-construct
15     openmp-directive:
16         barrier-directive
17         flush-directive
18     structured-block:
19         statement
20     parallel-construct:
21         parallel-directive structured-block
22     parallel-directive:
23         # pragma omp parallel parallel-clauseoptseq new-line
24     parallel-clause:
25         unique-parallel-clause
26         data-clause

```

```

1      unique-parallel-clause:
2          if ( expression )
3          num_threads ( expression )
4      for-construct:
5          for-directive iteration-statement
6      for-directive:
7          # pragma omp for for-clauseoptseq new-line
8      for-clause:
9          unique-for-clause
10         data-clause
11         nowait
12     unique-for-clause:
13         ordered
14         schedule ( schedule-kind )
15         schedule ( schedule-kind , expression )
16     schedule-kind:
17         static
18         dynamic
19         guided
20         runtime
21     sections-construct:
22         sections-directive section-scope
23     sections-directive:
24         # pragma omp sections sections-clauseoptseq new-line
25     sections-clause:
26         data-clause
27         nowait

```

1 *section-scope:*
2 { *section-sequence* }
3 *section-sequence:*
4 *section-directive*_{opt} *structured-block*
5 *section-sequence section-directive structured-block*
6 *section-directive:*
7 **# pragma omp section** *new-line*
8 *single-construct:*
9 *single-directive structured-block*
10 *single-directive:*
11 **# pragma omp single** *single-clause*_{optseq} *new-line*
12 *single-clause:*
13 *data-clause*
14 **nowait**
15 *parallel-for-construct:*
16 *parallel-for-directive iteration-statement*
17 *parallel-for-directive:*
18 **# pragma omp parallel for** *parallel-for-clause*_{optseq} *new-line*
19 *parallel-for-clause:*
20 *unique-parallel-clause*
21 *unique-for-clause*
22 *data-clause*
23 *parallel-sections-construct:*
24 *parallel-sections-directive section-scope*
25 *parallel-sections-directive:*
26 **# pragma omp parallel sections** *parallel-sections-clause*_{optseq} *new-line*

1 *parallel-sections-clause:*
2 *unique-parallel-clause*
3 *data-clause*
4 *master-construct:*
5 *master-directive structured-block*
6 *master-directive:*
7 **# pragma omp master** *new-line*
8 *critical-construct:*
9 *critical-directive structured-block*
10 *critical-directive:*
11 **# pragma omp critical** *region-phrase_{opt} new-line*
12 *region-phrase:*
13 (*identifier*)
14 *barrier-directive:*
15 **# pragma omp barrier** *new-line*
16 *atomic-construct:*
17 *atomic-directive expression-statement*
18 *atomic-directive:*
19 **# pragma omp atomic** *new-line*
20 *flush-directive:*
21 **# pragma omp flush** *flush-vars_{opt} new-line*
22 *flush-vars:*
23 (*variable-list*)
24 *ordered-construct:*
25 *ordered-directive structured-block*
26 *ordered-directive:*
27 **# pragma omp ordered** *new-line*


```

1      declaration:
2          /* standard declarations */
3          threadprivate-directive
4      threadprivate-directive:
5          # pragma omp threadprivate ( variable-list ) new-line
6      data-clause:
7          private ( variable-list )
8          copyprivate ( variable-list )
9          firstprivate ( variable-list )
10         lastprivate ( variable-list )
11         shared ( variable-list )
12         default ( shared )
13         default ( none )
14         reduction ( reduction-operator : variable-list )
15         copyin ( variable-list )
16     reduction-operator:
17         One of: + * - & ^ | && ||
18     /* in C */
19     variable-list:
20         identifier
21         variable-list , identifier
22     /* in C++ */
23     variable-list:
24         id-expression
25         variable-list , id-expression

```


Using the `schedule` Clause

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

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

```
#pragma omp parallel
{
  #pragma omp for schedule(static)
  for(i=0; i<n; i++)
    a[i] = work1(i);
  #pragma omp for schedule(static)
  for(i=0; i<n; i++)
    if(i>=k) a[i] += work2(i);
}
```

In the remaining examples, it is assumed that memory access is not the dominant consideration, and, unless otherwise stated, that all threads receive comparable computational resources. In these cases, the choice of schedule for a `for` construct depends on all the shared work that is to be performed between the nearest preceding barrier and either the implied closing barrier or the nearest subsequent

1 barrier, if there is a **nowait** clause. For each kind of schedule, a short example
2 shows how that schedule kind is likely to be the best choice. A brief discussion
3 follows each example.

4 The **static** schedule is also appropriate for the simplest case, a parallel region
5 containing a single **for** construct, with each iteration requiring the same amount of
6 work.

