OpenMP C and C++ Application Program Interface

Version 2.0 March 2002

Contents

CHAPTER 1

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).

shared A shared variable names a single block of storage. All threads in a team that access this variable will access this single block of storage.

structured block A structured block is a statement (single or compound) that has a single entry and a single exit. No statement is a structured block if there is a jump into or out of that statement (including a call to **`longjmp`**(3C) or the use of **`throw`**, but a call to **`exit`** is permitted). A compound statement is a structured block if its execution always begins at the opening **`{`** and always ends at the closing **`}`**. An expression statement, selection statement, iteration statement, or **`try`** block is a structured block if the corresponding compound statement obtained by enclosing it in **`{`** and **`}`** would be a structured block. A jump statement, labeled statement, or declaration statement is not a structured block.

team One or more threads cooperating in the execution of a construct.

thread An execution entity having a serial flow of control, a set of private variables, and access to shared variables.

variable An identifier, optionally qualified by namespace names, that names an 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

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

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

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

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

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

1.4 Compliance

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

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

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

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

1.5 Normative References

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

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

1.6 Organization

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

CHAPTER 2

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
```

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

2.2 Conditional Compilation

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

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

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

2.3 `parallel` Construct

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

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

The *clause* is one of the following:

if(*scalar-expression***)**

private(*variable-list***)**

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

default(shared | none)

shared(*variable-list***)**

copyin(*variable-list***)**

reduction(*operator***:** *variable-list***)**

num_threads(*integer-expression***)**

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

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

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

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

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

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

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

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. An implementation may, for example, interrupt the execution of the program, or it may serialize the parallel region.

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

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

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

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

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

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

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

Cross References:

- **`private`**, **`firstprivate`**, **`default`**, **`shared`**, **`copyin`**, and **`reduction`** clauses, see Section 2.7.2 on page 25.
- **`OMP_NUM_THREADS`** environment variable, Section 4.2 on page 48.
- **`omp_set_dynamic`** library function, see Section 3.1.7 on page 39.
- **`OMP_DYNAMIC`** environment variable, see Section 4.3 on page 49.
- **`omp_set_nested`** function, see Section 3.1.9 on page 40.
- **`OMP_NESTED`** environment variable, see Section 4.4 on page 49.
- **`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`

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

for (*init-expr***;** *var logical-op b***;** *incr-expr***)**	
init-expr	One of the following: *var* **=** *lb* *integer-type var* **=** *lb*
incr-expr	One of the following: **++***var* *var***++** **--***var* *var***--** *var* **+=** *incr* *var* **-=** *incr* *var* **=** *var* **+** *incr* *var* **=** *incr* **+** *var* *var* **=** *var* **-** *incr*
var	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.
logical-op	One of the following: **<** **<=** **>** **>=**
lb, *b*, and *incr*	Loop invariant integer expressions. There is no synchronization during the evaluation of these expressions. Thus, any evaluated side effects produce indeterminate results.

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

The **schedule** clause specifies how iterations of the **for** loop are divided among threads of the team. The correctness of a program must not depend on which thread executes a particular iteration. The value of *chunk_size*, if specified, must be a loop invariant integer expression with a positive value. There is no synchronization during the evaluation of this expression. Thus, any evaluated side effects produce 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 *chunk_size*. The chunks are statically assigned to threads in the team in a round-robin fashion in the order of the thread number. When no *chunk_size* 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 *chunk_size* 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 *chunk_size* 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 *chunk_size* 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 *chunk_size* with value *k* greater than 1, the sizes decrease approximately exponentially to *k*, except that the last chunk may have fewer than *k* iterations. When no *chunk_size* is specified, it defaults to 1.
`runtime`	When **`schedule(runtime)`** is specified, the decision regarding scheduling is deferred until runtime. The schedule *kind* 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, *chunk_size* 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.

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

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

Cross References:

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

2.4.2 sections Construct

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

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

The clause is one of the following:

private(*variable-list***)**

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

lastprivate(*variable-list***)**

reduction(*operator***:** *variable-list***)**

nowait

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

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

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

Cross References:

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

2.4.3 `single` Construct

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

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

The clause is one of the following:

private(*variable-list***)**

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

copyprivate(*variable-list***)**

nowait

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

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

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

Cross References:

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

2.5 Combined Parallel Work-sharing Constructs

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

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

- the **parallel for** directive.
- the **parallel sections** directive.

2.5.1 `parallel for` Construct

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

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

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

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

