OpenMP Fortran Application Program Interface

Oct 1997 1.0

Contents

Page

Page

Tables

Introduction [1]

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

`http://www.openmp.org`

The directives, library routines, and environment variables defined in this document will allow users to create and manage parallel programs while ensuring portability. The directives extend the Fortran sequential programming model with single-program multiple data (SPMD) constructs, work-sharing constructs, synchronization constructs, and provide support for the sharing and privatization of data. The library routines and environment variables provide the functionality to control the run-time execution environment. The directive sentinels are structured so that the directives are treated as Fortran comments. Compilers that support the OpenMP Fortran API will include a command line option that activates and allows interpretation of all OpenMP compiler directives.

1.1 Scope

This specification describes 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 Fortran 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 Fortran API constructs execute correctly.

Compiler-generated automatic parallelization and directives to the compiler to assist such parallelization are not included in this specification.

1.2 Execution Model

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

Upon completion of the parallel construct, the threads in the team synchronize 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 Fortran API allows programmers to use directives in routines 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 allow users 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 routines.

1.3 Compliance

An implementation of the OpenMP Fortran 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, and 4. Appendixes A and B are for information purposes only and are not part of the specification.

The OpenMP Fortran 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 Fortran intrinsics and library routines and Fortran 90 ALLOCATE and DEALLOCATE statements must be thread-safe. Unsynchronized use of such intrinsics and routines by different threads in a parallel region must produce correct results (though not necessarily the same as serial execution results, as in the case of random number generation intrinsics, for example).

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

1.4 Organization

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

- Chapter 2, page 5, describes the compiler directives.
- Chapter 3, page 31, describes the run-time library routines.
- Chapter 4, page 39, describes the environment variables.
- Appendix A, page 41, contains examples.
- Appendix B, page 53, describes stub library routines.

Directives [2]

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

This chapter addresses the following topics:

2.1 OpenMP directive format

The format of an OpenMP directive is as follows:

```
sentinel directive_name [clause[[,] clause]...]
```

All OpenMP compiler directives must begin with a directive *sentinel*. Directives are case insensitive. Clauses can appear in any order after the directive name. Clauses on directives can be repeated as needed, subject to the restrictions listed in the description of each clause. Directives cannot be embedded within continued

statements, and statements cannot be embedded within directives. Comments cannot appear on the same line as a directive.

The following sections contain more information on directive sentinels and describe conditional compilation.

2.1.1 Directive sentinels

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

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

2.1.1.1 Fixed source form directive sentinels

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

```
!$OMP | C$OMP | *$OMP
```

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

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

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

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

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

2.1.1.2 Free source form directive sentinel

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

```
!$OMP
```

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

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

```
!23456789
       !$OMP PARALLEL DO &
                 !$OMP SHARED(A,B,C)

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

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

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

2.1.2 Conditional compilation

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

The sentinel must be followed by a legal Fortran statement on the same line. During OpenMP compilation, the sentinel is replaced by two spaces, and the rest of the line is treated as a normal Fortran statement.

In addition to the Fortran conditional compilation sentinels, a C preprocessor macro, _OPENMP, can be used for conditional compilation. OpenMP-compliant compilers will define this macro during OpenMP compilation.

The following sections contain more information on using the different sentinels for conditional compilation.

2.1.2.1 Fixed source form conditional compilation sentinels

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

```
!$ | C$ | *$
```

The sentinels must start in column one and appear as a single word with no intervening white space. Fortran fixed form line length, case sensitivity, white space, continuation, and column rules apply to the line. Initial lines must have a space or zero in column six, and continuation lines must have a character other than a space or zero in column six.

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

```
C23456789
!$ 10 IAM = OMP_GET_THREAD_NUM() +
!$   &          INDEX

#IFDEF _OPENMP
   10 IAM = OMP_GET_THREAD_NUM() +
     &          INDEX
#ENDIF
```

2.1.2.2 Free source form conditional compilation sentinel

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

```
!$
```

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

2.2 Parallel region construct

The PARALLEL and END PARALLEL directives define a *parallel region*. A parallel region is a block of code that is to be executed by multiple threads in parallel. This is the fundamental parallel construct in OpenMP that starts parallel execution. These directives have the following format:

```
!$OMP PARALLEL [clause[[,] clause]...]

block

!$OMP END PARALLEL
```

The *clause* can be one of the following:

- PRIVATE(*list*)
- SHARED(*list*)
- DEFAULT(PRIVATE | SHARED | NONE)
- FIRSTPRIVATE(*list*)
- REDUCTION ({*operator*|*intrinsic*}:*list*)
- IF(*scalar_logical_expression*)
- COPYIN(*list*)

For information on the PRIVATE, SHARED, DEFAULT, FIRSTPRIVATE, REDUCTION, and COPYIN clauses, see Section 2.6.2, page 22.

When a thread encounters a parallel region, it creates a team of threads, and it becomes the master of the team. The master thread is a member of the team and it has a thread number of 0 within the team. The number of threads in the team is controlled by environment variables and/or library calls. For more information on environment variables, see Chapter 4. For more information on library routines, see Chapter 3, page 31.

The number of physical processors actually hosting the threads at any given time is implementation dependent. Once created, the number of threads in the team remains constant for the duration of that parallel region, but it can be changed either explicitly by the user or automatically by the run-time system from one parallel region to another. The OMP_SET_DYNAMIC library routine 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 routine, see Section 3.1.7, page 34. For more information on the OMP_DYNAMIC environment variable, see Section 4.3, page 40.

The *block* denotes a structured block of Fortran statements. It is illegal to branch into or out of the block. The code contained within the dynamic extent of the parallel region is executed on each thread, and the code path can be different for different threads.

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

If a thread in a team executing a parallel region encounters another parallel region, it creates a new team, and it becomes the master of that new team. By default, nested parallel regions are serialized; that is, they are executed by a team composed of one thread. This default behavior can be changed by using either the `OMP_SET_NESTED` run-time library routine or the `OMP_NESTED` environment variable. For more information on the `OMP_SET_NESTED` library routine, see Section 3.1.9, page 35. For more information on the `OMP_NESTED` environment variable, see Section 4.4, page 40.

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

The following restrictions apply to parallel regions:

- The `PARALLEL/END PARALLEL` directive pair must appear in the same routine in the executable section of the code.
- The code contained by these two directives must be a structured block. It is illegal to branch into or out of a parallel region.
- Only a single `IF` clause can appear on the directive.

2.3 Work-sharing constructs

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

The following restrictions apply to the work-sharing directives:

- Work-sharing constructs and `BARRIER` directives must be encountered by all threads in a team or by none at all.

- Work-sharing constructs and BARRIER directives must be encountered in the same order by all threads in a team.

The following sections describe the work-sharing directives:

- Section 2.3.1, page 11, describes the DO and END DO directives.
- Section 2.3.2, page 13, describes the SECTIONS, SECTION, and END SECTIONS directives.
- Section 2.3.3, page 14, describes the SINGLE and END SINGLE directives.

2.3.1 DO *directive*

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

The format of this directive is as follows:

```
!$OMP DO [clause[[,] clause]...]

do_loop

[!$OMP END DO [NOWAIT]]
```

The *clause* can be one of the following:

- PRIVATE(*list*)
- FIRSTPRIVATE(*list*)
- LASTPRIVATE(*list*)
- REDUCTION({*operator*|*intrinsic*}:*list*)
- SCHEDULE(*type*[,*chunk*])
- ORDERED

The SCHEDULE and ORDERED clauses are described in this section. The PRIVATE, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses are described in Section 2.6.2, page 22.

If ordered sections are contained in the dynamic extent of the DO directive, the ORDERED clause must be present. For more information on ordered sections, see the ORDERED directive in Section 2.5.6, page 21.

The SCHEDULE clause specifies how iterations of the DO loop are divided among the threads of the team. Within the SCHEDULE(*type*[,*chunk*]) clause syntax, *type* can be one of the following:

type	Effect
STATIC	When SCHEDULE(STATIC, *chunk*) is specified, iterations are divided into pieces of a size specified by *chunk*. The pieces are statically assigned to threads in the team in a round-robin fashion in the order of the thread number. *chunk* must be a scalar integer expression. When no *chunk* is specified, the iterations are divided among threads in contiguous pieces, and one piece is assigned to each thread.
DYNAMIC	When SCHEDULE(DYNAMIC,*chunk*) is specified, the iterations are broken into pieces of a size specified by *chunk*. As each thread finishes a piece of the iteration space, it dynamically obtains the next set of iterations. When no *chunk* is specified, it defaults to 1.
GUIDED	When SCHEDULE(GUIDED,*chunk*) is specified, the *chunk* size is reduced in an exponentially decreasing manner with each dispatched piece of the iteration space. *chunk* specifies the minimum number of iterations to dispatch each time, except when there are less than *chunk* number of iterations, at which point the rest are dispatched. When no *chunk* is specified, it defaults to 1.
RUNTIME	When SCHEDULE(RUNTIME) is specified, the decision regarding scheduling is deferred until run time. The schedule type and chunk size can be chosen at run time by setting the OMP_SCHEDULE environment variable. If this environment variable is not set, the resulting schedule is implementation-dependent. For more information on the OMP_SCHEDULE environment variable, see Section 4.1, page 39. When SCHEDULE(RUNTIME) is specified, it is illegal to specify a *chunk*.

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

If an END DO directive is not specified, an END DO directive is assumed at the end of the DO loop. If NOWAIT is specified on the END DO directive, threads do not synchronize at the end of the parallel loop. Threads that finish early proceed straight

to the instructions following the loop without waiting for the other members of the team to finish the DO directive.

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

The following restrictions apply to the DO directives:

- It is illegal to branch out of a DO loop associated with a DO directive.
- The values of the loop control parameters of the DO loop associated with a DO directive must be the same for all the threads in the team.
- The DO loop iteration variable must be of type integer.
- If used, the END DO directive must appear immediately after the end of the loop.
- Only a single SCHEDULE clause can appear on a DO directive.
- Only a single ORDERED clause can appear on a DO directive.

2.3.2 SECTIONS *directive*

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

The format of this directive is as follows:

```
!$OMP SECTIONS [clause[[,] clause]...]

[!$OMP SECTION]

block

[!$OMP SECTION

block]

. . .

!$OMP END SECTIONS [NOWAIT]
```

The *clause* can be one of the following:

- PRIVATE(*list*)
- FIRSTPRIVATE(*list*)
- LASTPRIVATE(*list*)
- REDUCTION({ *operator*|*intrinsic*}:*list*)

The PRIVATE, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses are described in Section 2.6.2, page 22.

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

The following restrictions apply to the SECTIONS directive:

- The code enclosed in a SECTIONS/END SECTIONS directive pair must be a structured block. In addition, each constituent section must also be a structured block. It is illegal to branch into or out of the constituent section blocks.
- It is illegal for a SECTION directive to be outside the lexical extent of the SECTIONS/END SECTIONS directive pair.

2.3.3 SINGLE *directive*

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

The format of this directive is as follows:

```
!$OMP SINGLE [clause[[,] clause]...]

 block

!$OMP END SINGLE [NOWAIT]
```

The *clause* can be one of the following:

- PRIVATE(*list*)
- FIRSTPRIVATE(*list*)

The PRIVATE and FIRSTPRIVATE clauses are described in Section 2.6.2, page 22.

The following restriction applies to a SINGLE directive:

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

2.4 Combined parallel work-sharing constructs

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

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

- Section 2.4.1, page 15, describes the PARALLEL DO and END PARALLEL DO directives.
- Section 2.4.2, page 16, describes the PARALLEL SECTIONS and END PARALLEL SECTIONS directives.

2.4.1 PARALLEL DO *directive*

The PARALLEL DO directive provides a shortcut form for specifying a parallel region that contains a single DO directive.

The format of this directive is as follows:

```
!$OMP PARALLEL DO [clause[[,] clause]...]

do_loop

[!$OMP END PARALLEL DO]
```

The *clause* can be one of the *clause*s accepted by the PARALLEL and DO directives. For information on the PARALLEL directive and the IF clause, see Section 2.2, page 9. For information on the DO directive and the SCHEDULED and ORDERED clauses, see Section 2.3.1, page 11. For information on the remaining clauses, see Section 2.6.2, page 22.

If the END PARALLEL DO directive is not specified, the PARALLEL DO is assumed to end with the DO loop that immediately follows the PARALLEL DO directive. If used, the END PARALLEL DO directive must appear immediately after the end of the DO loop.

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

2.4.2 PARALLEL SECTIONS *directive*

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

The format of this directive is as follows:

```
!$OMP PARALLEL SECTIONS [clause[[,] clause]...]

[!$OMP SECTION ]

block

[!$OMP SECTION

block]

. . .