```
7 #pragma omp parallel for schedule(static)
8 for(i=0; i<n; i++) {
9     invariant_amount_of_work(i);
10 }
```

11 The **static** schedule is characterized by the properties that each thread gets
12 approximately the same number of iterations as any other thread, and each thread
13 can independently determine the iterations assigned to it. Thus no synchronization
14 is required to distribute the work, and, under the assumption that each iteration
15 requires the same amount of work, all threads should finish at about the same time.

16 For a team of p threads, let $\text{ceiling}(n/p)$ be the integer q , which satisfies $n = p*q - r$ with
17 $0 \leq r < p$. One implementation of the **static** schedule for this example would
18 assign q iterations to the first $p-1$ threads, and $q-r$ iterations to the last thread.
19 Another acceptable implementation would assign q iterations to the first $p-r$ threads,
20 and $q-1$ iterations to the remaining r threads. This illustrates why a program should
21 not rely on the details of a particular implementation.

22 The **dynamic** schedule is appropriate for the case of a **for** construct with the
23 iterations requiring varying, or even unpredictable, amounts of work.

```
24 #pragma omp parallel for schedule(dynamic)
25 for(i=0; i<n; i++) {
26     unpredictable_amount_of_work(i);
27 }
```

28 The **dynamic** schedule is characterized by the property that no thread waits at the
29 barrier for longer than it takes another thread to execute its final iteration. This
30 requires that iterations be assigned one at a time to threads as they become available,
31 with synchronization for each assignment. The synchronization overhead can be
32 reduced by specifying a minimum chunk size k greater than 1, so that threads are
33 assigned k at a time until fewer than k remain. This guarantees that no thread waits
34 at the barrier longer than it takes another thread to execute its final chunk of (at
35 most) k iterations.

1 The **dynamic** schedule can be useful if the threads receive varying computational
2 resources, which has much the same effect as varying amounts of work for each
3 iteration. Similarly, the dynamic schedule can also be useful if the threads arrive at
4 the **for** construct at varying times, though in some of these cases the **guided**
5 schedule may be preferable.

6 The **guided** schedule is appropriate for the case in which the threads may arrive at
7 varying times at a **for** construct with each iteration requiring about the same
8 amount of work. This can happen if, for example, the **for** construct is preceded by
9 one or more sections or **for** constructs with **nowait** clauses.