```
/* ERROR - The barrier directive cannot be the immediate
 *         substatement of an if statement
 */
if (x!=0)
         #pragma omp barrier
...

/* OK - The barrier directive is enclosed in a
 *      compound statement.
 */
if (x!=0) {
         #pragma omp barrier
}
```

2.6.4 `atomic` Construct

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

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

The expression statement must have one of the following forms:

x binop`=` *expr*

x`++`

`++`*x*

x`--`

`--`*x*

In the preceding expressions:

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

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

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

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

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

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

Examples:

```
extern float a[], *p = a, b;
/* Protect against races among multiple updates. */
#pragma omp atomic
a[index[i]] += b;
/* Protect against races with updates through a. */
#pragma omp atomic
p[i] -= 1.0f;

extern union {int n; float x;} u;
/* ERROR - References through incompatible types. */
#pragma omp atomic
u.n++;
#pragma omp atomic
u.x -= 1.0f;
```

2.6.5 flush Directive

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

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

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

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

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

- **barrier**
- At entry to and exit from **critical**
- At entry to and exit from **ordered**
- At entry to and exit from **parallel**
- At exit from **for**
- At exit from **sections**
- At exit from **single**
- At entry to and exit from **parallel for**
- At entry to and exit from **parallel sections**

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

- At entry to **for**
- At entry to or exit from **master**
- At entry to **sections**
- At entry to **single**

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

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

```
/* ERROR - The flush directive cannot be the immediate
 *         substatement of an if statement.
 */
if (x!=0)
         #pragma omp flush (x)
...

/* OK - The flush directive is enclosed in a
 *      compound statement
 */
if (x!=0) {
         #pragma omp flush (x)
}
```

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

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

2.6.6 ordered Construct

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

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

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

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

- 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.

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.

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

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

```
int x = 1;
T a(x);
const T b_aux(x); /* Capture value of x = 1 */
T b(b_aux);
#pragma omp threadprivate(a, b)

void f(int n) {
   x++;
   #pragma omp parallel for
   /* In each thread:
    * Object a is constructed from x (with value 1 or 2?)
    * Object b is copy-constructed from b_aux
    */
   for (int i=0; i<n; i++) {
       g(a, b); /* Value of a is unspecified. */
   }
}
```

Cross References:

- Dynamic threads, see Section 3.1.7 on page 39.
- **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***`)`**

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

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

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

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

2.7.2.2 firstprivate

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

```
firstprivate(variable-list)
```

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

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

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

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

2.7.2.3 lastprivate

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

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

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

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

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

2.7.2.4 shared

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

```
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)
```

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

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

where:

x	One of the reduction variables specified in the *list*.
variable-list	A comma-separated list of scalar reduction variables.
expr	An expression with scalar type that does not reference *x*.
op	Not an overloaded operator but one of **+, *, -, &, ^, |, &&,** or **||**.
binop	Not an overloaded operator but one of **+, *, -, &, ^,** or **|**.

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

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

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

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

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

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

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

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

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

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

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

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

CHAPTER 3

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.

- the **omp_set_dynamic** function.
- the **omp_get_dynamic** function.
- the **omp_set_nested** function.
- the **omp_get_nested** function.

3.1.1 omp_set_num_threads Function

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

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

The value of the parameter *num_threads* 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_set_num_threads** function and dynamic adjustment of threads, see Section 2.3 on page 8.

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_NUM_THREADS** environment variable. The default value for the number of threads, which may be established by calling **omp_set_num_threads** or by setting the **OMP_NUM_THREADS** environment variable, can be explicitly overridden on a single **parallel** directive by specifying the **num_threads** clause.

Cross References:

- **omp_set_dynamic** function, see Section 3.1.7 on page 39.
- **omp_get_dynamic** function, see Section 3.1.8 on page 40.
- **OMP_NUM_THREADS** environment variable, see Section 4.2 on page 48, and Section 2.3 on page 8.
- **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 <= **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 long 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);
```

