# OpenMP C and C++ Application Program Interface

## Version 1.0 – October 1998

Document Number 004–2229–001

# Contents

*Page*

*Page*

# Introduction [1]

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 is not covered in this specification.

## 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 `#pragma` directives and implicit barriers created by the implementation. |
| *construct* | A construct is a statement. It consists of an OpenMP `#pragma` directive and the subsequent structured block. Note that some directives are not part of a construct. (See *openmp-directive* in Appendix C, page 69). |
| *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 executed by multiple *threads* in parallel. |
| *private* | Accessible to only one thread in the team for a parallel region. Note that there are several ways to specify that a variable is private: a definition within the parallel region, a `threadprivate` directive, a `private`, `firstprivate`, `lastprivate`, or `reduction` clause, or use of the variable as a loop control variable. |
| *serial region* | Statements executed only by the *master thread* outside of a *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* | Accessible to all threads in the team for a parallel region. |
| *structured block* | A structured block is a statement 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 (this is an "as if" rule). A jump statement, labeled statement, or declaration statement is not a structured block. |
| *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. This model allows programs to be developed that exploit multiple threads of execution. It is permitted to develop a program that always executes on more than one thread (i.e., does not behave correctly when executed sequentially). It is possible to write a program that gives the expected behavior (allowing for numerical differences) when executed as an OpenMP program and as a sequential program (i.e., ignoring directives).

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 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, 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 thread, 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 the parallel construct but 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 non-deterministic order. Similarly, unsynchronized calls to input functions that read from the same file may read data in non-deterministic 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 2, 3, 4, and Appendix C. Appendix A, B, and D 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.)

## 1.5 Organization

This document is organized as follows:

- Directives (see Chapter 2, page 7).
- Run-time library functions (see Chapter 3, page 33).
- Environment variables (see Chapter 4, page 43).
- Examples (see Appendix A, page 47).

# Directives [2]

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.

This section addresses the following topics:

- Directive format (see Section 2.1, page 7).
- Conditional compilation (see Section 2.2, page 8).
- The parallel construct (see Section 2.3, page 8).
- Work-sharing constructs (see Section 2.4, page 10).
- Combined parallel work-sharing constructs (see Section 2.5, page 15).
- Master and synchronization constructs (see Section 2.6, page 16).
- Data environment constructs, including one directive and several clauses that affect the data environment (see Section 2.7, page 22).
- Directive binding (see Section 2.8, page 30).
- Directive nesting (see Section 2.9, page 31).

## 2.1 Directive Format

The syntax of an OpenMP directive is formally specified by the grammar in Appendix C, page 69, 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 a *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(`*list*`)`

`firstprivate(`*list*`)`

```
default(shared | none)
shared(list)
copyin(list)
reduction(operator: list)
```

For information on the `private`, `firstprivate`, `default`, `shared`, `copyin`, and `reduction` clauses, see Section 2.7.2, page 24.

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 non-zero value.

This thread becomes the master thread of the team, with a thread number of 0, and all threads, including the master thread, execute the region in parallel. The number of threads in the team is controlled by environment variables and/or library calls (see Section 4.2, page 44 and Chapter 3, page 33). If the value of the `if` expression is zero, the region is serialized.

Once created, the runtime system must not change the number of threads in the team, and it remains constant while that parallel region is being executed. It can be changed either explicitly by the user or automatically by the runtime system from one parallel region to another. The `omp_set_dynamic` library function and the `OMP_DYNAMIC` environment variable can be used to enable and disable the automatic adjustment of the number of threads. For more information on the `omp_set_dynamic` library function, see Section 3.1.7, page 36. For more information on the `OMP_DYNAMIC` environment variable, see Section 4.3, page 44.

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 static extent of a parallel region but inside the dynamic extent are referred to as orphaned directives.

There is an implied barrier at the end of a parallel region. 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` (see Section 3.1.9, page 37) or the environment variable `OMP_NESTED` (see Section 4.4, page 44). An implementation is allowed to always create a team composed of one thread when a nested parallel region is encountered, regardless of the specified default behavior.

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 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.

## 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:

- `for` directive (see Section 2.4.1, page 10).
- `sections` directive (see Section 2.4.2, page 14).
- `single` directive (see Section 2.4.3, page 15).

