OpenMP Fortran Application Program Interface

Version 2.0, November 2000

Line Numbers Added: Sat Oct 28 09:45:59 CDT 2000

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 is 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 and 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 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 executes correctly.

Compiler-generated automatic parallelization is not addressed in this specification.

1.2 Glossary

The following terms are used in this document:

defined - For the contents of a data object, the property of having or being given a valid value. For the allocation status or association status of a data object, the property of having or being given a valid status.

do-construct - The Fortran Standard term for the construct that specifies the repeated execution of a sequence of executable statements. The Fortran Standard calls such a repeated sequence a *loop*. The loop that follows a `DO` or `PARALLEL DO` directive cannot be a `WHILE` loop or a `DO` loop without loop control.

implementation-dependent - A behavior or value that is implementation-dependent is permitted to vary among different OpenMP-compliant implementations (possibly in response to limitations of hardware or operating system). Implementation-dependent items are listed in Appendix E, page 113, and OpenMP-compliant implementations are required to document how these items are handled.

lexical extent - Statements lexically contained within a structured block.

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

nested - a parallel region is said to be nested if it appears within the dynamic extent of a `PARALLEL` construct that (1) does not have an `IF` clause or (2) has an `IF` clause and the logical expression within the clause evaluates to `.TRUE.`.

noncompliant - Code structures or arrangements described as noncompliant are not required to be supported by OpenMP-compliant implementations. Upon encountering such structures, an OpenMP-compliant implementation may produce a compiler error. Even if an implementation produces an executable for a program containing such structures, its execution may terminate prematurely or have unpredictable behavior.

parallel region - Statements that bind to an OpenMP `PARALLEL` construct and are available for execution by multiple threads.

private - Accessible to only one thread in the team for a parallel region. Note that there are several ways to specify that a variable is private: use as a local variable in a subprogram called from a parallel region, in a `THREADPRIVATE` directive, in a `PRIVATE`, `FIRSTPRIVATE`, `LASTPRIVATE`, or `REDUCTION` clause, or use of the variable as a loop control variable.

serialize – When a parallel region is serialized, it is executed by a single thread. A parallel region is said to be serialized if and only if at least one of the following are true:

1. The logical expression in an `IF` clause attached to the parallel directive evaluates to `.FALSE.`.
2. It is a nested parallel region and nested parallelism is disabled.
3. It is a nested parallel region and the implementation chooses to serialize nested parallel regions.

serial region - Statements that do not bind to an OpenMP PARALLEL construct. In other words, these statements are executed by the master thread outside of a parallel region.

shared - Accessible to all threads in the team for a parallel region.

structured block - A structured block is a collection of one or more executable statements with a single point of entry at the top and a single point of exit at the bottom. Execution must always proceed with entry at the top of the block and exit at the bottom with only one exception: it is allowed to have a STOP statement inside a structured block. This statement has the well defined behavior of terminating the entire program.

undefined - For the contents of a data object, the property of not having a determinate value. The result of a reference to a data object with undefined contents is unspecified. For the allocation status or association status of a data object, the property of not having a valid status. The behavior of an operation which relies upon an undefined allocation status or association status is unspecified.

unspecified - A behavior or result that is unspecified is not constrained by requirements in the OpenMP Fortran API. Possibly resulting from the misuse of a language construct or other error, such a behavior or result may not be knowable prior to the execution of a program, and may lead to premature termination of the program.

variable – A data object whose value can be defined and redefined during the execution of a program. It may be a named data object, an array element, an array section, a structure component, or a substring.

white space - A sequence of space or tab characters.

1.3 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/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 *lexical* 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.

1.4 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 1, 2, 3, and 4. The appendixes 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 in a compliant implementation. 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.

In both Fortran 90 and Fortran 95, a variable that has explicit initialization implicitly has the `SAVE` attribute. This is not the case in FORTRAN 77. However, an implementation of OpenMP Fortran must give such a variable the `SAVE` attribute, regardless of the version of Fortran upon which it is based.

The OpenMP Fortran API specifies that certain behavior is "implementation-dependent". A conforming OpenMP implementation is required to define and document its behavior in these cases. See Appendix E, page 113, for a list of implementation-dependent behaviors.

1.5 Organization

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

- Chapter 2, page 7, describes the compiler directives.
- Chapter 3, page 47, describes the run-time library routines.
- Chapter 4, page 59, describes the environment variables.
- Appendix A, page 63, contains examples.
- Appendix B, page 95, describes stub run-time library routines.
- Appendix C, page 101, has information about using the `SCHEDULE` clause.
- Appendix D, page 105, has examples of interfaces for the run-time library routines.
- Appendix E, page 113, describes implementation-dependent behaviors.
- Appendix F, page 115, describes the new features in the OpenMP Fortran v2.0 API.

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 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:

- Section 2.1, page 7, describes the directive format.
- Section 2.2, page 12, describes the parallel region construct.
- Section 2.3, page 15, describes the work-sharing constructs.
- Section 2.4, page 22, describes the combined parallel work-sharing constructs.
- Section 2.5, page 25, describes the synchronization constructs and the `MASTER` directive.
- Section 2.6, page 31, describes the data environment, which includes directives and clauses that affect the data environment.
- Section 2.7, page 45, describes directive binding.
- Section 2.8, page 45, describes directive nesting.

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 preceded by an exclamation point may appear on the same line as a directive.

The following sections describe the OpenMP directive format:

- Section 2.1.1, page 8, describes directive sentinels.
- Section 2.1.2, page 10, describes comments inside directives.
- Section 2.1.3, page 10, describes 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 (spaces and/or tab characters). 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 (spaces and tab characters). 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, prior to any comment placed inside the directive. Continuation directive lines can have an ampersand after the directive sentinel with optional white space before and after the ampersand.

One or more blanks or tabs must be used to separate adjacent keywords in directives in free source form, except in the following cases, where white space is optional between the given pair of keywords:

```
END CRITICAL
END DO
END MASTER
END ORDERED
END PARALLEL
END SECTIONS
END SINGLE
END WORKSHARE
PARALLEL DO
PARALLEL SECTIONS
PARALLEL WORKSHARE
```

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 PARALLELDO SHARED(A,B,C)
```

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

2.1.2 Comments Inside Directives

The OpenMP Fortran API accepts comments placed inside directives. The rules governing such comments depend on the Fortran source form being used.

2.1.2.1 Comments in Directives with Fixed Source Form

Comments may appear on the same line as a directive. The exclamation point initiates a comment when it appears after column 6. The comment extends to the end of the source line and is ignored. If the first nonblank character after the directive sentinel of an initial or continuation directive line is an exclamation point, the line is ignored.

2.1.2.2 Comments in Directives with Free Source Form

Comments may appear on the same line as a directive. The exclamation point initiates a comment. The comment extends to the end of the source line and is ignored. If the first nonblank character after the directive sentinel is an exclamation point, the line is ignored.

2.1.3 Conditional Compilation

The OpenMP Fortran API permits Fortran lines to be compiled conditionally. The directive sentinels for conditional compilation that are accepted by an OpenMP-compliant compiler depend 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.

During OpenMP compilation, the sentinel is replaced by two spaces, and the rest of the line is treated as a normal Fortran line.

If an OpenMP-compliant compiler supports a macro preprocessor (for example, `cpp`), the Fortran processor must define the symbol `_OPENMP` to be used for conditional compilation. This symbol is defined during OpenMP compilation to have the decimal value `YYYYMM` where `YYYY` and `MM` are the year and month designations of the version of the OpenMP Fortran API that the implementation supports.

The following sections contain more information on using the different sentinels for conditional compilation. (See Section A.2, page 63, for an example.)

2.1.3.1 Fixed Source Form Conditional Compilation Sentinels

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

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

The sentinel must start in column 1 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. After the sentinel is replaced with two spaces, initial lines must have a space or zero in column 6 and only white space and numbers in columns 1 through 5. After the sentinel is replaced with two spaces, continuation lines must have a character other than a space or zero in column 6 and only white space in columns 1 through 5. If these criteria are not met, the line is treated as a comment and ignored.

Example: The following forms for specifying conditional compilation in fixed source form are equivalent:

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

#ifdef _OPENMP
   10 IAM = OMP_GET_THREAD_NUM() +
     &          INDEX
#endif
```

2.1.3.2 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, prior to any comment appearing on the conditionally compiled line. Continuation lines can have an ampersand after the sentinel, with optional white space before and after the ampersand.

Example: The following forms for specifying conditional compilation in free source form are equivalent:

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

#ifdef _OPENMP
    IAM = OMP_GET_THREAD_NUM() +    &
        INDEX
#endif
```

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

clause can be one of the following:

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

The IF and NUM_THREADS clauses are described in this section. The PRIVATE, SHARED, DEFAULT, FIRSTPRIVATE, REDUCTION, and COPYIN clauses are described in

Section 2.6.2, page 34. For an example of how to implement coarse-grain parallelism using these directives, see Section A.3, page 64.

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. The number of threads in the team is controlled by environment variables, the NUM_THREADS clause, and/or library calls. For more information on environment variables, see Chapter 4, page 59. For more information on library routines, see Chapter 3, page 47.

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

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. 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 51. For more information on the OMP_DYNAMIC environment variable, see Section 4.3, page 60.

Within the dynamic extent of a parallel region, thread numbers uniquely identify each thread. Thread numbers are consecutive whole numbers ranging from zero for the master thread up to one less than the number of threads within the team. The value of the thread number is returned by a call to the OMP_GET_THREAD_NUM library routine (for more information see Section 3.1.4, page 49). If dynamic threads are disabled when the parallel region is encountered, and remain disabled until a subsequent, non-nested parallel region is encountered, then the thread numbers for the two regions are consistent in that the thread identified with a given thread number in the earlier parallel region will be identified with the same thread number in the later region.

block denotes a structured block of Fortran statements. It is noncompliant to branch into or out of the block. The code contained within the dynamic extent of the parallel region is executed by each thread. 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 past 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. This second parallel region is called a nested parallel region. 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 library routine or the OMP_NESTED environment variable. For more information on the OMP_SET_NESTED library routine, see Section 3.1.9, page 52. For more information on the OMP_NESTED environment variable, see Section 4.4, page 61.

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 NUM_THREADS clause is used to request that a specific number of threads is used in a parallel region. It supersedes the number of threads indicated by the OMP_SET_NUM_THREADS library routine or the OMP_NUM_THREADS environment variable for the parallel region it is applied to. Subsequent parallel regions, however, are not affected unless they have their own NUM_THREADS clauses. *scalar_integer_expression* must evaluate to a positive scalar integer value.

If execution of the program terminates while inside a parallel region, execution of all threads terminates. All work before the previous barrier encountered by the threads is guaranteed to be completed; none of the work after the next barrier that the threads would have encountered will have been started. The amount of work done by each thread in between the barriers and the order in which the threads terminate are unspecified.

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 enclosed in a PARALLEL/END PARALLEL pair must be a structured block. It is noncompliant to branch into or out of a parallel region.
- Only a single IF clause can appear on the directive. The IF expression is evaluated outside the context of the parallel region. Results are unspecified if the IF expression contains a function reference that has side effects.
- Only a single NUM_THREADS clause can appear on the directive. The NUM_THREADS expression is evaluated outside the context of the parallel region. Results are unspecified if the NUM_THREADS expression contains a function reference that has side effects.
- If the dynamic threads mechanism is enabled, then the number of threads requested by the NUM_THREADS clause is the maximum number to use in the parallel region.
- The order of evaluation of IF clauses and NUM_THREADS clauses is unspecified.

- Unsynchronized use of Fortran I/O statements by multiple threads on the same unit has unspecified behavior.

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. When a work-sharing construct is not enclosed dynamically within a parallel region, it is treated as though the thread that encounters it were a team of size one. 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 15, describes the `DO` and `END DO` directives.
- Section 2.3.2, page 18, describes the `SECTIONS`, `SECTION`, and `END SECTIONS` directives.
- Section 2.3.3, page 20, describes the `SINGLE` and `END SINGLE` directives.
- Section 2.3.4, page 20, describes the `WORKSHARE` and `END WORKSHARE` directives.

If `NOWAIT` is specified on the `END DO`, `END SECTIONS`, `END SINGLE`, or `END WORKSHARE` directive, an implementation may omit any code to synchronize the threads at the end of the worksharing construct. In this case, threads that finish early may proceed straight to the instructions following the work-sharing construct without waiting for the other members of the team to finish the work-sharing construct. (See Section A.4, page 64, for an example with the `DO` directive.)

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 *do_loop* may be a *do_construct*, an *outer_shared_do_construct*, or an *inner_shared_do_construct*. A DO construct that contains several DO statements that share the same DO termination statement syntactically consists of a sequence of *outer_shared_do_constructs*, followed by a single *inner_shared_do_construct*. If an END DO directive follows such a DO construct, a DO directive can only be specified for the first (i.e., the outermost) *outer_shared_do_construct*. (See examples in Section A.22, page 81.)

clause can be one of the following:

- PRIVATE(*list*)
- FIRSTPRIVATE(*list*)
- LASTPRIVATE(*list*)
- REDUCTION({*operator*|*intrinsic_procedure_name*}:*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 34.

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 30.

The SCHEDULE clause specifies how iterations of the DO loop are divided among the threads of the team. *chunk* must be a scalar integer expression whose value is positive. The *chunk* expression is evaluated outside the context of the DO construct. Results are unspecified if the *chunk* expression contains a function reference that has side effects. Within the SCHEDULE(*type*[,*chunk*]) clause syntax, *type* can be one of the following:

Table 1. SCHEDULE Clause Values

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. When *chunk* is not specified, the iteration space is divided into contiguous chunks that are approximately equal in size with one chunk 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 iteration space is divided into pieces such that the size of each successive piece is exponentially decreasing. *chunk* specifies the size of the smallest piece, except possibly the last. The size of the initial piece is implementation-dependent. As each thread finishes a piece of the iteration space, it dynamically obtains the next available piece. 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 59. When SCHEDULE(RUNTIME) is specified, it is noncompliant to specify *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.

Threads that complete execution of their assigned loop iterations wait at a barrier at the END DO directive if the NOWAIT clause is not specified. The functionality of

NOWAIT is specified in Section 2.3, page 15. 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, the implied FLUSH at the END DO directive is not performed. (See Section A.4, page 64, for an example of using the NOWAIT clause. See Section 2.5.5, page 29, for a description of implied FLUSH.)

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 noncompliant 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.
- *chunk* must be a positive scalar integer expression.
- The value of the *chunk* parameter must be the same for all of the threads in the team.

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]
```