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

3.1.7 `omp_set_dynamic` Function

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

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

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

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

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.

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

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

Cross References:

- **`omp_get_num_threads`** function, see Section 3.1.2 on page 37.
- **`OMP_DYNAMIC`** environment variable, see Section 4.3 on page 49.
- **`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);
```

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

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

3.2.4 omp_unset_lock and omp_unset_nest_lock Functions

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

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

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

For a simple lock, the **omp_unset_lock** function releases the thread executing the function from ownership of the lock.

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

3.2.5 omp_test_lock and omp_test_nest_lock Functions

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

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

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

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

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

3.3 Timing Routines

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

- The **omp_get_wtime** function returns elapsed wall-clock time.
- The **omp_get_wtick** function returns seconds between successive clock ticks.

3.3.1 omp_get_wtime Function

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

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

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

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

The times returned are "per-thread times" by which is meant they are not required 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);
```

CHAPTER 4

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.

Example:

```
setenv OMP_NUM_THREADS 16
```

Cross References:

- **num_threads** clause, see Section 2.3 on page 8.
- **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.

4.3 OMP_DYNAMIC

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

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

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

Example:

```
setenv OMP_DYNAMIC TRUE
```

Cross References:

- For more information on parallel regions, see Section 2.3 on page 8.
- **omp_set_dynamic** function, see Section 3.1.7 on page 39.

4.4 OMP_NESTED

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

Example:

```
setenv OMP_NESTED TRUE
```

Cross Reference:

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

APPENDIX **A**

Examples

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

A.1 Executing a Simple Loop in Parallel

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

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

A.2 Specifying Conditional Compilation

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

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

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

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

A.3 Using Parallel Regions

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

```
#pragma omp parallel shared(x, npoints) private(iam, np, ipoints)
{
  iam = omp_get_thread_num();
  np =  omp_get_num_threads();
  ipoints = npoints / np;
  subdomain(x, iam, ipoints);
}
```

A.4 Using the nowait Clause

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

```
#pragma omp parallel
{
  #pragma omp for nowait
    for (i=1; i<n; i++)
      b[i] = (a[i] + a[i-1]) / 2.0;
  #pragma omp for nowait
    for (i=0; i<m; i++)
      y[i] = sqrt(z[i]);
}
```

A.5 Using the `critical` Directive

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

```
#pragma omp parallel shared(x, y) private(x_next, y_next)
{
  #pragma omp critical ( xaxis )
    x_next = dequeue(x);
  work(x_next);
  #pragma omp critical ( yaxis )
    y_next = dequeue(y);
  work(y_next);
}
```

A.6 Using the `lastprivate` Clause

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

```
#pragma omp parallel
{
  #pragma omp for lastprivate(i)
    for (i=0; i<n-1; i++)
      a[i] = b[i] + b[i+1];
}
a[i]=b[i];
```

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

A.7 Using the `reduction` Clause

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

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

A.8 Specifying Parallel Sections

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

```
#pragma omp parallel sections
{
  #pragma omp section
    xaxis();
  #pragma omp section
    yaxis();
  #pragma omp section
    zaxis();
}
```