### *2.4.1* `for` *Construct*

The `for` directive identifies an iterative work-sharing construct that specifies a region in which the iterations of the associated loop should be executed in parallel. The iterations of the `for` loop are distributed across threads that already exist. 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(`*list*`)`

`firstprivate(`*list*`)`

`lastprivate(`*list*`)`

`reduction(`*operator*`:` *list*`)`

`ordered`

`schedule(`*kind*`[,` *chunk_size*`])`

`nowait`

For information on the `private`, `firstprivate`, `lastprivate`, and `reduction` clauses, see Section 2.7.2, page 24.

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:<br>`var = lb`<br>*integer-type* `var = lb` |
| *incr-expr* | One of the following:<br>`++var`<br>`var++`<br>`--var`<br>`var--`<br>`var += incr`<br>`var -= incr`<br>`var = var + incr`<br>`var = incr + var`<br>`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:<br>`<`<br>`<=`<br>`>`<br>`>=` |

`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.

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:

`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, a chunk of *chunk_size* iterations are assigned to each thread. When a thread finishes its assigned chunk of iterations, it is dynamically assigned another chunk, until none remain. 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, then 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 exponentially to 1. For a *chunk_size* with value *k* greater than 1, the sizes decrease 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` (see Section 4.1, page 43). If this environment variable is not set, the resulting schedule is implementation-dependent. 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-dependent.

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. For more information, see Appendix D, page 75.

The `ordered` clause must be present when `ordered` directives (see Section 2.6.6, page 21) are contained in the dynamic extent of 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.

### *2.4.2* `sections` *Construct*

The `sections` directive identifies a non-iterative 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(`*list*`)`

`firstprivate(`*list*`)`

`lastprivate(`*list*`)`

`reduction(`*operator*`:` *list*`)`

`nowait`

For information on the `private`, `firstprivate`, `lastprivate`, and `reduction` clauses, see Section 2.7.2, page 24.

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.

### 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(`*list*`)`

`firstprivate(`*list*`)`

`nowait`

For information on the `private` and `firstprivate` clauses, see Section 2.7.2, page 24.

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.

## 2.5 Combined Parallel Work-sharing Constructs

Combined parallel work–sharing constructs are short cuts 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:

- Section 2.5.1, page 15, describes the `parallel for` directive.
- Section 2.5.2, page 16, describes the `parallel sections` directive.

### 2.5.1 `parallel for` *Construct*

The `parallel for` directive is a shortcut for a `parallel` region that contains 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 admits all the clauses of the `parallel` directive (see Section 2.3, page 8) and the `for` directive (see Section 2.4.1, page 10), except the `nowait` clause, with identical meanings and restrictions (see Section 2.7.2, page 24). The semantics are identical to explicitly specifying a `parallel` directive immediately followed by a `for` directive.

### *2.5.2* `parallel sections` *Construct*

The `parallel sections` directive provides a shortcut form for specifying a `parallel` region containing 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 *clause*s accepted by the `parallel` and `sections` directives, except the `nowait` clause. For more information on the `parallel` directive, see Section 2.3, page 8. For more information on the `sections` directive, see Section 2.4.2, page 14.

## 2.6 Master and Synchronization Constructs

The following sections describe the synchronization constructs:

- Section 2.6.1, page 17, describes the `master` directive.
- Section 2.6.2, page 17, describes the `critical` directive.

- Section 2.6.3, page 18, describes the `barrier` directive.
- Section 2.6.4, page 18, describes the `atomic` directive.
- Section 2.6.5, page 20, describes the `flush` directive.
- Section 2.6.6, page 21, describes the `ordered` directive.

### *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 statement. There is no implied barrier either on entry to or exit from the master section.

### *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 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 have encountered the barrier, each thread begins executing the statements after the `barrier` directive in parallel.