block denotes a structured block of Fortran statements.

clause can be one of the following:

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

The `PRIVATE`, `FIRSTPRIVATE`, `LASTPRIVATE`, and `REDUCTION` clauses are described in Section 2.6.2, page 34.

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 if the `NOWAIT` clause is not specified. The functionality of `NOWAIT` is described in Section 2.3, page 15.

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 noncompliant to branch into or out of the constituent section blocks.
- It is noncompliant for a `SECTION` directive to be outside the lexical extent of the `SECTIONS`/`END SECTIONS` directive pair. (See Section A.8, page 67 for an example that uses these directives.)

2.3.3 SINGLE Directive

The SINGLE directive specifies that the enclosed code is to be executed by only one thread in the team. Threads in the team that are not executing the SINGLE directive wait at a barrier at the END SINGLE directive if the NOWAIT clause is not specified. The functionality of NOWAIT is described in Section 2.3, page 15.

The format of this directive is as follows:

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

 block

!$OMP END SINGLE [end_single_modifier]
```

where *end_single_modifier* is either COPYPRIVATE(*list*)[[,]COPYPRIVATE(*list*)...] or NOWAIT.

block denotes a structured block of Fortran statements.

clause can be one of the following:

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

The PRIVATE, FIRSTPRIVATE, and COPYPRIVATE clauses are described in Section 2.6.2, page 34.

The following restriction applies to the SINGLE directive:

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

See Section A.9, page 67, for an example of the SINGLE directive.

The following restriction applies to the END SINGLE directive:

- Specification of both a COPYPRIVATE clause and a NOWAIT clause on the same END SINGLE directive is noncompliant.

2.3.4 WORKSHARE Directive

The WORKSHARE directive divides the work of executing the enclosed code into separate units of work, and causes the threads of the team to share the work of executing the enclosed code such that each unit is executed only once. The units of work may be assigned to threads in any manner as long as each unit is executed exactly once.

```
!$OMP WORKSHARE

block

!$OMP END WORKSHARE [NOWAIT]
```

A BARRIER is implied following the enclosed code if the NOWAIT clause is not specified on the END WORKSHARE directive. The functionality of NOWAIT is described in Section 2.3, page 15. An implementation of the WORKSHARE directive must insert any synchronization that is required to maintain standard Fortran semantics. For example, the effects of one statement within *block* must appear to occur before the execution of succeeding statements, and the evaluation of the right hand side of an assignment must appear to have been completed prior to the effects of assigning to the left hand side.

The statements in *block* are divided into units of work as follows:

- For array expressions within each statement, including transformational array intrinsic functions that compute scalar values from arrays:
  - Evaluation of each element of the array expression is a unit of work.
  - Evaluation of transformational array intrinsic functions may be freely subdivided into any number of units of work.
- If a WORKSHARE directive is applied to an array assignment statement, the assignment of each element is a unit of work.
- If a WORKSHARE directive is applied to a scalar assignment statement, the assignment operation is a single unit of work.
- If a WORKSHARE directive is applied to a reference to an elemental function, application of the function to the corresponding elements of any array argument is treated as a unit of work. Hence, if any actual argument in a reference to an elemental function is an array, the reference is treated in the same way as if the function had been applied separately to corresponding elements of each array actual argument.
- If a WORKSHARE directive is applied to a WHERE statement or construct, the evaluation of the mask expression and the masked assignments are workshared.
- If a WORKSHARE directive is applied to a FORALL statement or construct, the evaluation of the mask expression, expressions occurring in the specification of the iteration space, and the masked assignments are workshared.
- For ATOMIC directives and their corresponding assignments, the update of each scalar variable is a single unit of work.
- For CRITICAL constructs, each construct is a single unit of work.

- For PARALLEL constructs, each construct is a single unit of work with respect to the WORKSHARE construct. The statements contained in PARALLEL constructs are executed by new teams of threads formed for each PARALLEL directive.
- If none of the rules above apply to a portion of a statement in *block*, then that portion is a single unit of work.

The transformational array intrinsic functions are MATMUL, DOT_PRODUCT, SUM, PRODUCT, MAXVAL, MINVAL, COUNT, ANY, ALL, SPREAD, PACK, UNPACK, RESHAPE, TRANSPOSE, EOSHIFT, CSHIFT, MINLOC, and MAXLOC.

If an array expression in the block references the value, association status, or allocation status of PRIVATE variables, the value of the expression is undefined, unless the same value would be computed by every thread.

If an array assignment, a scalar assignment, a masked array assignment, or a FORALL assignment assigns to a private variable in the block, the result is unspecified.

The WORKSHARE directive causes the sharing of work to occur only in the lexically enclosed block.

The following restrictions apply to the WORKSHARE directive:

- *block* may contain statements which bind to lexically enclosed PARALLEL constructs. Statements in these PARALLEL constructs are not restricted.
- *block* may contain ATOMIC directives and CRITICAL constructs.
- *block* must only contain array assignment statements, scalar assignment statements, FORALL statements, FORALL constructs, WHERE statements, or WHERE constructs.
- *block* must not contain any user defined function calls unless the function is ELEMENTAL.
- The code enclosed in a WORKSHARE/END WORKSHARE directive pair must be a structured block. It is noncompliant 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 23, describes the PARALLEL DO and END PARALLEL DO directives.
- Section 2.4.2, page 24, describes the PARALLEL SECTIONS and END PARALLEL SECTIONS directives.
- Section 2.4.3, page 24, describes the PARALLEL WORKSHARE and END PARALLEL WORKSHARE 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. (See Section A.1, page 63, for an example.)

The format of this directive is as follows:

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

do_loop

[!$OMP END PARALLEL DO]
```

The *do_loop* may be a *do_construct*, an *outer_shared_do_construct*, or an *inner_shared_do_construct*. A DO construct that contains several DO statements that share the same DO termination statement syntactically consists of a sequence of *outer_shared_do_constructs*, followed by a single *inner_shared_do_construct*. If an END PARALLEL DO directive follows such a DO construct, a PARALLEL DO directive can only be specified for the first (i.e., the outermost) *outer_shared_do_construct*. (See Section A.22, page 81, for examples.)

clause can be one of the clauses accepted by either the PARALLEL or the DO directive. For more information about the PARALLEL directive and the IF and NUM_THREADS clauses, see Section 2.2, page 12. For more information about the DO directive and the SCHEDULE and ORDERED clauses, see Section 2.3.1, page 15. For more information on the remaining clauses, see Section 2.6.2, page 34.

If the END PARALLEL DO directive is not specified, the PARALLEL DO ends 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
```

block denotes a structured block of Fortran statements.

clause can be one of the clauses accepted by either the PARALLEL or the SECTIONS directive. For more information about the PARALLEL directive and the IF and NUM_THREADS clauses, see Section 2.2, page 12. For more information about the SECTIONS directive, see Section 2.3.2, page 18. For more information on the remaining clauses, see Section 2.6.2, page 34.

The last section ends at the END PARALLEL SECTIONS directive.

2.4.3 PARALLEL WORKSHARE *Directive*

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

The format of this directive is as follows:

```
!$OMP PARALLEL WORKSHARE [clause[[,] clause]...]
block
!$OMP END PARALLEL WORKSHARE
```

block denotes a structured block of Fortran statements.

clause can be one of the clauses accepted by either the PARALLEL or the WORKSHARE directive. For more information about the PARALLEL directive and the IF and NUM_THREADS clauses, see Section 2.2, page 12. For more information about the remaining clauses, see Section 2.3.4, page 20.

2.5 Synchronization Constructs and the MASTER Directive

The following sections describe the synchronization constructs and the MASTER directive:

- Section 2.5.1, page 25, describes the MASTER and END MASTER directives.
- Section 2.5.2, page 26, describes the CRITICAL and END CRITICAL directives.
- Section 2.5.3, page 26, describes the BARRIER directive.
- Section 2.5.4, page 27, describes the ATOMIC directive.
- Section 2.5.5, page 29, describes the FLUSH directive.
- Section 2.5.6, page 30, 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.

The format of this directive is as follows:

```
!$OMP MASTER
block
!$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.

The following restriction applies to the MASTER directive:

- The code enclosed in a MASTER/ END MASTER directive pair must be a structured block. It is noncompliant 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.