A.9 Using `single` Directives

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

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

```
#pragma omp parallel
{
  #pragma omp single
    printf("Beginning work1.\n");
  work1();
  #pragma omp single
    printf("Finishing work1.\n");
  #pragma omp single nowait
    printf("Finished work1 and beginning work2.\n");
  work2();
}
```

A.10 Specifying Sequential Ordering

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

```
#pragma omp for ordered schedule(dynamic)
  for (i=lb; i<ub; i+=st)
    work(i);

void work(int k)
{
  #pragma omp ordered
    printf(" %d", k);
}
```

A.11 Specifying a Fixed Number of Threads

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

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

```
omp_set_dynamic(0);
omp_set_num_threads(16);
#pragma omp parallel shared(x, npoints) private(iam, ipoints)
{
  if (omp_get_num_threads() != 16) abort();
  iam = omp_get_thread_num();
  ipoints = npoints/16;
  do_by_16(x, iam, ipoints);
}
```

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

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

A.12 Using the atomic Directive

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

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

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

Note that the **atomic** directive applies only to the C or C++ statement immediately 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:

```
int   sync[NUMBER_OF_THREADS];
float work[NUMBER_OF_THREADS];
#pragma omp parallel private(iam,neighbor) shared(work,sync)
{

  iam = omp_get_thread_num();
  sync[iam] = 0;
  #pragma omp barrier

  /*Do computation into my portion of work array */
  work[iam] = ...;

  /*  Announce that I am done with my work
   *  The first flush ensures that my work is
   *  made visible before sync.
   *  The second flush ensures that sync is made visible.
   */
  #pragma omp flush(work)
  sync[iam] = 1;
  #pragma omp flush(sync)

  /*Wait for neighbor*/
  neighbor = (iam>0 ? iam : omp_get_num_threads()) - 1;
  while (sync[neighbor]==0) {
    #pragma omp flush(sync)
  }

  /*Read neighbor's values of work array */
  ... = work[neighbor];
}
```

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:

```
int x, *p = &x;

void f1(int *q)
{
  *q = 1;
  #pragma omp flush
  // x, p, and *q are flushed
  //   because they are shared and accessible
  // q is not flushed because it is not shared.
}

void f2(int *q)
{
  #pragma omp barrier
  *q = 2;
  #pragma omp barrier
  // a barrier implies a flush
  // x, p, and *q are flushed
  //   because they are shared and accessible
  // q is not flushed because it is not shared.
}

int g(int n)
{
  int i = 1, j, sum = 0;
  *p = 1;
  #pragma omp parallel reduction(+: sum) num_threads(10)
  {
    f1(&j);
    // i, n and sum were not flushed
    //   because they were not accessible in f1
    // j was flushed because it was accessible
    sum += j;
    f2(&j);
    // i, n, and sum were not flushed
    //   because they were not accessible in f2
    // j was flushed because it was accessible
    sum += i + j + *p + n;
  }
  return sum;
}
```

A.15 Determining the Number of Threads Used

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

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

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

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

```
#pragma omp parallel private(i)
{
  i = omp_get_thread_num();
  work(i);
}
```

A.16 Using Locks

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

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

```
#include <omp.h>
int main()
{
  omp_lock_t lck;
  int id;

  omp_init_lock(&lck);
  #pragma omp parallel shared(lck) private(id)
  {
    id = omp_get_thread_num();

    omp_set_lock(&lck);
    printf("My thread id is %d.\n", id);
// only one thread at a time can execute this printf
    omp_unset_lock(&lck);

    while (! omp_test_lock(&lck)) {
      skip(id);   /* we do not yet have the lock,
                     so we must do something else */
    }
    work(id);      /* we now have the lock
                      and can do the work */
    omp_unset_lock(&lck);
  }
  omp_destroy_lock(&lck);
}
```

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.

```
#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);
}
```

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

```
void wrong3(int n)
{
  #pragma omp parallel default(shared)
  {
    int i;
    #pragma omp for
      for (i=0; i<n; i++) {
        #pragma omp single
          work(i);
      }
  }
}
```