Note that the `barrier` directive is not a statement (in the formal syntax), and that the smallest statement that contains a `barrier` directive must be a block (or compound-statement).

```
/* ERROR - the smallest statement containing the
 *         barrier directive is the if statement
 *         that begins on the preceding line
 */
if (x!=0)
  #pragma barrier
...

/* OK - the block within the if statement
 *      is the smallest statement
 *      containing the barrier directive
 */
if (x!=0) {
  #pragma barrier
}
```

Restrictions to the `barrier` directive are as follows:

- The smallest statement that contains a `barrier` directive must be a block (or compound-statement).

### 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 one of +, *, -, /, &, ^, |, <<, or >>.

Although a conforming implementation can replace 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 and reload their values from memory.

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

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

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

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

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

The directive is not implied if a `nowait` clause is present.

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 the `flush` directive is not a statement (in the formal syntax), and that the smallest statement that contains a `flush` directive must be a block (or compound-statement).

```
/* ERROR - the smallest statement containing the
 *         flush directive is the if statement
 *         that begins on the preceding line
 */
if (x!=0)
  #pragma flush (x)
...

/* OK - the block within the if statement
 *      is the smallest statement
 *      containing the flush directive
 */
if (x!=0) {
  #pragma flush (x)
}
```

Restrictions to the `flush` directive are as follows:

- A variable specified in a `flush` directive must not have a reference type.
- The smallest statement that contains a `flush` directive must be a block (or compound-statement).

### *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 that has an `ordered` clause. 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 `ordered` directive must not be in the dynamic extent of a `for` directive that does not have the `ordered` clause specified.
- 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 variables local to a thread.
- Clauses that may be specified on the directives to control the scope attributes of variables for the duration of the parallel or work-sharing constructs are described in Section 2.7.2, page 24.

### *2.7.1* `threadprivate` *Directive*

The `threadprivate` directive makes the named file-scope or namespace-scope variables specified in the *list* private to a thread but file-scope visible within the thread. *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(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 a 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. For more information on dynamic threads, see the `omp_set_dynamic` library function, Section 3.1.7, page 36, and the `OMP_DYNAMIC` environment variable, Section 4.3, page 44.

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

- A `threadprivate` directive must appear at file scope or namespace scope, must appear outside of any definition or declaration, and must lexically precede all references to any of the variables in its *list*.
- Each variable in the *list* of a `threadprivate` directive must have a file-scope or namespace-scope declaration that lexically precedes the directive.
- 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 other than the `copyin`, `schedule`, or the `if` clause. As a result, they are not permitted in `private`, `firstprivate`, `lastprivate`, `shared`, or `reduction` clauses. They are not affected by the `default` 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 (in case the initialization is implemented using a temporary shared object, as described above).

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. */
   }
}
```

### 2.7.2 Data Scope Attribute Clauses

Several directives accept clauses that allow a user to control the scope attributes of variables for the duration of the region. Scope 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 scope 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 *list* argument, which is a comma-separated list of variables that are visible. If a variable referenced in a data scope 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 scope attribute clauses:

- Section 2.7.2.1, page 25, describes the `private` clause.
- Section 2.7.2.2, page 25, describes the `firstprivate` clause.
- Section 2.7.2.3, page 26, describes the `lastprivate` clause.
- Section 2.7.2.4, page 26, describes the `shared` clause.
- Section 2.7.2.5, page 27, describes the `default` clause.

- Section 2.7.2.6, page 27, describes the `reduction` clause.
- Section 2.7.2.7, page 30, describes the `copyin` clause.

### *2.7.2.1* `private`

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

```
private(list)
```

The behavior of a variable specified in a `private` clause is as follows. A new object with automatic storage duration is allocated within the associated structured block (thus, its visibility is defined by that block). 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.
- Unless it has a class type with a `mutable` member, a variable specified in a `private` clause must not have a const-qualified type.
- A variable specified in a `private` clause must not have an incomplete type or a reference type.
- Variables that are private within a parallel region cannot be specified in a `private` clause on an enclosed work-sharing or `parallel` directive. As a result, variables that are specified `private` on a work-sharing or `parallel` directive must be shared in the enclosing parallel region.

### *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(list)
```

Variables specified in the *list* have `private` clause semantics described in the preceding section, except that each new private object is initialized as if there were an implied declaration inside the structured block, and the initializer is the value of the variable's original object. A copy constructor is invoked for a class object if necessary.