!$OMP END PARALLEL SECTIONS
```

The *clause* can be one of the *clause*s accepted by the PARALLEL and SECTIONS directives. For more information on the PARALLEL directive, see Section 2.2, page 9. For more information on the SECTIONS directive, see Section 2.3.2, page 13. The PRIVATE, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses are described in Section 2.6.2, page 22.

The last section ends at the END PARALLEL SECTIONS directive.

2.5 Synchronization constructs

The following sections describe the synchronization constructs:

- Section 2.5.1, page 17, describes the MASTER and END MASTER directives.
- Section 2.5.2, page 17, describes the CRITICAL and END CRITICAL directives.
- Section 2.5.3, page 18, describes the BARRIER directive.
- Section 2.5.4, page 18, describes the ATOMIC directive.
- Section 2.5.5, page 20, describes the FLUSH directive.
- Section 2.5.6, page 21, describes the ORDERED and END ORDERED directives.

2.5.1 MASTER *directive*

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

These directives have the following format:

```
!$OMP MASTER
block
!$OMP END MASTER
```

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

This directive has the following restriction:

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

2.5.2 CRITICAL *directive*

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

These directives have the following format:

```
!$OMP CRITICAL [(name)]

block

!$OMP END CRITICAL [(name)]
```

The optional *name* argument identifies the critical section.

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

The following restrictions apply to the CRITICAL directive:

- The section of code enclosed by the CRITICAL and END CRITICAL directive pair must be a structured block. It is illegal to branch into or out of the block.
- If a *name* is specified on a CRITICAL directive, the same *name* must also be specified on the END CRITICAL directive. If no *name* appears on the CRITICAL directive, no *name* can appear on the END CRITICAL directive.

2.5.3 BARRIER *directive*

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

This directive has the following format:

```
!$OMP BARRIER
```

2.5.4 ATOMIC *directive*

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

This directive has the following format:

```
!$OMP ATOMIC
```

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

```
x = x operator expr

x = expr operator x

x = intrinsic (x, expr)

x = intrinsic (expr, x)
```

In the preceding statements:

- *x* is a scalar variable of intrinsic type.
- *expr* is a scalar expression that does not reference *x*.
- *intrinsic* is one of MAX, MIN, IAND, IOR, or IEOR.
- *operator* is one of +, *, -, /, .AND., .OR., .EQV., or .NEQV. .

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

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

The following restriction applies to the ATOMIC directive:

- All references to the storage location *x* are required to have the same type and type parameters.

Example:

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

2.5.5 FLUSH *directive*

The FLUSH directive identifies synchronization points at which the implementation is required to provide a consistent view of memory. Thread-visible variables are written back to memory at the point at which this directive appears.

Thread-visible variables include the following data items:

- Globally visible variables (common blocks and modules).
- Local variables that do not have the SAVE attribute but have had their address taken and saved or have had their address passed to another subprogram.
- Local variables that do not have the SAVE attribute that are declared shared in a parallel region within the subprogram.
- Dummy arguments.
- All pointer dereferences.

Implementations must ensure that modifications to thread-visible variables are visible to all threads after this point. Subsequent reads of thread-visible variables fetch the latest copy of the data. For example, compilers must restore values from registers to memory, and hardware may need to flush write buffers.

This directive has the following format:

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

This directive must appear at the precise point in the code at which the synchronization is required. The optional *list* argument consists of a comma-separated list of variables that need to be flushed in order to avoid flushing all variables. The *list* should contain only named variables. The FLUSH directive is implied for the following directives:

- BARRIER
- CRITICAL and END CRITICAL
- END DO
- END PARALLEL
- END SECTIONS
- END SINGLE
- ORDERED and END ORDERED

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

2.5.6 ORDERED *directive*

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

These directives have the following format:

```
!$OMP ORDERED

block

!$OMP END ORDERED
```

An ORDERED directive can appear only in the dynamic extent of a DO or PARALLEL DO directive. The DO directive to which the ordered section binds must have the ORDERED clause specified (see Section 2.3.1, page 11). One thread is allowed in an ordered section at a time. Threads are allowed to enter in the order of the loop iterations. No thread can enter an ordered section until it is guaranteed that all previous iterations have completed or will never execute an ordered section. This sequentializes and orders code within ordered sections while allowing code outside the section to run in parallel. ORDERED sections that bind to different DO directives are independent of each other.

The following restrictions apply to the ORDERED directive:

- The code enclosed by the ORDERED and END ORDERED directives must be a structured block. It is illegal to branch into or out of the block.
- An ORDERED directive cannot bind to a DO directive that does not have the ORDERED clause specified.
- An iteration of a loop with a DO directive must not execute the same ORDERED directive more than once, and it must not execute more than one ORDERED directive.

2.6 Data environment constructs

This section presents constructs for controlling the data environment during the execution of parallel constructs. Section 2.6.1, page 22, describes the THREADPRIVATE directive, which makes common blocks local to a thread. Section 2.6.2, page 22, describes directive clauses that affect the data environment.

2.6.1 THREADPRIVATE *directive*

The THREADPRIVATE directive makes named common blocks private to a thread but global within the thread.

This directive must appear in the declaration section of the routine after the declaration of the listed common blocks. Each thread gets its own copy of the common block, so data written to the common block by one thread is not directly visible to other threads. During serial portions and MASTER sections of the program, accesses are to the master thread's copy of the common block.

On entry to the first parallel region, data in the THREADPRIVATE common blocks should be assumed to be undefined unless a COPYIN clause is specified on the PARALLEL directive. When a common block that is initialized using DATA statements appears in a THREADPRIVATE directive, each thread's copy is initialized once prior to its first use. For subsequent parallel regions, the data in the THREADPRIVATE common blocks is guaranteed to persist only if the dynamic threads mechanism has been disabled and if the number of threads is the same for all parallel regions. For more information on dynamic threads, see the OMP_SET_DYNAMIC library routine, Section 3.1.7, page 34, and the OMP_DYNAMIC environment variable, Section 4.3, page 40.

The format of this directive is as follows:

```
!$OMP THREADPRIVATE(/cb/[,/cb/]...)
```

cb is the name of the common block to be made private to a thread.

The following restrictions apply to the THREADPRIVATE directive:

- The THREADPRIVATE directive must appear after every declaration of a thread private common block.
- Only named common blocks can be made thread private.
- It is illegal for a THREADPRIVATE common block or its constituent variables to appear in any clause other than a COPYIN clause. As a result, they are not permitted in a PRIVATE, FIRSTPRIVATE, LASTPRIVATE, SHARED, or REDUCTION clause. They are not affected by the DEFAULT clause.

2.6.2 Data scope attribute clauses

Several directives accept clauses that allow a user to control the scope attributes of variables for the duration of the construct. Not all of the following clauses are allowed on all directives, but the clauses that are valid on a particular directive are included with the description of the directive. If no data scope clauses are specified

for a directive, the default scope for variables affected by the directive is SHARED. (See Section 2.6.3, page 28, for exceptions.)

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

The following sections describe the data scope attribute clauses:

- Section 2.6.2.1, page 23, describes the PRIVATE clause.
- Section 2.6.2.2, page 24, describes the SHARED clause.
- Section 2.6.2.3, page 24, describes the DEFAULT clause.
- Section 2.6.2.4, page 25, describes the FIRSTPRIVATE clause.
- Section 2.6.2.5, page 25, describes the LASTPRIVATE clause.
- Section 2.6.2.6, page 25, describes the REDUCTION clause.
- Section 2.6.2.7, page 27, describes the COPYIN clause.

2.6.2.1 PRIVATE *clause*

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

This clause has the following format:

```
PRIVATE(list)
```

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

- A new object of the same type is declared once for each thread in the team. The new object is no longer storage associated with the original object.
- All references to the original object in the lexical extent of the directive construct are replaced with references to the private object.
- Variables defined as PRIVATE are undefined for each thread on entering the construct, and the corresponding shared variable is undefined on exit from a parallel construct.
- Contents, allocation state, and association status of variables defined as PRIVATE are undefined when they are referenced outside the lexical extent (but inside the dynamic extent) of the construct, unless they are passed as actual arguments to called routines.

2.6.2.2 SHARED *clause*

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

This clause has the following format:

```
SHARED(list)
```

2.6.2.3 DEFAULT *clause*

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

This clause has the following format:

```
DEFAULT(PRIVATE | SHARED| NONE)
```

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

- Specifying DEFAULT(PRIVATE) makes all named objects in the lexical extent of the parallel region, including common block variables but excluding THREADPRIVATE variables, private to a thread as if each variable were listed explicitly in a PRIVATE clause.
- Specifying DEFAULT(SHARED) makes all named objects in the lexical extent of the parallel region shared among the threads in a team, as if each variable were listed explicitly in a SHARED clause. In the absence of an explicit DEFAULT clause, the default behavior is the same as if DEFAULT(SHARED) were specified.
- Specifying DEFAULT(NONE) declares that there is no implicit default as to whether variables are PRIVATE or SHARED. In this case, the PRIVATE, SHARED, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION attribute of each variable used in the lexical extent of the parallel region must be specified.

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

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

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

2.6.2.4 FIRSTPRIVATE

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

This clause has the following format:

```
FIRSTPRIVATE(list)
```

Variables that appear in the *list* are subject to PRIVATE clause semantics described in Section 2.6.2.1, page 23. In addition, private copies of the variables are initialized from the original object existing before the construct.

2.6.2.5 LASTPRIVATE *clause*

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

This clause has the following format:

```
LASTPRIVATE(list)
```

Variables that appear in the *list* are subject to the PRIVATE clause semantics described in Section 2.6.2.1, page 23. When the LASTPRIVATE clause appears on a DO directive, the thread that executes the sequentially last iteration updates the version of the object it had before the construct. When the LASTPRIVATE clause appears in a SECTIONS directive, the thread that executes the lexically last SECTION updates the version of the object it had before the construct. Subobjects that are not assigned a value by the last iteration of the DO or the lexically last SECTION of the SECTIONS directive are undefined after the construct.

2.6.2.6 REDUCTION *clause*

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

This clause has the following format:

```
REDUCTION({operator|intrinsic}:list)
```

Variables in *list* must be named scalar variables of intrinsic type.

Variables that appear in a REDUCTION clause must be SHARED in the enclosing context. A private copy of each variable in *list* is created for each thread as if the PRIVATE clause had been used. The private copy is initialized according to the operator. See Table 1, page 28, for more information.

At the end of the REDUCTION, 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 can 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 undefined when the first thread reaches the containing clause, and it remains so until the reduction computation is complete. Normally, the computation is 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 shared variable remains undefined until a barrier synchronization has been performed to ensure that all the threads have completed the REDUCTION clause.

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

```
x = x operator expr

x = expr operator x   (except for subtraction)

x = intrinsic (x,expr)

x = intrinsic (expr, x)
```

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

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

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

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

Table 1. Initialization Values

Operator/Intrinsic	Initialization

+	0
*	1
-	0
.AND.	.TRUE.
.OR.	.FALSE.
.EQV.	.TRUE.
.NEQV.	.FALSE.
MAX	Smallest representable number
MIN	Largest representable number
IAND	All bits on
IOR	0
IEOR	0

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

Example:

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

2.6.2.7 COPYIN *clause*

The COPYIN clause applies only to common blocks that are declared as THREADPRIVATE. A COPYIN clause on a parallel region specifies that the data in the master thread of the team be copied to the thread private copies of the common block at the beginning of the parallel region.

This clause has the following format:

```
COPYIN(list)
```

It is not necessary to specify a whole common block to be copied in. Named variables appearing in the THREADPRIVATE common block can be specified in the *list*.

Example: In the following example, the common blocks BLK1 and FIELDS are specified as thread private, but only one of the variables in common block FIELDS is specified to be copied in.

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

2.6.3 Data environment rules

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

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

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

2. Variables that are privatized in a parallel region cannot be privatized again on an enclosed work-sharing directive. As a result, variables that appear in the PRIVATE, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses on a work-sharing directive must have shared scope in the enclosing parallel region.

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

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

5. Fortran pointers and allocatable arrays can be declared as PRIVATE or SHARED but not as FIRSTPRIVATE or LASTPRIVATE.

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

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

6. PRIVATE or SHARED attributes can be declared for a Cray pointer but not for the pointee. The scope attribute for the pointee is determined at the point of pointer definition. It is illegal to declare a scope attribute for a pointee. Cray pointers may not be specified in FIRSTPRIVATE or LASTPRIVATE clauses.

7. Scope clauses apply only to variables in the static extent of the directive on which the clause appears, with the exception of variables passed as actual

arguments. Local variables in called routines that do not have the SAVE attribute are PRIVATE. Common blocks and modules in called routines in the dynamic extent of a parallel region always have an implicit SHARED attribute, unless they are THREADPRIVATE common blocks.

8. When a named common block is declared as PRIVATE, FIRSTPRIVATE, or LASTPRIVATE, none of its constituent elements may be declared in another scope attribute. It should be noted that when individual members of a common block are privatized, the storage of the specified variables is no longer associated with the storage of the common block itself.
9. Variables that are not allowed in the PRIVATE and SHARED clauses are not affected by DEFAULT(PRIVATE) or DEFAULT(SHARED) clauses, respectively.
10. Clauses can be repeated as needed, but each variable can appear explicitly in only one clause per directive, with the following exceptions:
    - A variable can be specified as both FIRSTPRIVATE and LASTPRIVATE.
    - Variables affected by the DEFAULT clause can be listed explicitly in a clause to override the default specification.

2.7 Directive binding

An implementation that conforms to the OpenMP Fortran API must adhere to the following rules with respect to the dynamic binding of directives:

- The DO, SECTIONS, SINGLE, MASTER, and BARRIER directives bind to the dynamically enclosing PARALLEL directive, if one exists.
- The ORDERED directive binds to the dynamically enclosing DO directive.
- 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.

2.8 Directive nesting

An implementation that conforms to the OpenMP Fortran API must adhere to the following rules with respect to the dynamic nesting of directives:

- A PARALLEL directive dynamically inside another PARALLEL directive logically establishes a new team, which is composed of only the current thread, unless nested parallelism is enabled.
- DO, SECTIONS, and SINGLE directives that bind to the same PARALLEL directive are not allowed to be nested one inside the other.
- DO, SECTIONS, and SINGLE directives are not permitted in the dynamic extent of CRITICAL and MASTER directives.
- BARRIER directives are not permitted in the dynamic extent of DO, SECTIONS, SINGLE, MASTER, and CRITICAL directives.
- MASTER directives are not permitted in the dynamic extent of DO, SECTIONS, and SINGLE directives.
- ORDERED sections are not allowed in the dynamic extent of CRITICAL sections.
- Any directive set that is legal when executed dynamically inside a PARALLEL region is also legal 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.

Run-time Library Routines [3]

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

OpenMP Fortran API run-time library routines are external procedures. In the following descriptions, *scalar_integer_expression* is a default scalar integer expression, and *scalar_logical_expression* is a default scalar logical expression. The return values of these routines are also of default kind.

3.1 Execution Environment Routines

The following sections describe the execution environment routines:

3.1.1 OMP_SET_NUM_THREADS *Subroutine*

The OMP_SET_NUM_THREADS subroutine sets the number of threads to use for the next parallel region.

The format of this subroutine is as follows:

```
SUBROUTINE OMP_SET_NUM_THREADS(scalar_integer_expression)
```

The *scalar_integer_expression* is evaluated, and its value is used as the number of threads to use. This function has effect only when called from serial portions of the program. If this function is called from a portion of the program where the OMP_IN_PARALLEL function returns .TRUE., the behavior of the function is undefined. When dynamic adjustment of the number of threads is enabled, calls to OMP_SET_NUM_THREADS sets the maximum number of threads to use for the next parallel region. For additional information on this subject, see the OMP_SET_DYNAMIC() subroutine described in Section 3.1.7, page 34, and the OMP_GET_DYNAMIC() function described in Section 3.1.8, page 35.

This call has precedence over the OMP_NUM_THREADS environment variable.

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.

This function has the following format:

```
INTEGER FUNCTION OMP_GET_NUM_THREADS()
```

The OMP_SET_NUM_THREADS() call and the OMP_NUM_THREADS environment variable control the number of threads in a team. For more information on the OMP_SET_NUM_THREADS() call, see Section 3.1.1, page 31.

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. For more information on the PARALLEL directive, see Section 2.2, page 9.

If this call is made from the 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 the OMP_GET_NUM_THREADS() function. For more information on OMP_GET_NUM_THREADS(), see Section 3.1.2, page 32.

This function has the following format:

```
INTEGER FUNCTION OMP_GET_MAX_THREADS()
```

If OMP_SET_NUM_THREADS() is used to change the number of threads, subsequent calls to OMP_GET_MAX_THREADS() will return the new value. This function can be used to allocate maximum sized per-thread data structures when the OMP_SET_DYNAMIC() subroutine is set to .TRUE.. For more information on OMP_SET_DYNAMIC(), see Section 3.1.7, page 34.

This function has global scope and returns the maximum value whether executing from 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 the team, that lies between 0 and OMP_GET_NUM_THREADS()-1, inclusive. The master thread of the team is thread 0.

The format of this function is as follows:

```
INTEGER FUNCTION OMP_GET_THREAD_NUM()
```

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

When called from a serial region, OMP_GET_THREAD_NUM returns 0. When 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 number of processors that are available to the program.

The format of this function is as follows:

```
INTEGER FUNCTION OMP_GET_NUM_PROCS()
```

3.1.6 OMP_IN_PARALLEL *Function*

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

The format of this function is as follows:

```
LOGICAL FUNCTION OMP_IN_PARALLEL()
```

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

3.1.7 OMP_SET_DYNAMIC *Subroutine*

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

The format of this subroutine is as follows:

```
SUBROUTINE OMP_SET_DYNAMIC(scalar_logical_expression)
```

If *scalar_logical_expression* evaluates to .TRUE., the number of threads that are used for executing subsequent parallel regions can be adjusted automatically by the run-time environment to obtain the best use of 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. For more information on the OMP_GET_NUM_THREADS() function, see Section 3.1.2, page 32.

If *scalar_logical_expression* evaluates to .FALSE., dynamic adjustment is disabled.

A call to OMP_SET_DYNAMIC has precedence over the OMP_DYNAMIC environment variable. For more information on the OMP_DYNAMIC environment variable, see Section 4.3, page 40.

The default for dynamic thread adjustment 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 platforms.

3.1.8 OMP_GET_DYNAMIC *Function*

The OMP_GET_DYNAMIC function returns .TRUE. if dynamic thread adjustment is enabled and returns .FALSE. otherwise. For more information on dynamic thread adjustment, see Section 3.1.7, page 34.

The format of this function is as follows:

```
LOGICAL FUNCTION OMP_GET_DYNAMIC()
```

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

3.1.9 OMP_SET_NESTED *Subroutine*

The OMP_SET_NESTED subroutine enables or disables nested parallelism.

The format of this subroutine is as follows:

```
SUBROUTINE OMP_SET_NESTED(scalar_logical_expression)
```

If *scalar_logical_expression* evaluates to .FALSE., which is the default, nested parallelism is disabled, and nested parallel regions are serialized and executed by the current thread. If set to .TRUE., nested parallelism is enabled, and parallel regions that are nested can deploy additional threads to form the team.

This call has precedence over the OMP_NESTED environment variable. For more information on the OMP_NESTED environment variable, see Section 4.4, page 40.

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 .TRUE. if nested parallelism is enabled and .FALSE. if nested parallelism is disabled. For more information on nested parallelism, see Section 3.1.9, page 35.

The format of this function is as follows:

```
LOGICAL FUNCTION OMP_GET_NESTED()
```

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

3.2 Lock Routines

The OpenMP run-time library includes a set of general-purpose locking routines. The lock variable, *var*, must be accessed only through the routines described in this section. For all these routines, *var* should be of type integer and of a KIND large enough to hold an address. For example, on 64-bit addressable systems, the *var* may be declared as INTEGER(KIND=8).

The lock control routines are as follows:

- Section 3.2.1, page 36, describes the OMP_INIT_LOCK subroutine.
- Section 3.2.2, page 37, describes the OMP_DESTROY_LOCK subroutine.
- Section 3.2.3, page 37, describes the OMP_SET_LOCK subroutine.
- Section 3.2.4, page 37, describes the OMP_UNSET_LOCK subroutine.
- Section 3.2.5, page 37, describes the OMP_TEST_LOCK function.

3.2.1 OMP_INIT_LOCK *Subroutine*

The OMP_INIT_LOCK subroutine initializes a lock associated with lock variable *var* for use in subsequent calls.

The format of this subroutine is as follows:

```
SUBROUTINE OMP_INIT_LOCK(var)
```

The initial state is unlocked. It is illegal to call this routine with a lock variable that is already associated with a lock.

3.2.2 OMP_DESTROY_LOCK *Subroutine*

The OMP_DESTROY_LOCK subroutine disassociates the given lock variable *var* from any locks.

The format of this subroutine is as follows:

```
SUBROUTINE OMP_DESTROY_LOCK(var)
```

It is illegal to call this routine with a lock variable that has not been initialized.

3.2.3 OMP_SET_LOCK *Subroutine*

The OMP_SET_LOCK subroutine forces the executing thread to wait until the specified lock is available.

The format of this subroutine is as follows:

```
SUBROUTINE OMP_SET_LOCK(var)
```

The thread is granted ownership of the lock when it is available. It is illegal to call this routine with a lock variable that has not been initialized.

3.2.4 OMP_UNSET_LOCK *Subroutine*

The OMP_UNSET_LOCK subroutine releases the executing thread from ownership of the lock.

The format of this subroutine as follows:

```
SUBROUTINE OMP_UNSET_LOCK(var)
```

The behavior is undefined if the thread does not own that lock. It is illegal to call this routine with a lock variable that has not been initialized.

3.2.5 OMP_TEST_LOCK *Function*

The OMP_TEST_LOCK function tries to set the lock associated with the lock variable *var*.

The format of this function is as follows:

```
LOGICAL FUNCTION OMP_TEST_LOCK(var)
```

This function returns .TRUE. if the lock was set successfully, otherwise it returns .FALSE.. It is illegal to call this routine with a lock variable that has not been initialized.

Environment Variables [4]

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

4.1 OMP_SCHEDULE Environment Variable

This variable applies only to DO and PARALLEL DO directives that have the schedule type RUNTIME. For more information on the DO directive, see Section 2.3.1, page 11. For more information on the PARALLEL DO directive, see Section 2.4.1, page 15.

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 DO and PARALLEL DO directives that have a schedule type other than RUNTIME, this environment variable is ignored. The default value for this environment variable is implementation dependent. If the optional chunk size is not set, a chunk size 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.

Examples:

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

4.2 OMP_NUM_THREADS Environment Variable

The OMP_NUM_THREADS environment variable sets the number of threads to use during execution, unless that number is explicitly changed by calling the OMP_SET_NUM_THREADS( ) subroutine. For more information on the OMP_SET_NUM_THREADS( ) subroutine, see Section 3.1.1, page 31.

When dynamic adjustment of the number of threads is enabled, the value of this environment variable is the maximum number of threads to use. The default value is implementation dependent.

Example:

```
setenv OMP_NUM_THREADS 16
```

4.3 OMP_DYNAMIC Environment Variable

The OMP_DYNAMIC environment variable enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. For more information on parallel regions, see Section 2.2, page 9.

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

If set to FALSE, dynamic adjustment is disabled. The default condition is implementation dependent. For more information, see the OMP_SET_DYNAMIC subroutine described in Section 3.1.7, page 34.

Example:

```
setenv OMP_DYNAMIC TRUE
```

4.4 OMP_NESTED Environment Variable

The OMP_NESTED environment variable enables or disables nested parallelism. If set to TRUE, nested parallelism is enabled; if it is set to FALSE, it is disabled. The default value is FALSE. See also Section 3.1.9, page 35.

Example:

```
setenv OMP_NESTED TRUE
```

Examples [A]

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

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 declare it explicitly.

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

The `END PARALLEL DO` directive is optional.

A.2 Specifying Conditional Compilation

The following example illustrates the use of the conditional compilation sentinelprefix. Assuming Fortran fixed source form, the following statement is illegalinvalid when using OpenMP constructs:

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

With OpenMP compilation, the conditional compilation sentinel `!$` is treated as two spaces. As a result, the statement infringes on the statement label field. To be legalvalid, the statement should begin after column 6, like any other fixed source form statement:

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

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

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:

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,NPOINTS)
      IAM = OMP_GET_THREAD_NUM()
      NP =  OMP_GET_NUM_THREADS()
      IPOINTS = NPOINTS/NP
      CALL SUBDOMAIN(X,IAM,IPOINTS)
!$OMP END PARALLEL
```

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 DO directive, as follows:

```
!$OMP PARALLEL
!$OMP DO
      DO I=2,N
        B(I) = (A(I) + A(I-1)) / 2.0
      ENDDO
!$OMP END DO NOWAIT
!$OMP DO
      DO I=1,M
        Y(I) = SQRT(Z(I))
      ENDDO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

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 there are two independent queues in this example, each queue is protected by CRITICAL directives with different names, XAXIS and YAXIS, respectively.

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,Y)
!$OMP CRITICAL(XAXIS)
      CALL DEQUEUE(IX_NEXT, X)
!$OMP END CRITICAL(XAXIS)
      CALL WORK(IX_NEXT, X)
!$OMP CRITICAL(YAXIS)
      CALL DEQUEUE(IY_NEXT,Y)
!$OMP END CRITICAL(YAXIS)
      CALL WORK(IY_NEXT, Y)
!$OMP END PARALLEL
```