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

```
void wrong4(int n)
{
  #pragma omp parallel default(shared)
  {
    int i;
    #pragma omp for
      for (i=0; i<n; i++) {
        work1(i);
        #pragma omp barrier
        work2(i);
      }
  }
}
```

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

```
void wrong5()
{
  #pragma omp parallel
  {
    #pragma omp critical
    {
       work1();
       #pragma omp barrier
       work2();
    }
  }
}
```

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

```
void wrong6()
{
  #pragma omp parallel
  {
    setup();
    #pragma omp single
    {
      work1();
      #pragma omp barrier
      work2();
    }
    finish();
  }
}
```

A.20 Binding of `barrier` Directives

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

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

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

```
int main()
{
  sub1(2);
  sub2(2);
  sub3(2);
}

void sub1(int n)
{
  int i;
  #pragma omp parallel private(i) shared(n)
  {
    #pragma omp for
    for (i=0; i<n; i++)
      sub2(i);
  }
}

void sub2(int k)
{
  #pragma omp parallel shared(k)
    sub3(k);
}

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

A.21 Scoping Variables with the `private` Clause

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

```
int i, j;
i = 1;
j = 2;
#pragma omp parallel private(i) firstprivate(j)
{
  i = 3;
  j = j + 2;
}
printf("%d %d\n", i, j);
```

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)

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

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

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

```
#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.

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

```
float read_next( ) {
float * tmp;
float return_val;

      #pragma omp single copyprivate(tmp)
      {
        tmp = (float *) malloc(sizeof(float));
      }

      #pragma omp master
      {
        get_float( tmp );
      }

      #pragma omp barrier
      return_val = *tmp;
      #pragma omp barrier

      #pragma omp single
      {
       free(tmp);
      }

return return_val;
}
```

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

```
#include <omp.h>

omp_lock_t *new_lock()
{
omp_lock_t *lock_ptr;

     #pragma omp single copyprivate(lock_ptr)
     {
      lock_ptr = (omp_lock_t *) malloc(sizeof(omp_lock_t));
      omp_init_lock( lock_ptr );
     }

return lock_ptr;
}
```

A.26 Using the `threadprivate` Directive

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

Example 1:

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

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

Example 2:

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

A.27 Use of C99 Variable Length Arrays

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

```
void f(int m, int C[m][m])
{
  double v1[m];
  ...
  #pragma omp parallel firstprivate(C, v1)
  ...
}
```

A.28 Use of num_threads Clause

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

```
#include <omp.h>
main()
{
  omp_set_dynamic(1);
  ...
  #pragma omp parallel num_threads(10)
  {
     ... parallel region ...
  }
}
```

A.29 Use of Work-Sharing Constructs Inside a `critical` Construct

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

```
void f()
{
  int i = 1;
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      #pragma omp critical (name)
      {
        #pragma omp parallel
        {
          #pragma omp single
          {
            i++;
          }
        }
      }
    }
  }
}
```

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;
}
```

APPENDIX **B**

Stubs for Run-time Library Functions

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

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

```
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"
#ifdef __cplusplus
extern “C” {
#endif

void omp_set_num_threads(int num_threads)
{
}
int omp_get_num_threads(void)
{
  return 1;
}
int omp_get_max_threads(void)
{
  return 1;
}
int omp_get_thread_num(void)
{
  return 0;
}
int omp_get_num_procs(void)
{
  return 1;
}
void omp_set_dynamic(int dynamic_threads)
{
}
int omp_get_dynamic(void)
{
  return 0;
}
int omp_in_parallel(void)
{
  return 0;
}
void omp_set_nested(int nested)
{
}
```