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

- All restrictions for `private` apply, except for the restrictions about default constructors and about const-qualified types.
- A variable with a class type that is specified as `firstprivate` must have an accessible, unambiguous copy constructor.

### *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(list)
```

Variables specified in the *list* have `private` clause semantics, described in Section 2.7.2.1, page 25. 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 that is specified as `lastprivate` must have an accessible, unambiguous copy assignment operator.

### *2.7.2.4* `shared`

This clause shares variables that appear in the *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(list)
```

#### 2.7.2.5 default

The `default` clause allows the user to affect the data scope 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 each currently visible variable that is referenced in the lexical extent of the parallel construct must be explicitly listed in a data scope attribute clause, unless it is `threadprivate`, const-qualified, specified in an enclosed data scope attribute clause, or used as a loop control variable referenced only in a corresponding `for` or `parallel for`.

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

Variables may be exceptioned from a defined default 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 *list*, with the operator *op*. The syntax of the `reduction` clause is as follows:

```
reduction(op:list)
```

A `reduction` is typically used in 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
```

| | |
|---|---|
| x | One of the reduction variables specified in the list. |
| *list* | A comma-separated list of 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.

Variables that appear in the `reduction` clause must be shared in the enclosing context.

```
#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)
```

A private copy of each variable in *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 shared variable is updated to reflect the result of combining the original value of the (shared) reduction variable 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 shared variable 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 `reduction` construct; however, if the `reduction` clause is used on a construct to which `nowait` is also applied, the value of the shared variable 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.
- A variable that is specified in the `reduction` clause must be shared in the enclosing context.

#### *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(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.

## 2.8 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. If no parallel region is currently being executed, the directives have no effect.
- 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 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.
- `for`, `sections`, and `single` directives are not permitted in the dynamic extent of `critical`, `ordered`, and `master` regions.
- `barrier` directives are not permitted in the dynamic extent of `for`, `ordered`, `sections`, `single`, `master`, and `critical` regions.
- `master` directives are not permitted in the dynamic extent of `for`, `sections`, and `single` directives.
- `ordered` directives are not allowed in the dynamic extent of `critical` 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 with respect to a team composed of only the master thread.

# Runtime Library Functions [3]

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.

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

- Execution environment functions (see Section 3.1, page 33).
- Lock functions (see Section 3.2, page 38).

## 3.1 Execution Environment Functions

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

- Section 3.1.1, page 34, describes the `omp_set_num_threads` function.
- Section 3.1.2, page 34, describes the `omp_get_num_threads` function.
- Section 3.1.3, page 35, describes the `omp_get_max_threads` function.
- Section 3.1.4, page 35, describes the `omp_get_thread_num` function.
- Section 3.1.5, page 35, describes the `omp_get_num_procs` function.
- Section 3.1.6, page 36, describes the `omp_in_parallel` function.
- Section 3.1.7, page 36, describes the `omp_set_dynamic` function.
- Section 3.1.8, page 37, describes the `omp_get_dynamic` function.
- Section 3.1.9, page 37, describes the `omp_set_nested` function.
- Section 3.1.10, page 37, describes the `omp_get_nested` function.

### *3.1.1* `omp_set_num_threads` *Function*

The `omp_set_num_threads` function sets the number of threads to use for subsequent parallel regions. 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 positive. Its effect depends upon whether dynamic adjustment of the number of threads is enabled. If dynamic adjustment is disabled, the value is used as the number of threads for all subsequent parallel regions prior to the next call to this function; otherwise, the value is the maximum number of threads that will be used. This function has effect only when called from serial portions of the program. If it is called from a portion of the program where the `omp_in_parallel` function returns non-zero, the behavior of this function is undefined. For more information on this subject, see the `omp_set_dynamic` function, Section 3.1.7, page 36, and the `omp_get_dynamic` function, Section 3.1.8, page 37.

This call has precedence over the `OMP_NUM_THREADS` environment variable (see Section 4.2, page 44).

### *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 `omp_set_num_threads` function (see the preceding section) and the `OMP_NUM_THREADS` environment variable (see Section 4.2, page 44) 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 dependent. This function binds to the closest enclosing `parallel` directive (see Section 2.3, page 8). If called from a serial portion of a program, or from a nested parallel region that is serialized, this function returns 1.

### *3.1.3* `omp_get_max_threads` *Function*

The `omp_get_max_threads` function returns the maximum value that can be returned by calls to `omp_get_num_threads`. (For more information on `omp_get_num_threads`, see Section 3.1.2, page 34.) The format is as follows:

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

If `omp_set_num_threads` (see Section 3.1.1, page 34) is used to change the number of threads, subsequent calls to this function will return the new value. A typical use of this function is to determine the size of an array for which all thread numbers are valid indices, even when `omp_set_dynamic` (see Section 3.1.7, page 36) is set to non-zero.

This function returns the maximum value whether executing within a serial region or a parallel region.

### *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`.

### *3.1.5* `omp_get_num_procs` *Function*

The `omp_get_num_procs` function returns the maximum number of processors that could be assigned to the program. 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 non-zero 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 non–zero from within a region executing in parallel, regardless of 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);
```