```
10 #pragma omp parallel  
11 {  
12     #pragma omp sections nowait  
13     {  
14         // ...  
15     }  
16     #pragma omp for schedule(guided)  
17     for(i=0; i<n; i++) {  
18         invariant_amount_of_work(i);  
19     }  
20 }
```

21 Like **dynamic**, the **guided** schedule guarantees that no thread waits at the barrier
22 longer than it takes another thread to execute its final iteration, or final k iterations if
23 a chunk size of k is specified. Among such schedules, the **guided** schedule is
24 characterized by the property that it requires the fewest synchronizations. For chunk
25 size k , a typical implementation will assign $q = \text{ceiling}(n/p)$ iterations to the first
26 available thread, set n to the larger of $n-q$ and $p*k$, and repeat until all iterations are
27 assigned.

28 When the choice of the optimum schedule is not as clear as it is for these examples,
29 the **runtime** schedule is convenient for experimenting with different schedules and
30 chunk sizes without having to modify and recompile the program. It can also be
31 useful when the optimum schedule depends (in some predictable way) on the input
32 data to which the program is applied.

33 To see an example of the trade-offs between different schedules, consider sharing
34 1000 iterations among 8 threads. Suppose there is an invariant amount of work in
35 each iteration, and use that as the unit of time.

36 If all threads start at the same time, the **static** schedule will cause the construct to
37 execute in 125 units, with no synchronization. But suppose that one thread is 100
38 units late in arriving. Then the remaining seven threads wait for 100 units at the
39 barrier, and the execution time for the whole construct increases to 225.

1
2
3
4
5
6
7
8

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

Implementation-Defined Behaviors in OpenMP C/C++

This appendix summarizes the behaviors that are described as “implementation-defined” in this API. Each behavior is cross-referenced back to its description in the main specification. An implementation is required to define and document its behavior in these cases, but this list may be incomplete.

- **Number of threads:** If a parallel region is encountered while dynamic adjustment of the number of threads is disabled, and the number of threads requested for the parallel region exceeds the number that the run-time system can supply, the behavior of the program is implementation-defined (see page 9).
- **Number of processors:** The number of physical processors actually hosting the threads at any given time is implementation-defined (see page 10).
- **Creating teams of threads:** The number of threads in a team that execute a nested parallel region is implementation-defined.(see page 10).
- **schedule(runtime):** 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-defined (see page 13).
- **Default scheduling:** In the absence of the schedule clause, the default schedule is implementation-defined (see page 13).
- **ATOMIC:** It is implementation-defined whether an implementation replaces all `atomic` directives with `critical` directives that have the same unique name (see page 20).
- **omp_get_num_threads:** If the number of threads has not been explicitly set by the user, the default is implementation-defined (see page 9, and Section 3.1.2 on page 37).
- **omp_set_dynamic:** The default for dynamic thread adjustment is implementation-defined (see Section 3.1.7 on page 39).

- 1 ■ **omp_set_nested**: When nested parallelism is enabled, the number of threads
2 used to execute nested parallel regions is implementation-defined (see
3 Section 3.1.9 on page 40).
- 4 ■ **OMP_SCHEDULE** environment variable: The default value for this environment
5 variable is implementation-defined (see Section 4.1 on page 48).
- 6 ■ **OMP_NUM_THREADS** environment variable: If no value is specified for the
7 **OMP_NUM_THREADS** environment variable, or if the value specified is not a
8 positive integer, or if the value is greater than the maximum number of threads
9 the system can support, the number of threads to use is implementation-defined
10 (see Section 4.2 on page 48).
- 11 ■ **OMP_DYNAMIC** environment variable: The default value is implementation-
12 defined (see Section 4.3 on page 49).

New Features and Clarifications in Version 2.0

This appendix summarizes the key changes made to the OpenMP C/C++ specification in moving from version 1.0 to version 2.0. The following items are new features added to the specification:

- Commas are permitted in OpenMP directives (Section 2.1 on page 7).
- Addition of the **num_threads** clause. This clause allows a user to request a specific number of threads for a parallel construct (Section 2.3 on page 8).
- The **threadprivate** directive has been extended to accept static block-scope variables (Section 2.7.1 on page 23).
- C99 Variable Length Arrays are complete types, and thus can be specified anywhere complete types are allowed, for instance in the lists of **private**, **firstprivate**, and **lastprivate** clauses (Section 2.7.2 on page 25).
- A private variable in a parallel region can be marked private again in a nested directive (Section 2.7.2.1 on page 25).
- The **copyprivate** clause has been added. It provides a mechanism to use a private variable to broadcast a value from one member of a team to the other members. It is an alternative to using a shared variable for the value when providing such a shared variable would be difficult (for example, in a recursion requiring a different variable at each level). The **copyprivate** clause can only appear on the **single** directive (Section 2.7.2.8 on page 32).
- Addition of timing routines **omp_get_wtick** and **omp_get_wtime** similar to the MPI routines. These functions are necessary for performing wall clock timings (Section 3.3.1 on page 44 and Section 3.3.2 on page 45).
- An appendix with a list of implementation-defined behaviors in OpenMP C/C++ has been added. An implementation is required to define and document its behavior in these cases (Appendix E on page 97).
- The following changes serve to clarify or correct features in the previous OpenMP API specification for C/C++:

1
2
3
4
5
6
7
8

- Clarified that the behavior of `omp_set_nested` and `omp_set_dynamic` when `omp_in_parallel` returns nonzero is undefined (Section 3.1.7 on page 39, and Section 3.1.9 on page 40).
- Clarified directive nesting when nested parallel is used (Section 2.9 on page 33).
- The lock initialization and lock destruction functions can be called in a parallel region (Section 3.2.1 on page 42 and Section 3.2.2 on page 42).
- New examples have been added (Appendix A on page 51).