```
int omp_get_nested(void)
{
  return 0;
}

enum {UNLOCKED = -1, INIT, LOCKED};

void omp_init_lock(omp_lock_t *lock)
{
  *lock = UNLOCKED;
}

void omp_destroy_lock(omp_lock_t *lock)
{
  *lock = INIT;
}

void omp_set_lock(omp_lock_t *lock)
{
  if (*lock == UNLOCKED) {
    *lock = LOCKED;
  } else if (*lock == LOCKED) {
    fprintf(stderr, "error: deadlock in using lock variable\n");
    exit(1);
  } else {
    fprintf(stderr, "error: lock not initialized\n");
    exit(1);
  }
}


void omp_unset_lock(omp_lock_t *lock)
{
  if (*lock == LOCKED) {
    *lock = UNLOCKED;
  } else if (*lock == UNLOCKED) {
    fprintf(stderr, "error: lock not set\n");
    exit(1);
  } else {
    fprintf(stderr, "error: lock not initialized\n");
    exit(1);
  }
}
```

```
int omp_test_lock(omp_lock_t *lock)
{
  if (*lock == UNLOCKED) {
    *lock = LOCKED;
    return 1;
  } else if (*lock == LOCKED) {
    return 0;
  } else {
    fprintf(stderr, "error: lock not initialized\n");
    exit(1);
  }
}

#ifndef OMP_NEST_LOCK_T
typedef struct {  /* This really belongs in omp.h */
    int owner;
    int count;
} omp_nest_lock_t;
#endif

enum {MASTER = 0};

void omp_init_nest_lock(omp_nest_lock_t *lock)
{
  lock->owner = UNLOCKED;
  lock->count = 0;
}

void omp_destroy_nest_lock(omp_nest_lock_t *lock)
{
  lock->owner = UNLOCKED;
  lock->count = UNLOCKED;
}


void omp_set_nest_lock(omp_nest_lock_t *lock)
{
  if (lock->owner == MASTER && lock->count >= 1) {
    lock->count++;
  } else if (lock->owner == UNLOCKED && lock->count == 0) {
    lock->owner = MASTER;
    lock->count = 1;
  } else {
   fprintf(stderr, "error: lock corrupted or not initialized\n");
    exit(1);
  }
}
```

```
void omp_unset_nest_lock(omp_nest_lock_t *lock)
{
  if (lock->owner == MASTER && lock->count >= 1) {
    lock->count--;
    if (lock->count == 0) {
      lock->owner = UNLOCKED;
    }
  } else if (lock->owner == UNLOCKED && lock->count == 0) {
    fprintf(stderr, "error: lock not set\n");
    exit(1);
  } else {
   fprintf(stderr, "error: lock corrupted or not initialized\n");
    exit(1);
  }
}

int omp_test_nest_lock(omp_nest_lock_t *lock)
{
  omp_set_nest_lock(lock);
  return lock->count;
}

double omp_get_wtime(void)
{
/* This function does not provide a working
   wallclock timer. Replace it with a version
   customized for the target machine.
*/
   return 0.0;
}