The format of this directive is as follows:

```
!$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 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 unspecified.

The following restrictions apply to the CRITICAL directive:

- The code enclosed in a CRITICAL/END CRITICAL directive pair must be a structured block. It is noncompliant 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.

See Section A.5, page 64, for an example that uses named CRITICAL sections.

2.5.3 BARRIER Directive

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

The format of this directive is as follows:

```
!$OMP BARRIER
```

The following restrictions apply to the BARRIER directive:

- 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.

2.5.4 ATOMIC Directive

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 format of this directive is as follows:

```
!$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_procedure_name (x, expr_list)

x = intrinsic_procedure_name (expr_list, x)
```

In the preceding statements:

- *x* is a scalar variable of intrinsic type.
- *expr* is a scalar expression that does not reference *x*.
- *expr_list* is a comma-separated, non-empty list of scalar expressions that do not reference *x*. When *intrinsic_procedure_name* refers to IAND, IOR, or IEOR, exactly one expression must appear in *expr_list*.
- *intrinsic_procedure_name* is one of MAX, MIN, IAND, IOR, or IEOR.

- *operator* is one of `+`, `*`, `-`, `/`, `.AND.`, `.OR.`, `.EQV.`, or `.NEQV.` .
- The operators in *expr* must have precedence equal to or greater than the precedence of *operator*, *x operator expr* must be mathematically equivalent to *x operator (expr)*, and *expr operator x* must be mathematically equivalent to *(expr) operator x*.
- The function *intrinsic_procedure_name*, the operator *operator*, and the assignment must be the intrinsic procedure name, the intrinsic operator, and intrinsic assignment.

This directive permits optimization beyond that of the necessary critical section around the update of *x*. An implementation can rewrite the `ATOMIC` directive and the corresponding assignment in the following way using a uniquely named critical section for each object:

```
!$OMP ATOMIC
    x = x operator expr
```

can be rewritten as

```
xtmp = expr
!$OMP CRITICAL (name)
    x = x operator xtmp
!$OMP END CRITICAL (name)
```

where *name* is a unique name corresponding to the type or address of *x*.

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 following restriction applies to the `ATOMIC` directive:

- All atomic references to the storage location of variable *x* throughout the program are required to have the same type and type parameters.

Example:

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

See Section A.12, page 69, and Section A.23, page 82, for more examples using the `ATOMIC` directive.

2.5.5 FLUSH Directive

The FLUSH directive, whether explicit or implied, identifies a sequence point at which the implementation is required to ensure that each thread in the team has a consistent view of certain variables in memory.

A consistent view requires that all memory operations (both reads and writes) that occur before the FLUSH directive in the program be performed before the sequence point in the executing thread; similarly, all memory operations that occur after the FLUSH must be performed after the sequence point in the executing thread.

Implementations must ensure that modifications made to thread-visible variables within the executing thread are made visible to all other threads at the sequence point. For example, compilers must restore values from registers to memory, and hardware may need to flush write buffers. Furthermore, implementations must assume that thread-visible variables may have been updated by other threads at the sequence point and must be retrieved from memory before their first use past the sequence point.

Thread-visible variables are the following data items:

- Globally visible variables (in common blocks and in modules).
- Variables visible through host association.
- Local variables that have the SAVE attribute.
- Variables that appear in an EQUIVALENCE statement with a thread-visible variable.
- Local variables that 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 the enclosing parallel region.
- Dummy arguments.
- All pointer dereferences.

The FLUSH directive only provides consistency between operations within the executing thread and global memory. To achieve a globally consistent view across all threads, each thread must execute a FLUSH operation.

The format of this directive is as follows:

```
!$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 (see Section A.13, page 69). The FLUSH directive is implied for the following directives:

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

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

It should be noted that the FLUSH directive is not implied by the following constructs:

- DO
- MASTER and END MASTER
- SECTIONS
- SINGLE
- WORKSHARE

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.

The format of this directive is as follows:

```
!$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 15). 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 in an ORDERED/END ORDERED directive pair must be a structured block. It is noncompliant 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 to which a DO directive is applied must not execute the same ORDERED directive more than once, and it must not execute more than one ORDERED directive.

See Section A.10, page 68, and Section A.24, page 83, for examples using the 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 32, describes the THREADPRIVATE directive, which makes common blocks or variables local to a thread.
- Section 2.6.2, page 34, describes directive clauses that affect the data environment.
- Section 2.6.3, page 42, describes the data environment rules.

2.6.1 **THREADPRIVATE** *Directive*

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

This directive must appear in the declaration section of a scoping unit in which the common block or variable is declared. Although variables in common blocks can be accessed by use association or host association, common block names cannot. This means that a common block name specified in a THREADPRIVATE directive must be declared to be a common block in the same scoping unit in which the THREADPRIVATE directive appears. Each thread gets its own copy of the common block or variable, so data written to the common block or variable 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 or variable. (See Section A.25, page 84, for examples.)

On entry to the first parallel region, an instance of a variable or common block that appears in a THREADPRIVATE directive is created for each thread. A variable is said to be affected by a COPYIN clause if the variable appears in the COPYIN clause or it is in a common block that appears in the COPYIN clause. If a THREADPRIVATE variable or a variable in a THREADPRIVATE common block is not affected by any COPYIN clause that appears on the first parallel region in a program, the variable or any subobject of the variable is initially defined or undefined according to the following rules:

- If it has the ALLOCATABLE attribute, each copy created will have an initial allocation status of not currently allocated.
- If it has the POINTER attribute:
  - if it has an initial association status of disassociated, either through explicit initialization or default initialization, each copy created will have an association status of disassociated;
  - otherwise, each copy created will have an association status of undefined.
- If it does not have either the POINTER or the ALLOCATABLE attribute:
  - if it is initially defined, either through explicit initialization or default initialization, each copy created is so defined;
  - otherwise, each copy created is undefined.

On entry to a subsequent region, if the dynamic threads mechanism has been disabled, the definition, association, or allocation status of a thread's copy of a THREADPRIVATE variable or a variable in a THREADPRIVATE common block, that is not affected by any COPYIN clause that appears on the region, will be retained, and if it was defined, its value will be retained as well. In this case, if a THREADPRIVATE variable is referenced in both regions, then threads with the same thread number in their respective regions will reference the same copy of that variable. If the dynamic threads mechanism is enabled, the definition and association status of a thread's copy of the variable is undefined, and the allocation status of an allocatable array will be implementation-dependent. A variable with the allocatable attribute must not appear in a `COPYIN` clause, although a structure that has an ultimate component with the allocatable attribute may appear in a `COPYIN` clause. For more information on dynamic threads, see the `OMP_SET_DYNAMIC` library routine, Section 3.1.7, page 51, and the `OMP_DYNAMIC` environment variable, Section 4.3, page 60.

On entry to any parallel region, each thread's copy of a variable that is affected by a `COPYIN` clause for the parallel region will acquire the allocation, association, or definition status of the master thread's copy, according to the following rules:

- If it has the `POINTER` attribute:
  - if the master thread's copy is associated with a target that each copy can become associated with, each copy will become associated with the same target;
  - if the master thread's copy is disassociated, each copy will become disassociated;
  - otherwise, each copy will have an undefined association status.
- If it does not have the `POINTER` attribute, each copy becomes defined with the value of the master thread's copy as if by intrinsic assignment.

If a common block or a variable that is declared in the scope of a module appears in a `THREADPRIVATE` directive, it implicitly has the `SAVE` attribute.

The format of this directive is as follows:

```
!$OMP THREADPRIVATE(list)
```

where *list* is a comma-separated list of named variables and named common blocks. Common block names must appear between slashes.

The following restrictions apply to the `THREADPRIVATE` directive:

- The `THREADPRIVATE` directive must appear after every declaration of a thread private common block.
- A blank common block cannot appear in a `THREADPRIVATE` directive.
- It is noncompliant for a `THREADPRIVATE` variable or common block or its constituent variables to appear in any clause other than a `COPYIN` clause or a `COPYPRIVATE` 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.

- A variable can only appear in a THREADPRIVATE directive in the scope in which it is declared. It must not be an element of a common block or be declared in an EQUIVALENCE statement.
- A variable that appears in a THREADPRIVATE directive and is not declared in the scope of a module must have the SAVE attribute.

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 42, for exceptions.)

Scope attribute clauses that appear on a PARALLEL directive indicate how the specified variables are to be treated with respect to the parallel region associated with the PARALLEL directive. They do not indicate the scope attributes of these variables for any enclosing parallel regions, if they exist.

In determining the appropriate scope attribute for a variable used in the lexical extent of a parallel region, all references and definitions of the variable must be considered, including references and definitions which occur in any nested parallel regions.

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.

When a named common block appears in a list, it has the same meaning as if every explicit member of the common block appeared in the list. A member of a common block is an explicit member if it is named in a COMMON statement which declares the common block, and it was declared in the same scoping unit in which the clause appears.

Although variables in common blocks can be accessed by use association or host association, common block names cannot. This means that a common block name specified in a data scope attribute clause must be declared to be a common block in the same scoping unit in which the data scope attribute clause appears.

The following sections describe the data scope attribute clauses:

- Section 2.6.2.1, page 35, describes the PRIVATE clause.
- Section 2.6.2.2, page 36, describes the SHARED clause.

- Section 2.6.2.3, page 36, describes the DEFAULT clause.
- Section 2.6.2.4, page 37, describes the FIRSTPRIVATE clause.
- Section 2.6.2.5, page 38, describes the LASTPRIVATE clause.
- Section 2.6.2.6, page 38, describes the REDUCTION clause.
- Section 2.6.2.7, page 41, describes the COPYIN clause.
- Section 2.6.2.8, page 41, describes the COPYPRIVATE 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:

1. A new object of the same type is declared once for each thread in the team. One thread in the team is permitted, but not required, to re-use the existing storage as the storage for the new object. For all other threads, new storage is created for the new object.
2. All references to the original object in the lexical extent of the directive construct are replaced with references to the private object.
3. Variables declared as PRIVATE are undefined for each thread on entering the construct, and the corresponding shared variable is undefined on exit from a parallel construct.
4. A variable declared as PRIVATE may be storage-associated with other variables when the PRIVATE clause is encountered. Storage association may exist because of constructs such as EQUIVALENCE, COMMON, etc. If A is a variable appearing in a PRIVATE clause and B is a variable which was storage-associated with A, then:
   a. The contents, allocation, and association status of B are undefined on entry to the parallel construct.
   b. Any definition of A, or of its allocation or association status, causes the contents, allocation, and association status of B to become undefined.
   c. Any definition of B, or of its allocation or association status, causes the contents, allocation, and association status of A to become undefined.

See Section A.20, page 78, and Section A.21, page 78, for examples.

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

6. If a variable is declared as PRIVATE, and the variable is referenced in the definition of a statement function, and the statement function is used within the lexical extent of the directive construct, then the statement function may reference either the SHARED version of the variable or the PRIVATE version. Which version is referenced is implementation-dependent.

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)
```

That each thread in the team access the same storage area for a shared variable does not guarantee that the threads are immediately aware of changes made to the variable by another thread. An implementation may store the new values of shared variables in registers or caches, and those new values may not be stored into the shared storage area until a FLUSH is performed.

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)` requires that each variable used in the lexical extent of the parallel region be explicitly listed in a data scope attribute clause on the parallel region, unless it is one of the following:
  - `THREADPRIVATE`.
  - A Cray pointee (Note: the associated Cray pointer must have its data scope attribute implicitly or explictly specified).
  - A loop iteration variable used only as a loop iteration variable for sequential loops in the lexical extent of the region or parallel `DO` loops that bind to the region.
  - `IMPLIED-DO` or `FORALL` indices.
  - Only used in work-sharing constructs that bind to the region, and is specified in a data scope attribute clause for each such construct.

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 *Clause*

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 35. 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 35. 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 (see Section A.6, page 65, for an example). 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.

If the LASTPRIVATE 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 the thread that executed the sequentially last iteration has stored that variable.

