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

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

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

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

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

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

Cross References:

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

2.4 Work-sharing Constructs

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

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

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

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

2.4.1 `for` Construct

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

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

The clause is one of the following:

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

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

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

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

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

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

static	When **schedule(static,** *chunk_size***)** is specified, iterations are divided into chunks of a size specified by *chunk_size*. The chunks are statically assigned to threads in the team in a round-robin fashion in the order of the thread number. When no *chunk_size* is specified, the iteration space is divided into chunks that are approximately equal in size, with one chunk assigned to each thread.
dynamic	When **schedule(dynamic,** *chunk_size***)** is specified, the iterations are divided into a series of chunks, each containing *chunk_size* iterations. Each chunk is assigned to a thread that is waiting for an assignment. The thread executes the chunk of iterations and then waits for its next assignment, until no chunks remain to be assigned. Note that the last chunk to be assigned may have a smaller number of iterations. When no *chunk_size* is specified, it defaults to 1.
guided	When **schedule(guided,** *chunk_size***)** is specified, 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 approximately exponentially to 1. For a *chunk_size* with value *k* greater than 1, the sizes decrease approximately exponentially to *k*, except that the last chunk may have fewer than *k* iterations. When no *chunk_size* is specified, it defaults to 1.
runtime	When **schedule(runtime)** is specified, the decision regarding scheduling is deferred until runtime. The schedule kind and size of the chunks can be chosen at run time by setting the environment variable **OMP_SCHEDULE** . If this environment variable is not set, the resulting schedule is implementation-defined. When **schedule(runtime)** is specified, *chunk_size* must not be specified.

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

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

The **ordered** clause must be present when **ordered** directives 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.

Cross References:

- **private**, **firstprivate**, **lastprivate**, and **reduction** clauses, see Section 2.7.2 on page 25.
- **OMP_SCHEDULE** environment variable, see Section 4.1 on page 48.
- **ordered** construct, see Section 2.6.6 on page 23.
- Appendix D, page 93, gives more information on using the **schedule** clause.

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(variable-list)
firstprivate(variable-list)
lastprivate(variable-list)
reduction(operator: variable-list)
nowait
```

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

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

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

Cross References:

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

2.4.3 single Construct

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

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

The clause is one of the following:

```
private(variable-list)
firstprivate(variable-list)
copyprivate(variable-list)
nowait
```

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

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

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

Cross References:

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

2.5 Combined Parallel Work-sharing Constructs

Combined parallel work–sharing constructs are 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:

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

2.5.1 `parallel for` Construct

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

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

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

Cross References:

- **`parallel`** directive, see Section 2.3 on page 8.

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

2.5.2 parallel sections Construct

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

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

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

Cross References:

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

2.6 Master and Synchronization Constructs

The following sections describe the synchronization constructs:

- the **master** directive.
- the **critical** directive.
- the **barrier** directive.
- the **atomic** directive.
- the **flush** directive.
- 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 in the team waits until all of the others have reached this point. The syntax of the **`barrier`** directive is as follows:

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

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

Note that because the **barrier** directive does not have a C language statement as part of its syntax, there are some restrictions on its placement within a program. The directive may appear anywhere a statement may appear, except that it may not appear as the immediate subordinate of a C/C++ control statement (**if**, **switch**, **while**, **do**, **for**), and it can not be labeled (with either a user or a **case**/**default** label). In such a context, it is necessary to enclose the directive in a compound statement.

Note that the C99 standard imposes this same restriction on the placement of declarations that occur after the first executable statement in a function.

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

/* OK - the block within the if statement
 *      is the smallest statement
 *      containing the barrier directive
 */
if (x!=0) {
         #pragma omp 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 it is implementation-defined whether an implementation replaces all **`atomic`** directives with **`critical`** directives that have the same unique *name*, the **`atomic`** directive permits better optimization. Often hardware instructions are available that can perform the atomic update with the least overhead.

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

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

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