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

#ifdef __cplusplus
}
#endif
```

APPENDIX C

OpenMP C and C++ Grammar

C.1 Notation

The grammar rules consist of the name for a non-terminal, followed by a colon, followed by replacement alternatives on separate lines.

The syntactic expression $term_{opt}$ indicates that the term is optional within the replacement.

The syntactic expression $term_{optseq}$ is equivalent to $term\text{-}seq_{opt}$ with the following additional rules:

term-seq :

term

term-seq term

term-seq , term

C.2 Rules

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

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

statement-seq:

statement

openmp-directive

statement-seq statement

statement-seq openmp-directive

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

statement-list:

statement

openmp-directive

statement-list statement

statement-list openmp-directive

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

block-item:

declaration

statement

openmp-directive

statement:

/* standard statements */

openmp-construct

openmp-construct:

parallel-construct

for-construct

sections-construct

single-construct

parallel-for-construct

parallel-sections-construct

master-construct

critical-construct

atomic-construct

ordered-construct

openmp-directive:

barrier-directive

flush-directive

structured-block:

statement

parallel-construct:

parallel-directive structured-block

parallel-directive:

 # pragma omp parallel $parallel\text{-}clause_{optseq}$ *new-line*

parallel-clause:

unique-parallel-clause

data-clause

unique-parallel-clause:

if (*expression* **)**

num_threads (expression **)**

for-construct:

for-directive iteration-statement

for-directive:

 # pragma omp for *for-clause*$_{optseq}$ *new-line*

for-clause:

unique-for-clause

data-clause

nowait

unique-for-clause:

ordered

schedule (*schedule-kind* **)**

schedule (*schedule-kind* **,** *expression* **)**

schedule-kind:

static

dynamic

guided

runtime

sections-construct:

sections-directive section-scope

sections-directive:

 # pragma omp sections *sections-clause*$_{optseq}$ *new-line*

sections-clause:

data-clause

nowait

section-scope:

{ section-sequence }

section-sequence:

section-directive$_{opt}$ *structured-block*

section-sequence section-directive structured-block

section-directive:

 # pragma omp section *new-line*

single-construct:

single-directive structured-block

single-directive:

 # pragma omp single *single-clause*$_{optseq}$ *new-line*

single-clause:

data-clause

nowait

parallel-for-construct:

parallel-for-directive iteration-statement

parallel-for-directive:

 # pragma omp parallel for *parallel-for-clause*$_{optseq}$ *new-line*

parallel-for-clause:

unique-parallel-clause

unique-for-clause

data-clause

parallel-sections-construct:

parallel-sections-directive section-scope

parallel-sections-directive:

 # pragma omp parallel sections *parallel-sections-clause*$_{optseq}$ *new-line*

parallel-sections-clause:

unique-parallel-clause

data-clause

master-construct:

master-directive structured-block

master-directive:

 # pragma omp master *new-line*

critical-construct:

critical-directive structured-block

critical-directive:

 # pragma omp critical *region-phrase*$_{opt}$ *new-line*

region-phrase:

(identifier)

barrier-directive:

 # pragma omp barrier *new-line*

atomic-construct:

atomic-directive expression-statement

atomic-directive:

 # pragma omp atomic *new-line*

flush-directive:

 # pragma omp flush *flush-vars*$_{opt}$ *new-line*

flush-vars:

(variable-list)

ordered-construct:

ordered-directive structured-block

ordered-directive:

 # pragma omp ordered *new-line*

declaration:

/* standard declarations */

threadprivate-directive

threadprivate-directive:

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

data-clause:

private (*variable-list* **)**

copyprivate (*variable-list* **)**

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

lastprivate (*variable-list* **)**

shared (*variable-list* **)**

default (shared)

default (none)

reduction (*reduction-operator* **:** *variable-list* **)**

copyin (*variable-list* **)**

reduction-operator:

One of: **+ * - & ^ | && ||**

/* in C */

variable-list:

identifier

variable-list **,** *identifier*

/* in C++ */

variable-list:

id-expression

variable-list **,** *id-expression*

APPENDIX **D**

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

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

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

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

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

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

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

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

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

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

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

```
#pragma omp parallel
{
  #pragma omp sections nowait
  {
    // ...
  }
  #pragma omp for schedule(guided)
  for(i=0; i<n; i++) {
      invariant_amount_of_work(i);
  }
}
```

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

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

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

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

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.

APPENDIX **E**

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).

- **omp_set_nested**: When nested parallelism is enabled, the number of threads used to execute nested parallel regions is implementation-defined (see Section 3.1.9 on page 40).
- **OMP_SCHEDULE** environment variable: The default value for this environment variable is implementation-defined (see Section 4.1 on page 48).
- **OMP_NUM_THREADS** environment variable: 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 (see Section 4.2 on page 48).
- **OMP_DYNAMIC** environment variable: The default value is implementation-defined (see Section 4.3 on page 49).

APPENDIX **F**

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++:

- 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).