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_procedure_name*, where *operator* is one of the following: +, *, -, .AND., .OR., .EQV., or .NEQV., and *intrinsic_procedure_name* refers to one of the following: MAX, MIN, IAND, IOR, or IEOR.

This clause has the following format:

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

Variables in *list* must be named variables of intrinsic type. Deferred shape and assumed size arrays are not allowed on the reduction clause. Since the intermediate values of the REDUCTION variables may be combined in random order, there is no guarantee that bit-identical results will be obtained for either integer or floating point reductions from one parallel run to another.

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 2, page 40, 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 or a subobject of 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_procedure_name* (*x*,*expr_list*)

x = *intrinsic_procedure_name* (*expr_list*, *x*)

In the preceding statements:

- *x* is a scalar variable of intrinsic type.
- *expr* is a scalar expression that does not reference *x*.
- *expr_list* is a comma-separated, non-empty list of scalar expressions that do not reference *x*. When *intrinsic_procedure_name* refers to `IAND`, `IOR`, or `IEOR`, exactly one expression must appear in *expr_list*.
- *intrinsic_procedure_name* is one of `MAX`, `MIN`, `IAND`, `IOR`, or `IEOR`.
- *operator* is one of `+`, `*`, `-`, `.AND.`, `.OR.`, `.EQV.`, or `.NEQV.` .
- The operators in *expr* must have precedence equal to or greater than the precedence of *operator*, *x operator expr* must be mathematically equivalent to *x*

operator (expr), and *expr operator x* must be mathematically equivalent to *(expr) operator x*.

- The function *intrinsic_procedure_name*, the operator *operator*, and the assignment must be the intrinsic procedure name, the intrinsic operator, and intrinsic assignment.

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 2. Reduction Variable 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

See Section A.7, page 65, for an example that uses the + operator.

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

Example:

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

2.6.2.7 COPYIN *Clause*

The COPYIN clause applies only to variables, common blocks, and variables in 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 blocks or variables at the beginning of the parallel region as described in Section 2.6.1, page 32.

This clause has the following format:

```
COPYIN(list)
```

If a common block appears in a THREADPRIVATE directive, it is not necessary to specify the whole common block. Named variables appearing in the THREADPRIVATE common block can be specified in the *list*.

Although variables in common blocks can be accessed by use association or host association, common block names cannot. This means that a common block name specified in a COPYIN clause must be declared to be a common block in the same scoping unit in which the COPYIN clause appears. See Section A.25, page 84, for more information.

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)
```

An OpenMP-compliant implementation is required to ensure that the value of each thread private copy is the same as the value of the master thread copy when the master thread reached the directive containing the COPYIN clause.

2.6.2.8 COPYPRIVATE *Clause*

The COPYPRIVATE clause uses a private variable to broadcast a value, or a pointer to a shared object, from one member of a team to the other members. It is an alternative to using a shared variable for the value, or pointer association, and is useful when providing such a shared variable would be difficult (for example, in a recursion requiring a different variable at each level). The COPYPRIVATE clause can only appear on the END SINGLE directive.

This clause has the following format:

```
COPYPRIVATE(list)
```

Variables in the *list* must not appear in a PRIVATE or FIRSTPRIVATE clause for the SINGLE construct. If the directive is encountered in the dynamic extent of a parallel region, variables in the list must be private in the enclosing context. If a common block is specified, then it must be THREADPRIVATE, and the effect is the same as if the variable names in its common block object list were specified.

The effect of the COPYPRIVATE clause on the variables in its list occurs after the execution of the code enclosed within the SINGLE construct, and before any threads in the team have left the barrier at the end of the construct. If the variable is not a pointer, then in all other threads in the team, that variable becomes defined (as if by assignment) with the value of the corresponding variable in the thread that executed the enclosed code. If the variable is a pointer, then in all other threads in the team, that variable becomes pointer associated (as if by pointer assignment) with the corresponding variable in the thread that executed the enclosed code. (See Section A.27, page 89, for examples of the COPYPRIVATE clause.)

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 made private 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 may be privatized again on an enclosed work-sharing directive. As a result, variables that appear in a PRIVATE clause on a work-sharing directive may either have a shared or a private scope in the enclosing parallel region. Variables that appear on the FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses on a work-sharing directive must have shared scope in the enclosing parallel region.

3. Variables that appear in a reduction list in a parallel region cannot be privatized on an enclosed work-sharing directive.

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

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

6. Fortran pointers and allocatable arrays can be declared PRIVATE or SHARED but not 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 in a PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION clause 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.

7. 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 noncompliant to declare a scope attribute for a pointee. Cray pointers may not be specified in FIRSTPRIVATE or LASTPRIVATE clauses.

8. Scope clauses apply only to variables in the lexical 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 module variables in called routines in the dynamic extent of a parallel region always have an implicit SHARED attribute, unless they are THREADPRIVATE. Local variables in called routines that have the SAVE attribute are SHARED. (See Section A.26, page 88, for examples.)

9. When a named common block is specified in a PRIVATE, FIRSTPRIVATE, or LASTPRIVATE clause of a directive, none of its constituent elements may be declared in another data scope attribute clause in that directive. 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. (See Section A.25, page 84, for examples.)

10. Variables that are not allowed in the PRIVATE and SHARED clauses are not affected by DEFAULT(PRIVATE) or DEFAULT(SHARED) clauses, respectively.

11. Clauses can be repeated as needed, but each variable and each named common block can appear explicitly in only one clause per directive, with the following exceptions:

    - A variable can be declared both FIRSTPRIVATE and LASTPRIVATE.

- Variables affected by the DEFAULT clause can be listed explicitly in a clause to override the default specification.

12. Variables that are declared LASTPRIVATE or REDUCTION for a work-sharing directive for which NOWAIT appears must not be used prior to a barrier.

13. Variables that appear in namelist statements, in variable format expressions, and in expressions for statement function definitions must not be specified in PRIVATE, FIRSTPRIVATE, or LASTPRIVATE clauses.

14. The shared variables that are specified in REDUCTION or LASTPRIVATE clauses become defined at the end of the construct. Any concurrent uses or definitions of those variables must be synchronized with the definition that occurs at the end of the construct to avoid race conditions.

15. If the following three conditions hold regarding an actual argument in a reference to a non-intrinsic procedure, then any references to (or definitions of) the shared storage that is associated with the dummy argument by any other thread must be synchronized with the procedure reference to avoid possible race conditions:

    a. The actual argument is one of the following:

       - A SHARED variable
       - A subobject of a SHARED variable
       - An object associated with a SHARED variable
       - An object associated with a subobject of a SHARED variable

    b. The actual argument is also one of the following:

       - An array section with a vector subscript
       - An array section
       - An assumed-shape array
       - A pointer array

    c. The associated dummy argument for this actual argument is an explicit-shape array or an assumed-size array.

    The situations described above may result in the value of the shared variable being copied into temporary storage before the procedure reference, and back out of the temporary storage into the actual argument storage after the procedure reference. This effectively results in references to and definitions of the storage during the procedure reference.

16. An OpenMP-compliant implementation must adhere to the following rule:

- If a variable is specified as FIRSTPRIVATE and LASTPRIVATE, the implementation must ensure that the update required for LASTPRIVATE occurs after all initializations for FIRSTPRIVATE.

17. An implementation may generate references to any object that appears or an object in a common block that appears in a REDUCTION, FIRSTPRIVATE, LASTPRIVATE, COPYPRIVATE, or COPYIN clause, on entry to (for FIRSTPRIVATE and COPYIN) or exit from (for REDUCTION, LASTPRIVATE, and COPYPRIVATE) a construct. Except for an object with the pointer attribute in a COPYPRIVATE clause, if a reference to the object as the expression in an intrinsic assignment statement would give an exceptional value, or have undefined behavior, at that point in the program, then the generated reference may have the same behavior.

2.7 Directive Binding

An OpenMP-compliant implementation must adhere to the following rules with respect to the dynamic binding of directives:

- A parallel region is available for binding purposes, whether it is serialized or executed in parallel.
- The DO, SECTIONS, SINGLE, MASTER, BARRIER, and WORKSHARE directives bind to the dynamically enclosing PARALLEL directive, if one exists. (See Section A.19, page 77, for an example.) The dynamically enclosing PARALLEL directive is the closest enclosing PARALLEL directive regardless of the value of the expression in the IF clause, should the clause be present.
- 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 OpenMP-compliant implementation 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, SINGLE, and WORKSHARE directives that bind to the same PARALLEL directive are not allowed to be nested one inside the other.
- DO, SECTIONS, SINGLE, and WORKSHARE directives are not permitted in the dynamic extent of CRITICAL, ORDERED, and MASTER directives.
- BARRIER directives are not permitted in the dynamic extent of DO, SECTIONS, SINGLE, WORKSHARE, MASTER, CRITICAL, and ORDERED directives.
- MASTER directives are not permitted in the dynamic extent of DO, SECTIONS, SINGLE, WORKSHARE, MASTER, CRITICAL, and ORDERED directives.
- ORDERED directives must appear in the dynamic extent of a DO or PARALLEL DO directive which has an ORDERED clause.
- ORDERED directives are not allowed in the dynamic extent of SECTIONS, SINGLE, WORKSHARE, CRITICAL, and MASTER directives.
- CRITICAL directives with the same name are not allowed to be nested one inside the other.
- 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.

See Section A.17, page 73, for legal examples of directive nesting, and Section A.18, page 74, for invalid examples.

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 and two portable timer 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, unless otherwise specified.

Interface declarations for the OpenMP Fortran runtime library routines described in this chapter shall be provided by an OpenMP-compliant implementation in the form of a Fortran `INCLUDE` file named `omp_lib.h` or a Fortran 90 `MODULE` named `omp_lib`. This file must define the following:

- The interfaces of all of the routines in this chapter.
- The `INTEGER PARAMETER omp_lock_kind` that defines the `KIND` type parameters used for simple lock variables in the `OMP_*_LOCK` routines.
- the `INTEGER PARAMETER omp_nest_lock_kind` that defines the `KIND` type parameters used for the nestable lock variables in the `OMP_*_NEST_LOCK` routines.
- the `INTEGER PARAMETER openmp_version` with a value of the C preprocessor macro `_OPENMP` (see Section 2.1.3, page 10) that has the form `YYYYMM` where `YYYY` and `MM` are the year and month designations of the version of the OpenMP Fortran API that the implementation supports.

See Appendix D, page 105, for examples of these files.

3.1 Execution Environment Routines

The following sections describe the execution environment routines:

- Section 3.1.1, page 48, describes the `OMP_SET_NUM_THREADS` subroutine.
- Section 3.1.2, page 48, describes the `OMP_GET_NUM_THREADS` function.
- Section 3.1.3, page 49, describes the `OMP_GET_MAX_THREADS` function.
- Section 3.1.4, page 49, describes the `OMP_GET_THREAD_NUM` function.
- Section 3.1.5, page 50, describes the `OMP_GET_NUM_PROCS` function.
- Section 3.1.6, page 50, describes the `OMP_IN_PARALLEL` function.

- Section 3.1.7, page 51, describes the OMP_SET_DYNAMIC subroutine.
- Section 3.1.8, page 51, describes the OMP_GET_DYNAMIC function.
- Section 3.1.9, page 52, describes the OMP_SET_NESTED subroutine.
- Section 3.1.10, page 52, describes the OMP_GET_NESTED function.

3.1.1 OMP_SET_NUM_THREADS *Subroutine*

The OMP_SET_NUM_THREADS subroutine sets the number of threads to use for subsequent parallel regions.

The format of this subroutine is as follows:

```
SUBROUTINE OMP_SET_NUM_THREADS(scalar_integer_expression)
```

The value of the *scalar_integer_expression* must be positive. The effect of this function depends on whether dynamic adjustment of the number of threads is enabled. If dynamic adjustment is disabled, the value of the *scalar_integer_expression* is used as the number of threads for all subsequent parallel regions prior to the next call to this function; otherwise, the value is used as the maximum number of threads that will be used. This function has effect only when called from serial portions of the program. If it is called from a portion of the program where the OMP_IN_PARALLEL function returns .TRUE., the behavior of this function is unspecified. For additional information on this subject, see the OMP_SET_DYNAMIC subroutine described in Section 3.1.7, page 51, and the OMP_GET_DYNAMIC function described in Section 3.1.8, page 51, and the example in Section A.11, page 68.

Resource constraints on an OpenMP parallel program may change the number of threads that a user is allowed to create at different phases of a program's execution. When dynamic adjustment of the number of threads is enabled, requests for more threads than an implementation can support are satisfied by a smaller number of threads. If dynamic adjustment of the number of threads is disabled, the behavior of this function is implementation-dependent.

This call has precedence over the OMP_NUM_THREADS environment variable (see Section 4.2, page 60).

3.1.2 OMP_GET_NUM_THREADS *Function*

The OMP_GET_NUM_THREADS function returns the number of threads currently in the team executing the parallel region from which it is called.

The format of this function is as follows:

```
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 library routine, see Section 3.1.1, page 48. For more information on the OMP_NUM_THREADS environment variable, see Section 4.2, page 60.