This function has effect only when called from serial portions of the program. If it is called from a portion of the program where the `omp_in_parallel` function returns non-zero, the behavior of the function is undefined. If `dynamic_threads` evaluates to non-zero, 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 always remains fixed over the duration of each parallel region and is reported by the `omp_get_num_threads` function (see Section 3.1.2, page 34).

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

A call to `omp_set_dynamic` has precedence over the `OMP_DYNAMIC` environment variable (see Section 4.3, page 44).

The default for the dynamic adjustment of threads is implementation dependent. 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.

### *3.1.8* `omp_get_dynamic` *Function*

The `omp_get_dynamic` function returns non-zero if dynamic thread adjustments enabled and returns `0` otherwise. For a description of dynamic thread adjustment, see Section 3.1.7, page 36. 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`.

### *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`, which is the default, nested parallelism is disabled, and nested parallel regions are serialized and executed by the current thread. If `nested` evaluates to non-zero, nested parallelism is enabled, and parallel regions that are nested may deploy additional threads to form the team.

This call has precedence over the `OMP_NESTED` environment variable (see Section 4.4, page 44).

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

### *3.1.10* `omp_get_nested` *Function*

The `omp_get_nested` function returns non-zero if nested parallelism is enabled and `0` if it is disabled. For more information on nested parallelism, see the preceding section. 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 (see Section 3.2.1, page 39).
- The `omp_destroy_lock` function removes a simple lock (see Section 3.2.2, page 39).
- The `omp_set_lock` function waits until a simple lock is available (see Section 3.2.3, page 39).
- The `omp_unset_lock` function releases a simple lock (see Section 3.2.4, page 40).
- The `omp_test_lock` function tests a simple lock (see Section 3.2.5, page 40).

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 (see Section 3.2.1, page 39).
- The `omp_destroy_nest_lock` function removes a nestable lock (see Section 3.2.2, page 39).
- The `omp_set_nest_lock` function waits until a nestable lock is available (see Section 3.2.3, page 39).
- The `omp_unset_nest_lock` function releases a nestable lock (see Section 3.2.4, page 40).
- The `omp_test_nest_lock` function tests a nestable lock (see Section 3.2.5, page 40).

### *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.

### *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);
```

The argument to these functions must point to an initialized lock variable that is 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 non-zero 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.

# Environment Variables [4]

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 not case sensitive. 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 (see Section 4.1, page 43).
- `OMP_NUM_THREADS` sets the number of threads to use during execution (see Section 4.2, page 44).
- `OMP_DYNAMIC` enables or disables dynamic adjustment of the number of threads (see Section 4.3, page 44).
- `OMP_NESTED` enables or disables nested parallelism (see Section 4.4, page 44).

## 4.1 OMP_SCHEDULE

`OMP_SCHEDULE` applies only to `for` (see Section 2.4.1, page 10) and `parallel for` (see Section 2.5.1, page 15) 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 dependent. 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"
```

## 4.2 OMP_NUM_THREADS

The value of the OMP_NUM_THREADS environment variable must be positive. Its effect depends upon whether dynamic adjustment of the number of threads is enabled. If dynamic adjustment is disabled, the value of this environment variable is the number of threads to use for each parallel region, until that number is explicitly changed during execution by calling omp_set_num_threads() (see Section 3.1.1, page 34).