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.

```
!$OMP PARALLEL
!$OMP DO LASTPRIVATE(I)
      DO I=1,N
        A(I) = B(I) + C(I)
      ENDDO
!$OMP END PARALLEL
      CALL REVERSE(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:

```
!$OMP PARALLEL DO DEFAULT(PRIVATE) REDUCTION(+: A,B)
      DO I=1,N
        CALL WORK(ALOCAL,BLOCAL)
        A = A + ALOCAL
        B = B + BLOCAL
      ENDDO
```

```
!$OMP END PARALLEL DO
```

A.8 Specifying Parallel Sections

In the following example, subroutines 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/END PARALLEL SECTIONS construct.

```
!$OMP PARALLEL SECTIONS
!$OMP SECTION
      CALL XAXIS
!$OMP SECTION
      CALL YAXIS
!$OMP SECTION
      CALL ZAXIS
!$OMP END PARALLEL SECTIONS
```

A.9 Using SINGLE Directives

TheIn the following example, the first thread that encounters the SINGLE directive executes subroutines OUTPUT and INPUT. 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 SINGLE construct. If other threads can proceed without waiting for the thread executing the SINGLE section, a NOWAIT clause can be specified on the END SINGLE directive.

```
!$OMP PARALLEL DEFAULT(SHARED)
      CALL WORK(X)
!$OMP BARRIER
!$OMP SINGLE
      CALL OUTPUT(X)
      CALL INPUT(Y)
!$OMP END SINGLE
      CALL WORK(Y)
!$OMP END PARALLEL
```

A.10 Specifying Sequential Ordering

Ordered sections are useful for sequentially ordering the output from work that is done in parallel. Assuming that a reentrant I/O library exists, the following program prints out the indexes in sequential order:

```
!$OMP DO ORDERED SCHEDULE(DYNAMIC)
      DO I=LB,UB,ST
        CALL WORK(I)
      END DO


      SUBROUTINE WORK(K)
!$OMP ORDERED
      WRITE(*,*) K
!$OMP END ORDERED
      END
```

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:

```
      CALL OMP_SET_DYNAMIC(.FALSE.)
      CALL OMP_SET_NUM_THREADS(16)
!$OMP PARALLEL DEFAULT(PRIVATE)SHARED(X,NPOINTS)
      IAM = OMP_GET_THREAD_NUM()
      IPOINTS = NPOINTS/16
      CALL DO_BY_16(X,IAM,IPOINTS)
!$OMP END PARALLEL
```

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 program avoids race conditions by protecting all simultaneous updates of the location, by multiple threads, with the ATOMIC directive:

```
!$OMP PARALLEL DO DEFAULT(PRIVATE) SHARED(X,Y,INDEX,N)
      DO I=1,N
        CALL WORK(XLOCAL, YLOCAL)
!$OMP ATOMIC
        X(INDEX(I)) = X(INDEX(I)) + XLOCAL
        Y(I) = Y(I) + YLOCAL
      ENDDO
```

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

A.13 Using the FLUSH Directive

The following example uses the FLUSH directive for point-to-point synchronization between pairs of threads:

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(ISYNC)
      IAM = OMP_GET_THREAD_NUM()
      ISYNC(IAM) = 0
!$OMP BARRIER
      CALL WORK()
C     I AM DONE WITH MY WORK, SYNCHRONIZE WITH MY NEIGHBOR
      ISYNC(IAM) = 1
!$OMP FLUSH(ISYNC)
C     WAIT TILL NEIGHBOR IS DONE
      DO WHILE (ISYNC(NEIGH) .EQ. 0)
!$OMP FLUSH(ISYNC)
      END DO
!$OMP END PARALLEL
```

A.14 Determining the Number of Threads Used

Consider the following incorrect example:

```
      NP = OMP_GET_NUM_THREADS()
!$OMP PARALLEL DO SCHEDULE(STATIC)
      DO I = 0, NP-1
        CALL WORK(I)
      ENDDO
!$OMP END PARALLEL DO
```

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:

```
!$OMP PARALLEL PRIVATE(I)
      I = OMP_GET_THREAD_NUM()
      CALL WORK(I)
!$OMP END PARALLEL
```

A.15 Using Locks

In the following example, note that the argument to the lock routines should be of size POINTER:

```
      PROGRAM LOCK_USAGE
      EXTERNAL OMP_TEST_LOCK
      LOGICAL OMP_TEST_LOCK

      INTEGER LCK         ! THIS VARIABLE SHOULD BE POINTER SIZED

      CALL OMP_INIT_LOCK(LCK)
!$OMP PARALLEL SHARED(LCK) PRIVATE(ID)
      ID = OMP_GET_THREAD_NUM()
      CALL OMP_SET_LOCK(LCK)
      PRINT *, 'MY THREAD ID IS ', ID
      CALL OMP_UNSET_LOCK(LCK)

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

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

      CALL OMP_DESTROY_LOCK( LCK )

      END
```

A.16 Nested DO Directives

The following program is legalcorrect because the inner and outer DO directives bind to different PARALLEL regions:

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
!$OMP PARALLEL SHARED(I,N)
!$OMP DO
        DO J = 1, N
          CALL WORK(I,J)
        END DO
!$OMP END PARALLEL
      END DO
!$OMP END PARALLEL
```

A following variation of the preceding example is also legalcorrect:

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
        CALL SOME_WORK(I,N)
      END DO
!$OMP END PARALLEL



      SUBROUTINE SOME_WORK(I,N)
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO J = 1, N
        CALL WORK(I,J)
```

```
      END DO
!$OMP END PARALLEL
      RETURN
      END
```

A.17 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 2.8, page 30.