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 12.

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. (See Section A.14, page 70, for an example.)

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 48.

The format of this function is as follows:

```
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 the OMP_SET_DYNAMIC library routine, see Section 3.1.7, page 51.

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 number of the current thread within the team. The thread number lies between 0 and OMP_GET_NUM_THREADS()–1, inclusive. (See the second example in Section A.14, page 70.) 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 12.

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*

OMP_IN_PARALLEL returns the logical OR of the IF clause from all dynamically enclosing parallel regions.

- If a parallel region does not have an IF clause, this is equivalent to IF(.TRUE.) and OMP_IN_PARALLEL returns .TRUE. .
- If there are no dynamically enclosing parallel regions, then OMP_IN_PARALLEL returns .FALSE. .

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` library routine. This function has effect only when called from serial portions of the program. For more information on the `OMP_GET_NUM_THREADS` library routine, see Section 3.1.2, page 48.

If *scalar_logical_expression* evaluates to `.FALSE.`, dynamic adjustment is disabled. (See Section A.11, page 68, for an example.)

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 60.

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 51.

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., nested parallelism is disabled, which is the default, 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 61.

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 52.

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 that take lock variables as arguments. A lock variable must be accessed only through the routines described in this section. For all of these routines, a lock variable should be of type integer and of a KIND large enough to hold an address.

Two types of locks are supported: simple locks and nestable locks. Nestable locks may be locked multiple times by the same thread before being unlocked; simple locks may not be locked if they are already in a locked state. Simple lock variables are associated with simple locks and may only be passed to simple lock routines. Nestable lock variables are associated with nestable locks and may only be passed to nestable lock routines.

In the descriptions that follow, *svar* is a simple lock variable and *nvar* is a nestable lock variable. Using the defined parameters described at the beginning of this chapter (Chapter 3, page 47), these lock variables may be declared as follows:

```
INTEGER (KIND=OMP_LOCK_KIND) :: svar
```

```
INTEGER (KIND=OMP_NEST_LOCK_KIND) :: nvar
```

The simple locking routines are as follows:

- The `OMP_INIT_LOCK` subroutine initializes a simple lock (see Section 3.2.1, page 54).
- The `OMP_DESTROY_LOCK` subroutine removes a simple lock (see Section 3.2.2, page 54).
- The `OMP_SET_LOCK` subroutine sets a simple lock when it becomes available (see Section 3.2.3, page 54).
- The `OMP_UNSET_LOCK` subroutine releases a simple lock (see Section 3.2.4, page 55).
- The `OMP_TEST_LOCK` function tests and possibly sets a simple lock (see Section 3.2.5, page 55).

The nestable lock routines are as follows:

- The `OMP_INIT_NEST_LOCK` subroutine initializes a nestable lock (see Section 3.2.1, page 54).
- The `OMP_DESTROY_NEST_LOCK` subroutine removes a nestable lock (see Section 3.2.2, page 54).
- The `OMP_SET_NEST_LOCK` subroutine sets a nestable lock when it becomes available (see Section 3.2.3, page 54).
- The `OMP_UNSET_NEST_LOCK` subroutine releases a nestable lock (see Section 3.2.4, page 55).
- The `OMP_TEST_NEST_LOCK` function tests and possibly sets a nestable lock (see Section 3.2.5, page 55).

See Section A.15, page 70, and Section A.16, page 71, for examples of using the simple and the nestable lock routines.

3.2.1 OMP_INIT_LOCK *and* OMP_INIT_NEST_LOCK *Subroutines*

These subroutines provide the only means of initializing a lock. Each subroutine initializes a lock associated with the lock variable argument for use in subsequent calls.

The format of these subroutines is as follows:

```
SUBROUTINE OMP_INIT_LOCK(svar)
```

```
SUBROUTINE OMP_INIT_NEST_LOCK(nvar)
```

The initial state is unlocked (that is, no thread owns the lock). For a nestable lock, the initial nesting count is zero. *svar* must be an uninitialized simple lock variable. *nvar* must be an uninitialized nestable lock variable. It is noncompliant to call either of these routines with a lock variable that is already associated with a lock.

3.2.2 OMP_DESTROY_LOCK *and* OMP_DESTROY_NEST_LOCK *Subroutines*

These subroutines insure that the lock variable is uninitialized and cause the lock variable to become undefined.

The format for these subroutines is as follows:

```
SUBROUTINE OMP_DESTROY_LOCK(svar)
```

```
SUBROUTINE OMP_DESTROY_NEST_LOCK(nvar)
```

svar must be an initialized simple lock variable that is unlocked. *nvar* must be an initialized nestable lock variable that is unlocked.

3.2.3 OMP_SET_LOCK *and* OMP_SET_NEST_LOCK *Subroutines*

These subroutines force the thread executing the subroutine to wait until the specified lock is available and then set the lock. A simple lock is available if it is unlocked. A nestable lock is available if it is unlocked or if it is already owned by the thread executing the subroutine.

The format of these subroutines is as follows:

```
SUBROUTINE OMP_SET_LOCK(svar)
```

```
SUBROUTINE OMP_SET_NEST_LOCK(nvar)
```

svar must be an initialized simple lock variable. Ownership of the lock is granted to the thread executing the subroutine.

nvar must be an initialized nestable lock variable. The nesting count is incremented, and the thread is granted, or retains, ownership of the lock.

3.2.4 **OMP_UNSET_LOCK** *and* **OMP_UNSET_NEST_LOCK** *Subroutines*

These subroutines provide the means of releasing ownership of a lock.

The format of these subroutines is as follows:

```
SUBROUTINE OMP_UNSET_LOCK(svar)
```

```
SUBROUTINE OMP_UNSET_NEST_LOCK(nvar)
```

The argument to each of these subroutines must be an initialized lock variable owned by the thread executing the subroutine. The behavior is unspecified if the thread does not own the lock.

The `OMP_UNSET_LOCK` subroutine releases the thread executing the subroutine from ownership of the simple lock associated with *svar*.

The `OMP_UNSET_NEST_LOCK` subroutine decrements the nesting count and releases the thread executing the subroutine from ownership of the nestable lock associated with *nvar* if the resulting count is zero.

3.2.5 **OMP_TEST_LOCK** *and* **OMP_TEST_NEST_LOCK** *Functions*

These functions attempt to set a lock but do not cause the execution of the thread to wait.

The format of these functions is as follows:

```
LOGICAL FUNCTION OMP_TEST_LOCK(svar)
```

```
INTEGER FUNCTION OMP_TEST_NEST_LOCK(nvar)
```

The argument must be an initialized lock variable. These functions attempt to set a lock in the same manner as OMP_SET_LOCK and OMP_SET_NEST_LOCK, except that they do not cause execution of the thread to wait if the lock is already set.

The OMP_TEST_LOCK function returns .TRUE. if the simple lock associated with *svar* is successfully set; otherwise it returns .FALSE. .

The OMP_TEST_NEST_LOCK function returns the new nesting count if the nestable lock associated with *nvar* is successfully set; otherwise, it returns zero. OMP_TEST_NEST_LOCK returns a default integer.

3.3 Timing Routines

The OpenMP run-time library includes two routines supporting a portable wall-clock timer. The routines are as follows:

- The OMP_GET_WTIME function, described in Section 3.3.1, page 56.
- The OMP_GET_WTICK function, described in Section 3.3.2, page 57.

3.3.1 **OMP_GET_WTIME** *Function*

The OMP_GET_WTIME function returns a double precision value equal to the elapsed wallclock time in seconds since some "time in the past". The actual "time in the past" is arbitrary, but it is guaranteed not to change during the execution of the application program.

The format of this function is as follows:

```
DOUBLE PRECISION FUNCTION OMP_GET_WTIME()
```

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

```
DOUBLE PRECISION START, END
START = OMP_GET_WTIME()
!.... work to be timed
END = OMP_GET_WTIME()
PRINT *,'Stuff took ', END-START,' seconds'
```

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

3.3.2 OMP_GET_WTICK Function

The OMP_GET_WTICK function returns a double precision value equal to the number of seconds between successive clock ticks.

The format of this function is as follows:

```
DOUBLE PRECISION FUNCTION OMP_GET_WTICK()
```

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. Character values assigned to them are case insensitive and may have leading or trailing white space.

4.1 `OMP_SCHEDULE` Environment Variable

The `OMP_SCHEDULE` environment 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 15. For more information on the `PARALLEL DO` directive, see Section 2.4.1, page 23.

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. The value takes the form:

type[`,`*chunk*]

where *type* is one of `STATIC`, `DYNAMIC`, or `GUIDED` (see Table 1, page 17) and *chunk* is an optional chunk size. If a chunk size is specified, it must be a positive scalar integer. If *chunk* is present, there may be white space on either side of the "`,`".

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 count 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 library routine. For more information on the OMP_SET_NUM_THREADS library routine, see Section 3.1.1, page 48.

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 value specified must be a positive scalar integer. The default value is implementation dependent. The behavior of the program is implementation-dependent if the requested value of OMP_NUM_THREADS is more than the number of threads an implementation can support.

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 12.

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 value is implementation-dependent. For more information on the OMP_SET_DYNAMIC library routine, see Section 3.1.7, page 51.

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. For more information on nested parallelism, see Section 3.1.9, page 52.

Example:

```
setenv OMP_NESTED TRUE
```

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 using the PARALELL DO directive (specified in Section 2.4.1, page 23). The loop iteration variable is private by default, so it is not necessary to declare it explicitly.

```
!$OMP PARALLEL DO  !I is private by default
      DO I=2,N
        B(I) = (A(I) + A(I-1)) / 2.0
      ENDDO
!$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 sentinel (specified in Section 2.1.3, page 10). Assuming Fortran fixed source form, the following statement is illegal 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 legal, 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 sentinel is replaced with two spaces.

A.3 Using Parallel Regions

The PARALLEL directive (specified in Section 2.2, page 12) 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 (specified in Section 2.3.1, page 15) 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 (for Section 2.5.2, page 26) 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 in a LASTPRIVATE clause (specified in Section 2.6.2.5, page 38) 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 (for Section 2.6.2.6, page 38) 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
```

The following program is noncompliant because the reduction is on the *intrinsic_procedure_name* MAX but that name has been redefined to be the variable named MAX.