If dynamic adjustment of the number of threads is enabled, the value of this environment variable is interpreted as the maximum number of threads to use.

The default value is implementation dependent.

Example:

```
setenv OMP_NUM_THREADS 16
```

## 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. Its value must be TRUE or FALSE. (For more information on parallel regions, see Section 2.3, page 8.)

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 dependent. (For more information, see Section 3.1.7, page 36.)

Example:

```
setenv OMP_DYNAMIC TRUE
```

## 4.4 OMP_NESTED

The OMP_NESTED environment variable enables or disables nested parallelism. If set to TRUE, nested parallelism is enabled; if it is set to FALSE, nested parallelism is disabled. The default value is FALSE. (For more information, see Section 3.1.9, page 37).

Example:

```
setenv OMP_NESTED TRUE
```

# Examples [A]

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 shows how to parallelize a simple loop. 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. With OpenMP compilation, the macro `_OPENMP` is 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 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 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. 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 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 shows how to use the `reduction` clause:

```
#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, 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

In the following 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 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-dependent, 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:

```
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.

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:

```
#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 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:

```
#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 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
}

void f2(int *q)
{
  *q = 2;
  #pragma omp barrier
  // a barrier implies a flush
  // x, p, and *q are flushed
  //   because they are shared and accessible
}

int g(int n)
{
  int i = 1, j, sum = 0;
  *p = 1;
  #pragma omp parallel reduction(+: sum)
  {
    f1(&j);
```

```
    // i and n were not flushed
    //   because they were not accessible in f1
    // j was flushed because it was accessible
    sum += j;
    f2(&j);
    // i and n 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:

```
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, 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.

```
#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);
    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 shows 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 program is correct 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 correct:

```
#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, page 31.

The following example is incorrect 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 incorrect:

```
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 incorrect 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 incorrect 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 incorrect 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 incorrect 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, page 30.

In the following example, the calls in `main`, to `sub1` and `sub2`, are both valid, and the `barrier` in `sub3` binds to the `parallel` region in `sub2` in both cases. The effect is different, however, because in the call to `sub1`, the `barrier` affects only a subteam. The number of threads in a subteam is implementation-dependent if nested parallelism is enabled (with the `OMP_NESTED` environment variable), and otherwise is one (in which case the `barrier` has no real effect).

```
int main()
{
  sub1(2);
  sub2(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)
{
  work1(n);
  #pragma omp barrier
  work2(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, 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, page 27Section 2.7.2.1, page 25.

# Stubs for Run-time Library Functions [B]

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"

void omp_set_num_threads(int num_threads)
{
  return;
}


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)
{
  return;
}


int omp_get_dynamic(void)
{
  return 0;
}


int omp_in_parallel(void)
{
  return 0;
}


void omp_set_nested(int nested)
{
  return;
}


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;
}
```

**/* in C++ (ISO/IEC 14882:1998(E)) */**
*statement-seq:*

*statement*

*openmp-directive*

*statement-seq statement*

*statement-seq openmp-directive*

**/* in C (ISO/IEC 9899:1990) */**
*statement-list:*

*statement*

*openmp-directive*

*statement-list statement*

*statement-list openmp-directive*

**/* in C9X (FCD WG14/N843 August 3, 1998) */**
*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-clause*$_{optseq}$ *new-line*

*parallel-clause:*

*unique-parallel-clause*

*data-clause*

*unique-parallel-clause:*

`if (` *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 $sections\text{-}clause_{optseq}$ *new-line*

*sections-clause:*

*data-clause*

nowait

*section-scope:*

{ *section-sequence* }

*section-sequence:*

$section\text{-}directive_{opt}$ *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 $single\text{-}clause_{optseq}$ *new-line*

*single-clause:*

*data-clause*

nowait

*parallel-for-construct:*

*parallel-for-directive iteration-statement*

*parallel-for-directive:*

`# pragma omp parallel for` $parallel\text{-}for\text{-}clause_{optseq}$ *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` $parallel\text{-}sections\text{-}clause_{optseq}$ *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` $region\text{-}phrase_{opt}$ *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* `)`

`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*

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.