The following example is illegalincorrect because the inner and outer DO directives are nested and bind to the same PARALLEL directive:

```
      PROGRAM WRONG1
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
!$OMP DO
        DO J = 1, N
          CALL WORK(I,J)
        END DO
      END DO
!$OMP END PARALLEL
      END
```

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

```
      PROGRAM WRONG2
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
        CALL SOME_WORK(I,N)
      END DO
!$OMP END PARALLEL
      END

      SUBROUTINE SOME_WORK(I,N)
!$OMP DO
      DO J = 1, N
        CALL WORK(I,J)
      END DO
      RETURN
```

```
      END
```

The following example is illegalincorrect because the DO and SINGLE directives are nested, and they bind to the same PARALLEL region:

```
      PROGRAM WRONG3
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
!$OMP SINGLE
      CALL WORK(I)
!$OMP END SINGLE
      END DO
!$OMP END PARALLEL
      END
```

The following example is illegalincorrect because a BARRIER directive inside a SINGLE or a DO can result in deadlock:

```
      PROGRAM WRONG3
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
        CALL WORK(I)
!$OMP BARRIER
        CALL MORE_WORK(I)
      END DO
!$OMP END PARALLEL
      END
```

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

```
      PROGRAM WRONG4
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP CRITICAL
      CALL WORK(N,1)
!$OMP BARRIER
      CALL MORE_WORK(N,2)
!$OMP END CRITICAL
!$OMP END PARALLEL
      END
```

The following example is illegalincorrect because the BARRIER results in deadlock due to the fact that only one thread executes the SINGLE section:

```
      PROGRAM WRONG5
!$OMP PARALLEL DEFAULT(SHARED)
      CALL SETUP(N)
!$OMP SINGLE
      CALL WORK(N,1)
!$OMP BARRIER
      CALL MORE_WORK(N,2)
!$OMP END SINGLE
      CALL FINISH(N)
!$OMP END PARALLEL
      END
```

A.18 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 2.7, page 29.

In the following example, the call from MAIN to SUB2 is legalvalid because the BARRIER (in SUB3) binds to the PARALLEL region in SUB2. The call from MAIN to SUB1 is legalvalid because the BARRIER binds to the PARALLEL region in subroutine SUB2.

```
      PROGRAM MAIN
      CALL SUB1(2)
      CALL SUB2(2)
      END

      SUBROUTINE SUB1(N)
!$OMP PARALLEL PRIVATE(I) SHARED(N)
!$OMP DO
      DO I = 1, N
      CALL SUB2(I)
      END DO
!$OMP END PARALLEL
      END

      SUBROUTINE SUB2(K)
!$OMP PARALLEL SHARED(K)
      CALL SUB3(K)
!$OMP END PARALLEL
      END
```

```
      SUBROUTINE SUB3(N)
      CALL WORK(N)
!$OMP BARRIER
      CALL WORK(N)
      END
```

A.19 Scoping Variables with the PRIVATE Clause

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

```
      INTEGER I,J
      I = 1
      J = 2
!$OMP PARALLEL PRIVATE(I) FIRSTPRIVATE(J)
      I = 3
      J = J+ 2
!$OMP END PARALLEL
      PRINT *, I, J
```

For more information on the PRIVATE clause, see Section 2.6.2.1, page 23.

Stubs for Run-time Library Routines [B]

This section provides stubs for the runtime library routines defined in the OpenMP Fortran API. The stubs are provided to enable portability to platforms that do not support the OpenMP Fortran API. On these platforms, OpenMP programs must be linked with a library containing these stub routines. The stub routines 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 routines must be accessed exclusively through these routines. It should not be initialized or otherwise modified in the user program. Users should not make assumptions about mechanisms used by OpenMP Fortran implementations to implement locks based on the scheme used by the stub routines.

```
SUBROUTINE OMP_SET_NUM_THREADS(NP)
INTEGER NP
RETURN
END

INTEGER FUNCTION OMP_GET_NUM_THREADS()
OMP_GET_NUM_THREADS = 1
RETURN
END

INTEGER FUNCTION OMP_GET_MAX_THREADS()
OMP_GET_MAX_THREADS = 1
RETURN
END

INTEGER FUNCTION OMP_GET_THREAD_NUM()
OMP_GET_THREAD_NUM = 0
RETURN
END

INTEGER FUNCTION OMP_GET_NUM_PROCS()
OMP_GET_NUM_PROCS = 1
RETURN
END

SUBROUTINE OMP_SET_DYNAMIC(FLAG)
LOGICAL FLAG
RETURN
END
```

```
LOGICAL FUNCTION OMP_GET_DYNAMIC()
OMP_GET_DYNAMIC = .FALSE.
RETURN
END

LOGICAL FUNCTION OMP_IN_PARALLEL()
OMP_IN_PARALLEL = .FALSE.
RETURN
END

SUBROUTINE OMP_SET_NESTED(FLAG)
LOGICAL FLAG
RETURN
END

LOGICAL FUNCTION OMP_GET_NESTED()
OMP_GET_NESTED = .FALSE.
RETURN
END

SUBROUTINE OMP_INIT_LOCK(LOCK)
POINTER (LOCK,IL)
INTEGER IL
LOCK = -1
RETURN
END

SUBROUTINE OMP_DESTROY_LOCK(LOCK)
POINTER (LOCK,IL)
INTEGER IL
LOCK = 0
RETURN
END

SUBROUTINE OMP_SET_LOCK(LOCK)
POINTER (LOCK,IL)
INTEGER IL

IF(LOCK .EQ. 0) THEN
  PRINT*, 'ERROR: LOCK NOT INITIALIZED'
  STOP
ELSEIF(LOCK .EQ. 1) THEN
  PRINT*, 'ERROR: DEADLOCK IN USING LOCK VARIABLE'
  STOP
ELSE
```

```
  LOCK = 1
ENDIF
RETURN
END

SUBROUTINE OMP_UNSET_LOCK(LOCK)
POINTER (LOCK,IL)
INTEGER IL
IF(LOCK .EQ. 0) THEN
  PRINT*, 'ERROR: LOCK NOT INITIALIZED'
  STOP
ELSEIF(LOCK .EQ. 1) THEN
  LOCK = -1
ELSE
  PRINT*, 'ERROR: LOCK NOT SET'
  STOP
ENDIF
RETURN
END

LOGICAL FUNCTION OMP_TEST_LOCK(LOCK)
POINTER (LOCK,IL)
INTEGER IL
IF (LOCK .EQ. -1) THEN
  LOCK = 1
  OMP_TEST_LOCK = .TRUE.
ELSEIF(LOCK .EQ. 1) THEN
  OMP_TEST_LOCK = .FALSE.
ELSE
  PRINT*, 'ERROR: LOCK NOT INITIALIZED'
  STOP
ENDIF
RETURN
END
```