```
        MAX = HUGE(0)
        M = 0
!$OMP PARALLEL DO REDUCTION(MAX: M)  ! MAX is no longer the
                                     ! intrinsic so this
                                     ! is invalid
        DO I = 1, 100
          CALL SUB(M,I)
        END DO
        END

        SUBROUTINE SUB(M,I)
          M = MAX(M,I)
        END SUBROUTINE SUB
```

The following compliant program performs the reduction using the *intrinsic_procedure_name* MAX even though the intrinsic MAX has been renamed to REN.

```
        MODULE M
          INTRINSIC MAX
        END MODULE M
        PROGRAM P
          USE M, REN => MAX
          M = 0
 !$OMP PARALLEL DO REDUCTION(REN: M) ! still does MAX
          DO I = 1, 100
            M = MAX(M,I)
          END DO
        END PROGRAM P
```

The following compliant program performs the reduction using *intrinsic_procedure_name* MAX even though the intrinsic MAX has been renamed to MIN.

```
      MODULE MOD
        INTRINSIC MAX, MIN
      END MODULE MOD
      PROGRAM P
        USE MOD, MIN=>MAX, MAX=>MIN
        REAL :: R
        R = -HUGE(0.0)
!$OMP PARALLEL DO REDUCTION(MIN: R) ! still does MAX
        DO I = 1, 1000
          R = MIN(R, SIN(REAL(I)))
        END DO
        PRINT *, R
      END PROGRAM P
```

A.8 Specifying Parallel Sections

In the following example (for Section 2.3.2, page 18), 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

The first thread that encounters the SINGLE directive (specified in Section 2.3.3, page 20) 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 (specified in Section 2.5.6, page 30) 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 (for Section 3.1.1, page 48) 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. If the implementation is not capable of supporting 16 threads, the behavior of this example is implementation-dependent. Note that the number of threads executing a parallel region remains constant during a parallel region, regardless of the dynamic threads setting. The dynamic threads mechanism determines the number of threads to use at the start of the parallel region and keeps it constant for the duration of the region.

A.12 Using the ATOMIC Directive

The following example (for Section 2.5.4, page 27) 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 (for Section 2.5.5, page 29) uses the FLUSH directive for point-to-point synchronization between pairs of threads:

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(ISYNC)
      IAM = OMP_GET_THREAD_NUM()
      ISYNC(IAM) = 0
      NEIGH = GET_NEIGHBOR (IAM)
!$OMP BARRIER
      CALL WORK()
```

```
C     I am done with my work, synchronize with my neighbor
      ISYNC(IAM) = 1
!$OMP FLUSH(ISYNC)
C     Wait until neighbor is done
      DO WHILE (ISYNC(NEIGH) .EQ. 0)
!$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 (specified in Section 3.1.2, page 48) 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

This is an example of the use of the simple lock routines (specified in Section 3.2, page 52).

In the following program, note that the argument to the lock routines should be of type `INTEGER` and of a `KIND` large enough to hold an address:

```
      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 Using Nestable Locks

The following example shows how a nestable lock (specified in Section 3.2, page 52) can be used to synchronize updates both to a structure and to one of its components.

```
      MODULE DATA
       USE OMP_LIB, ONLY OMP_NEXT_LOCK_KIND

       TYPE LOCKED_PAIR
         INTEGER A
         INTEGER B
         INTEGER (OMP_NEST_LOCK_KIND) LCK
```

```
     END TYPE
    END MODULE DATA

    SUBROUTINE INCR_A(P, A)
      ! called only from INCR_PAIR, no need to lock
      USE DATA
      TYPE(LOCKED_PAIR) :: P
      INTEGER A

      P%A = P%A + A
    END SUBROUTINE INCR_A

    SUBROUTINE INCR_B(P, B)
      ! called from both INCR_PAIR and elsewhere,
      ! so we need a nestable lock
      USE OMP_LIB
      USE DATA
      TYPE(LOCKED_PAIR) :: P
      INTEGER B

      CALL OMP_SET_NEST_LOCK(P%LCK)
      P%B = P%B + B
      CALL OMP_UNSET_NEST_LOCK(P%LCK)
    END SUBROUTINE INCR_B

    SUBROUTINE INCR_PAIR(P, A, B)
      USE OMP_LIB
      USE DATA
      TYPE(LOCKED_PAIR) :: P
      INTEGER A
      INTEGER B

      CALL OMP_SET_NEST_LOCK(P%LCK)
      CALL INCR_A(P, A)
      CALL INCR_B(P, B)
      CALL OMP_UNSET_NEST_LOCK(P%LCK)
    END SUBROUTINE INCR_PAIR

    SUBROUTINE F(P)
      USE OMP_LIB
      USE DATA
      TYPE(LOCKED_PAIR) :: P
      INTEGER WORK1, WORK2, WORK3
      EXTERNAL WORK1, WORK2, WORK3
```

```
!$OMP PARALLEL SECTIONS
!$OMP SECTION
        CALL INCR_PAIR(P, WORK1, WORK2)
!$OMP SECTION
        CALL INCR_B(P, WORK3)
!$OMP END PARALLEL SECTIONS
      END SUBROUTINE F
```

A.17 Nested DO Directives

The following example of directive nesting (specified in Section 2.8, page 45) is compliant 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
```

The following variation of the preceding example is also compliant:

```
!$OMP PARALLEL DEFAULT(SHARED)
!$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.18 Examples Showing Incorrect Nesting of Work-sharing Directives

The examples in this section illustrate the directive nesting rules (specified in Section 2.8, page 45).

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

Example 1: Noncompliant Example

```
!$OMP PARALLEL DEFAULT(SHARED)
!$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 noncompliant:

Example 2: Noncompliant Example

```
!$OMP PARALLEL DEFAULT(SHARED)
!$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 noncompliant because the DO and SINGLE directives are nested, and they bind to the same PARALLEL region:

Example 3: Noncomplaint Example

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

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

Example 4: Noncompliant Example

```
!$OMP PARALLEL DEFAULT(SHARED)
!$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 noncompliant because the BARRIER results in deadlock since only one thread at a time can enter the CRITICAL section:

Example 5: Noncompliant Example

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

The following example is noncompliant because the BARRIER results in deadlock since only one thread executes the SINGLE section:

Example 6: Noncompliant Example

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

A.19 Binding of BARRIER Directives

The directive binding rules call for a BARRIER directive to bind to the closest enclosing PARALLEL directive. For more information, see Section 2.7, page 45.

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

The call from MAIN to SUB3 is OpenMP-compliant because the BARRIER does not bind to any parallel region and is ignored. Also note that the BARRIER only synchronizes the team of threads in the enclosing parallel region and not all the threads created in SUB1.

```
      PROGRAM MAIN
      CALL SUB1(2)
      CALL SUB2(2)
      CALL SUB3(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.20 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, see Section 2.6.2.1, page 35.)

A.21 Examples of Noncompliant Storage Association

The following examples illustrate the implications of the PRIVATE clause rules (see Section 2.6.2.1, page 35, rule 4) with regard to storage association:

Example 1: Noncompliant Example

```
       COMMON /BLOCK/ X
       X = 1.0
!$OMP  PARALLEL PRIVATE (X)
       X = 2.0
       CALL SUB()
       ...
!$OMP  END PARALLEL
       ...
       SUBROUTINE SUB()
       COMMON /BLOCK/ X
       ...
       PRINT *,X             ! X is undefined

       ...
       END SUBROUTINE SUB
       END PROGRAM
```

Example 2: Noncompliant Example

```
      COMMON /BLOCK/ X
      X = 1.0
!$OMP PARALLEL PRIVATE (X)
      X = 2.0
      CALL SUB()
      ...
!$OMP END PARALLEL
      ...
      CONTAINS
         SUBROUTINE SUB()
         COMMON /BLOCK/ Y
         ...
         PRINT *,X               ! X is undefined
         PRINT *,Y               ! Y is undefined
         ...
         END SUBROUTINE SUB
      END PROGRAM
```

Example 3: Noncompliant Example

```
      EQUIVALENCE (X,Y)
      X = 1.0
!$OMP PARALLEL PRIVATE(X)
      ...
      PRINT *,Y                  ! Y is undefined
      Y = 10
      PRINT *,X                  ! X is undefined
!$OMP END PARALLEL
```

Example 4: Noncompliant Example

```
      INTEGER A(100), B(100)
      EQUIVALENCE (A(51), B(1))

!$OMP PARALLEL DO DEFAULT(PRIVATE) PRIVATE(I,J) LASTPRIVATE(A)
       DO I=1,100
          DO J=1,100
            B(J) = J - 1
          ENDDO

          DO J=1,100
            A(J) = J         ! B becomes undefined at this point
```

```
              ENDDO
              DO J=1,50
                B(J) = B(J) + 1  ! B is undefined
                                 ! A becomes undefined at this point
              ENDDO
           ENDDO
!$OMP END PARALLEL DO            ! The LASTPRIVATE write for A has
                                 ! undefined results

          PRINT *, B             ! B is undefined since the LASTPRIVATE
                                 ! write of A was not defined
          END
```

Example 5: Noncompliant Example

```
      COMMON /FOO/ A
      DIMENSION B(10)
      EQUIVALENCE (A,B(1))
      ! the common block has to be at least 10 words
      A = 0
!$OMP PARALLEL PRIVATE(/FOO/)
      !
      ! Without the private clause,
      ! we would be passing a member of a sequence
      ! that is at least ten elements long.  With the private
      ! clause, A may no longer be sequence-associated.
      !
      CALL BAR(A)
!$OMP MASTER
      PRINT *, A
!$OMP END MASTER
!$OMP END PARALLEL
      END

      SUBROUTINE BAR(X)
      DIMENSION X(10)
      !
      ! This use of X does not conform to the specification.
      ! It would be legal Fortran 90, but the OpenMP private
      ! directive allows the compiler to break the sequence
      ! association that A had with the rest of the common block.
      !
      FORALL (I = 1:10) X(I) = I
      END
```

A.22 Examples of Syntax of Parallel DO Loops

Both block-do and non-block-do are permitted with PARALLEL DO and work-sharing DO directives. However, if a user specifies an ENDDO directive for a non-block-do construct with shared termination, then the matching DO directive must precede the outermost DO. For more information, see Section 2.3.1, page 15, and Section 2.4.1, page 23.

The following are some examples:

Example 1:

```
      DO 100 I = 1,10
!$OMP   DO
        DO 100 J = 1,10
            ...
100   CONTINUE
```

Example 2:

```
!$OMP DO
      DO 100 J = 1,10
            ...
100     A(I) = I + 1
!$OMP ENDDO
```

Example 3:

```
!$OMP DO
      DO 100 I = 1,10
        DO 100 J = 1,10
            ...
100     CONTINUE
!$OMP ENDDO
```

Example 4: Noncompliant Example

```
      DO 100 I = 1,10
!$OMP   DO
        DO 100 J = 1,10
            ...
100   CONTINUE
!$OMP ENDDO
```

A.23 Examples of the `ATOMIC` Directive

All atomic references to the storage location of each variable that appears on the left-hand side of an `ATOMIC` assignment statement throughout the program are required to have the same type and type parameters. For more information, see Section 2.5.4, page 27.

The following are some examples:

Example 1: Noncompliant Example

```
      INTEGER:: I
      REAL:: R
      EQUIVALENCE(I,R)
!$OMP PARALLEL
      ...
!$OMP ATOMIC
      I = I + 1
      ...
!$OMP ATOMIC
      R = R + 1.0
!$OMP END PARALLEL
```

Example 2: Noncompliant Example

```
      SUBROUTINE FRED()
      COMMON /BLK/ I
      INTEGER I
!$OMP PARALLEL
      ...
!$OMP ATOMIC
      I = I + 1
      ...
      CALL SUB()
!$OMP END PARALLEL
      END

      SUBROUTINE SUB()
      COMMON /BLK/ R
      REAL R
      ...
!$OMP ATOMIC
      R = R + 1
      END
```

Example 3: Noncompliant Example

Although the following example might work on some implementation, this is considered a noncompliant example.

```
      INTEGER:: I
      REAL:: R
      EQUIVALENCE(I,R)
!$OMP PARALLEL
      ...
!$OMP ATOMIC
      I = I + 1
!$OMP END PARALLEL
      ...
!$OMP PARALLEL
      ...
!$OMP ATOMIC
      R = R + 1.0
!$OMP END PARALLEL
```

A.24 Examples of the ORDERED Directive

It is possible to have multiple ORDERED sections within a DO specified with the ORDERED clause. Example 1 is noncompliant, because the API states the following:

> 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.

For more information, see Section 2.5.6, page 30.

Example 1: Noncompliant Example

In this example, all iterations execute 2 ORDERED sections:

```
!$OMP DO
      DO I = 1, N
      ...
!$OMP ORDERED
      ...
!$OMP END ORDERED
      ...
!$OMP ORDERED
      ...
!$OMP END ORDERED
      ...
      END DO
```

Example 2:

This is a compliant example of a DO with more than one ORDERED section:

```
!$OMP DO ORDERED
      DO I = 1,N
        ...
        IF (I <= 10) THEN
          ...
!$OMP ORDERED
          WRITE(4,*) I
!$OMP END ORDERED
        ENDIF
          ...
        IF (I > 10) THEN
          ...
!$OMP ORDERED
          WRITE(3,*) I
!$OMP END ORDERED
        ENDIF
      ENDDO
```

A.25 Examples of THREADPRIVATE Data

The following examples show noncompliant uses and correct uses of the THREADPRIVATE directive. For more information, see Section 2.6.1, page 32, item 8 of Section 2.6.3, page 42, and Section 2.6.2.7, page 41.

Example 1: Noncompliant Example

```
      MODULE FOO
      COMMON /T/ A
      END MODULE FOO

      SUBROUTINE BAR()
      USE FOO
!$OMP THREADPRIVATE(/T/)
      !noncompliant because /T/ not declared in BAR
      !See Section 2.6.1
!$OMP PARALLEL
      ...
!$OMP END PARALLEL
      END SUBROUTINE BAR
```

Example 2: Noncompliant Example

```
      COMMON /T/ A
!$OMP THREADPRIVATE(/T/)
      ...
      CONTAINS
        SUBROUTINE BAR()
!$OMP PARALLEL COPYIN(/T/)
      !noncompliant because /T/ not declared in BAR
      !See Section 2.6.2.7
        ...
!$OMP END PARALLEL
        END SUBROUTINE BAR
      END PROGRAM
```

Example 3: Correct Rewrite of the Previous Example

```
      COMMON /T/ A
!$OMP THREADPRIVATE(/T/)
      ...
      CONTAINS
        SUBROUTINE BAR()
        COMMON /T/ A
!$OMP THREADPRIVATE(/T/)
!$OMP PARALLEL COPYIN(/T/)
        ...
!$OMP END PARALLEL
        END SUBROUTINE BAR
      END PROGRAM
```

Example 4: An example of THREADPRIVATE for local variables

```
      PROGRAM P
      INTEGER, ALLOCATABLE, SAVE :: A(:)
      INTEGER, POINTER, SAVE :: PTR
      INTEGER, SAVE :: I
      INTEGER, TARGET :: TARG
      LOGICAL :: FIRSTIN = .TRUE.
!$OMP THREADPRIVATE(A, B, I, PTR)

      ALLOCATE (A(3))
      A = (/1,2,3/)
      PTR => TARG
      I = 5

!$OMP PARALLEL COPYIN(I, PTR)
!$OMP   CRITICAL
         IF (FIRSTIN) THEN
           TARG = 4           ! Update target of ptr
           I = I + 10
           IF (ALLOCATED(A)) A = A + 10
           FIRSTIN = .FALSE.
         END IF
         IF (ALLOCATED(A)) THEN
           PRINT *, 'a = ', A
         ELSE
           PRINT *, 'A is not allocated'
         END IF
         PRINT *, 'ptr = ', PTR
         PRINT *, 'i = ', I
         PRINT *
!$OMP   END CRITICAL
!$OMP END PARALLEL
      END PROGRAM P
```

This program, if executed by two threads, will print the following.

```
a = 11 12 13
ptr = 4
i = 15

A is not allocated
ptr = 4
i = 5
```

or

```
A is not allocated
ptr = 4
i = 15

a = 1 2 3
ptr = 4
i = 5
```

Example 5: An example of THREADPRIVATE for module variables

```
      MODULE FOO
       REAL, POINTER :: WORK(:)
       SAVE WORK
!$OMP THREADPRIVATE(WORK)
      END MODULE FOO

    SUBROUTINE SUB1(N)
     USE FOO
!$OMP PARALLEL PRIVATE(THE_SUM)
     ALLOCATE(WORK(N))
     CALL SUB2(N,THE_SUM)
     WRITE(*,*)THE_SUM
!$OMP END PARALLEL
     END SUBROUTINE SUB1

     SUBROUTINE SUB2(N,THE_SUM)
       USE FOO
       WORK = 10
       THE_SUM=SUM(WORK)
     END SUBROUTINE SUB2

     PROGRAM BONK
      USE FOO
      N = 10
      CALL SUB1(N)
     END PROGRAM BONK
```

A.26 Examples of the Data Attribute Clauses: SHARED and PRIVATE

When a named common block is specified in a PRIVATE, FIRSTPRIVATE, or LASTPRIVATE clause of a directive, none of its constituent elements may be declared in another scope attribute clause in that directive. The following examples, both compliant and noncompliant, illustrate this point. For more information, see item 8 of Section 2.6.3, page 42.

Example 1:

```
      COMMON /C/ X,Y
!$OMP PARALLEL PRIVATE (/C/)
      ...
!$OMP END PARALLEL
      ...
!$OMP PARALLEL SHARED (X,Y)
      ...
!$OMP END PARALLEL
```

Example 2:

```
      COMMON /C/ X,Y
!$OMP PARALLEL
      ...
!$OMP DO PRIVATE(/C/)
      ...
!$OMP END DO
!
!$OMP DO PRIVATE(X)
      ...
!$OMP END DO
      ...
!$OMP END PARALLEL
```

Example 3: Noncompliant Example

```
      COMMON /C/ X,Y
!$OMP PARALLEL PRIVATE(/C/), SHARED(X)
      ...
!$OMP END PARALLEL
```

Example 4:

```
      COMMON /C/ X,Y
!$OMP PARALLEL PRIVATE (/C/)
      ...
!$OMP END PARALLEL
      ...
!$OMP PARALLEL SHARED (/C/)
      ...
!$OMP END PARALLEL
```

Example 5: Noncompliant Example

```
      COMMON /C/ X,Y
!$OMP PARALLEL PRIVATE(/C/), SHARED(/C/)
      ...
!$OMP END PARALLEL
```

Example 6:

```
       MODULE M
        REAL A
      CONTAINS
        SUBROUTINE SUB
!$OMP PARALLEL PRIVATE(A)
        CALL SUB1()
!$OMP END PARALLEL
        END SUBROUTINE SUB
        SUBROUTINE SUB1()
        A = 5   ! This is A in module M, not the PRIVATE
                ! A in SUB
        END SUBROUTINE SUB1
      END MODULE M
```

A.27 Examples of the Data Attribute Clause: COPYPRIVATE

Example 1. The COPYPRIVATE clause (specified in Section 2.6.2.8, page 41) can be used to broadcast the value resulting from a read statement directly to all instances of a private variable.

```
      SUBROUTINE INIT(A,B)
```

```
      COMMON /XY/ X,Y
!$OMP THREADPRIVATE (/XY/)
!$OMP SINGLE
      READ (11) A,B,X,Y
!$OMP END SINGLE COPYPRIVATE (A,B,/XY/)
      END
```

If subroutine INIT is called from a serial region, its behavior is not affected by the presence of the directives. If it is called from a parallel region, then the actual arguments with which A and B are associated must be private. After the read statement has been executed by one thread, no thread leaves the construct until the private objects designated by A, B, X, and Y in all threads have become defined with the values read.

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

```
      REAL FUNCTION READ_NEXT()
      REAL, POINTER :: TMP
!$OMP SINGLE
      ALLOCATE (TMP)
!$OMP END SINGLE COPYPRIVATE (TMP)

!$OMP MASTER
      READ (11) TMP
!$OMP END MASTER

!$OMP BARRIER
      READ_NEXT = TMP
!$OMP BARRIER

!$OMP SINGLE
      DEALLOCATE (TMP)
!$OMP END SINGLE NOWAIT
      END FUNCTION READ_NEXT
```

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

```
   FUNCTION NEW_LOCK()
    INTEGER(OMP_LOCK_KIND), POINTER :: NEW_LOCK
```

```
  !$OMP SINGLE
     ALLOCATE(NEW_LOCK)
     CALL OMP_INIT_LOCK(NEW_LOCK)
  !$OMP END SINGLE COPYPRIVATE(NEW_LOCK)
END FUNCTION NEW_LOCK
```

Example 4. Note that the effect of the copyprivate clause on a variable with the allocatable attribute is different than on a variable with the pointer attribute.

```
      SUBROUTINE S(N)
      REAL, DIMENSION(:), ALLOCATABLE :: A
      REAL, DIMENSION(:), POINTER :: B
      ALLOCATE (A(N))
!$OMP SINGLE
      ALLOCATE (B(N))
      READ (11) A,B
!$OMP END SINGLE COPYPRIVATE(A,B)
      ! Variable A designates a private object
      !   which has the same value in each thread
      ! Variable B designates a shared object
      ...
!$OMP BARRIER
!$OMP SINGLE
      DEALLOCATE (B)
!$OMP END SINGLE NOWAIT
      END SUBROUTINE S
```

A.28 Examples of the WORKSHARE Directive

In the following examples of the WORKSHARE directive (specified in Section 2.3.4, page 20), assume that all 2 letter variable names (e.g., AA, BB) are conformable arrays and single letter names (e.g., I, X) are scalars; implicit typing rules hold. Each of the examples is enclosed in a parallel region. All of the examples are fixed source form so the directives start in column 1.

Example 1. WORKSHARE spreads work across some number of threads and there is a barrier after the last statement. Implementations must enforce Fortran execution rules inside of the WORKSHARE block.

```
!$OMP WORKSHARE
      AA = BB
```

```
        CC = DD
        EE = FF
!$OMP END WORKSHARE
```

Example 2. The final barrier can be eliminated with NOWAIT:

```
!$OMP WORKSHARE
      AA = BB
      CC = DD
!$OMP END WORKSHARE NOWAIT

!$OMP WORKSHARE
      EE = FF
!$OMP END WORKSHARE
```

Threads doing CC = DD immediately begin work on EE = FF when they are done with CC = DD.

Example 3. ATOMIC can be used with WORKSHARE:

```
!$OMP WORKSHARE
      AA = BB
!$OMP ATOMIC
      I = I + SUM(AA)
      CC = DD
!$OMP END WORKSHARE
```

The computation of SUM(AA) is workshared, but the update to I is ATOMIC.

Example 4. Fortran WHERE and FORALL statements are *compound statements* of the form:

```
WHERE (EE .ne. 0) FF = 1 / EE
FORALL (I=1:N, XX(I) .ne. 0) YY(I) = 1 / XX(I)
```

They are made up of a *control* part and a *statement* part. When WORKSHARE is applied to one of these compound statements, both the *control* and the *statement* parts are workshared.

```
!$OMP WORKSHARE
      AA = BB
      CC = DD
      WHERE (EE .ne. 0) FF = 1 / EE
      GG = HH
!$OMP END WORKSHARE
```

Each task gets worked on in order by the threads:

```
      AA = BB     then
      CC = DD     then
      EE .ne. 0   then
      FF = 1 / EE then
      GG = HH
```

Example 5. An assignment to a shared scalar variable is performed by one thread in a WORKSHARE while all other threads in the team wait. SHR is a shared scalar variable in this example.

```
!$OMP WORKSHARE
      AA = BB
      SHR = 1
      CC = DD
!$OMP END WORKSHARE
```

Noncompliant Example 6. An assignment to a private scalar variable is performed by one thread in a WORKSHARE while all other threads wait. The private scalar variable is undefined after the assignment statement. PRI is a private scalar variable in this example.

```
!$OMP WORKSHARE
      AA = BB
      PRI = 1
      CC = DD
!$OMP END WORKSHARE
```

Example 7. Fortran execution rules must be enforced inside a WORKSHARE construct. Hence, the same result is produced in the following program fragment regardless of whether the code is executed sequentially or inside an OpenMP program with multiple threads:

```
!$OMP WORKSHARE
    A(1:50) = B(11:60)
    G(11:20) = A(1:10)
!$OMP END WORKSHARE
```

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 such 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. It is declared as a `POINTER` to guarantee that it is capable of holding an address. Alternatively, for Fortran 90 implementations, it could be declared as an `INTEGER(OMP_LOCK_KIND)` or `INTEGER(OMP_NEST_LOCK_KIND)`, as appropriate. In an actual implementation the lock variable might be used to hold the address of an allocated object, but here it is used to hold an integer value. 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
```

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

      SUBROUTINE OMP_SET_DYNAMIC(FLAG)
      LOGICAL FLAG
      RETURN
      END

      LOGICAL FUNCTION OMP_GET_DYNAMIC()
      OMP_GET_DYNAMIC = .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)
      ! LOCK is 0 if the simple lock is not initialized
      !        -1 if the simple lock is initialized but not set
      !         1 if the simple lock is set
      POINTER (LOCK,IL)
      INTEGER IL
      LOCK = -1
      RETURN
      END

      SUBROUTINE OMP_INIT_NEST_LOCK(NLOCK)
      ! NLOCK is  0 if the nestable lock is not initialized
      !          -1 if the nestable lock is initialized but not set
      !           1 if the nestable lock is set
      ! no use count is maintained
      POINTER (NLOCK,NIL)
      INTEGER NIL
      NLOCK = -1
      RETURN
      END
```

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

      SUBROUTINE OMP_DESTROY_NEST_LOCK(NLOCK)
      POINTER (NLOCK,NIL)
      INTEGER NIL
      NLOCK = 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_SET_NEST_LOCK(NLOCK)
      POINTER (NLOCK,NIL)
      INTEGER NIL

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

      SUBROUTINE OMP_UNSET_NEST_LOCK(NLOCK)
      POINTER (NLOCK,NIL)
      INTEGER NIL

      IF (NLOCK .EQ. 0) THEN
        PRINT *, 'ERROR: NESTED LOCK NOT INITIALIZED'
        STOP
      ELSEIF (NLOCK .EQ. 1) THEN
        NLOCK = -1
      ELSE
        PRINT *, 'ERROR: NESTED 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

INTEGER FUNCTION OMP_TEST_NEST_LOCK(NLOCK)
POINTER (NLOCK,NIL)
INTEGER NIL

IF (NLOCK .EQ. -1) THEN
  NLOCK = 1
  OMP_TEST_NEST_LOCK = 1
ELSEIF (NLOCK .EQ. 1) THEN
  OMP_TEST_NEST_LOCK = 0
ELSE
  PRINT *, 'ERROR: NESTED LOCK NOT INITIALIZED'
  STOP
ENDIF

RETURN
END

DOUBLE PRECISION OMP_WTIME()
! This function does not provide a working
! wall-clock timer. Replace it with a version
! customized for the target machine.
OMP_WTIME = 0
RETURN
END

DOUBLE PRECISION OMP_WTICK()
! This function does not provide a working
! clock tick function. Replace it with
! a version customized for the target machine.
DOUBLE PRECISION ONE_YEAR
PARAMETER  (ONE_YEAR=365.D0*86400.D0)
OMP_WTICK=ONE_YEAR
RETURN
END
```

Using the SCHEDULE Clause [C]

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

When there are repeated references to the same objects, the choice of schedule for a DO construct may be determined primarily by characteristics of the memory system, such as the presence and size of caches and whether memory access times are uniform or nonuniform. Such considerations may make it preferable to have each thread consistently refer to the same set of elements of an array in a series of loops, even if some threads are assigned relatively less work in some of the loops. This can be done by using the STATIC schedule with the same bounds for all the loops. In the following example, note that 1 is used as the lower bound in the second loop, even though K would be more natural if the schedule were not important.

```
!$OMP PARALLEL
!$OMP DO SCHEDULE(STATIC)
      DO I=1,N
        A(I) = WORK1(I)
      ENDDO
!$OMP DO SCHEDULE(STATIC)
      DO I=1,N
        IF(I .GE. K) A(I) = A(I) + WORK2(I)
      ENDDO
!$OMP END PARALLEL
      ENDDO
```

In the remaining examples, it is assumed that memory access is not the dominant consideration, and, unless otherwise stated, that all threads receive comparable computational resources. In these cases, the choice of schedule for a DO construct depends on all the shared work that is to be performed between the nearest preceding barrier and either the implied closing barrier or the nearest subsequent barrier, if there is a NOWAIT clause. For each kind of schedule, a short example shows how that schedule kind is likely to be the best choice. A brief discussion follows each example.

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

```
!$OMP PARALLEL DO SCHEDULE(STATIC)
      DO I=1,N
        CALL INVARIANT_AMOUNT_OF_WORK(I)
```

```
      ENDDO
```

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

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

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

```
!$OMP PARALLEL DO SCHEDULE(DYNAMIC)
      DO I=1,N
        CALL UNPREDICTABLE_AMOUNT_OF_WORK(I)
      ENDDO
```

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

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

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

```
!$OMP PARALLEL
!$OMP SECTIONS
      ..........
!$OMP END SECTIONS NOWAIT
```

```
!$OMP DO SCHEDULE(GUIDED)
      DO I=1,N
        CALL INVARIANT_AMOUNT_OF_WORK(I)
      ENDDO
```

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

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

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

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

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

Interface Declaration Module [D]

This appendix gives examples of the Fortran INCLUDE file and Fortran 90 module that shall be provided by implementations as specified in Chapter 3, page 47.

It has three sections:

- Section D.1, page 105, contains an example of a FORTRAN 77 interface declaration INCLUDE file.
- Section D.2, page 107, contains an example of a Fortran 90 interface declaration MODULE.
- Section D.3, page 111, contains an example of a Fortran 90 generic interface for a library routine.

D.1 Example of an Interface Declaration INCLUDE File

```
C     the "C" of this comment starts in column 1
       integer     omp_lock_kind
       parameter ( omp_lock_kind = 8 )

       integer     omp_nest_lock_kind
       parameter ( omp_nest_lock_kind = 8 )

C                          default integer type assumed below
C                          default logical type assumed below
C                          OpenMP Fortran API v1.1
       integer     openmp_version
       parameter ( openmp_version = 200011 )

       external omp_destroy_lock

       external omp_destroy_nest_lock

       external omp_get_dynamic
       logical  omp_get_dynamic

       external omp_get_max_threads
       integer  omp_get_max_threads

       external omp_get_nested
```

```
      logical  omp_get_nested

      external omp_get_num_procs
      integer  omp_get_num_procs

      external omp_get_num_threads
      integer  omp_get_num_threads

      external omp_get_thread_num
      integer  omp_get_thread_num

      external omp_get_wtick
      double precision  omp_get_wtick

      external omp_get_wtime
      double precision  omp_get_wtime

      external omp_init_lock

      external omp_init_nest_lock

      external omp_in_parallel
      logical  omp_in_parallel

      external omp_set_dynamic

      external omp_set_lock

      external omp_set_nest_lock

      external omp_set_nested

      external omp_set_num_threads

      external omp_test_lock
      logical  omp_test_lock

      external omp_test_nest_lock
      integer  omp_test_nest_lock

      external omp_unset_lock

      external omp_unset_nest_lock
```

D.2 Example of a Fortran 90 Interface Declaration MODULE

```
!     the "!" of this comment starts in column 1

      module omp_lib_kinds

       integer, parameter :: omp_integer_kind      = 4
       integer, parameter :: omp_logical_kind      = 4
       integer, parameter :: omp_lock_kind         = 8
       integer, parameter :: omp_nest_lock_kind    = 8

      end module omp_lib_kinds

      module omp_lib

        use omp_lib_kinds

!                              OpenMP Fortran API v1.1
       integer, parameter :: openmp_version = 199910

       interface
         subroutine omp_destroy_lock ( var )
         use omp_lib_kinds
         integer ( kind=omp_lock_kind ), intent(inout) :: var
         end subroutine omp_destroy_lock
       end interface

       interface
         subroutine omp_destroy_nest_lock ( var )
         use omp_lib_kinds
         integer ( kind=omp_nest_lock_kind ), intent(inout) :: var
         end subroutine omp_destroy_nest_lock
       end interface

       interface
         function omp_get_dynamic ()
         use omp_lib_kinds
         logical ( kind=omp_logical_kind ) :: omp_get_dynamic
         end function omp_get_dynamic
       end interface

       interface
         function omp_get_max_threads ()
         use omp_lib_kinds
```

```
        integer ( kind=omp_integer_kind ) :: omp_get_max_threads
        end function omp_get_max_threads
      end interface

      interface
        function omp_get_nested ()
        use omp_lib_kinds
        logical ( kind=omp_logical_kind ) :: omp_get_nested
        end function omp_get_nested
      end interface

      interface
        function omp_get_num_procs ()
        use omp_lib_kinds
        integer ( kind=omp_integer_kind ) :: omp_get_num_procs
        end function omp_get_num_procs
      end interface

      interface
        function omp_get_num_threads ()
        use omp_lib_kinds
        integer ( kind=omp_integer_kind ) :: omp_get_num_threads
        end function omp_get_num_threads
      end interface

      interface
        function omp_get_thread_num ()
        use omp_lib_kinds
        integer ( kind=omp_integer_kind ) :: omp_get_thread_num
        end function omp_get_thread_num
      end interface

      interface
        function omp_get_wtick ()
        double precision :: omp_get_wtick
        end function omp_get_wtick
       end interface

      interface
        function omp_get_wtime ()
        double precision :: omp_get_wtime
        end function omp_get_wtime
      end interface

      interface
```

```
    subroutine omp_init_lock ( var )
    use omp_lib_kinds
    integer ( kind=omp_lock_kind ), intent(out) :: var
    end subroutine omp_init_lock
  end interface

  interface
    subroutine omp_init_nest_lock ( var )
    use omp_lib_kinds
    integer ( kind=omp_nest_lock_kind ), intent(out) :: var
    end subroutine omp_init_nest_lock
  end interface

  interface
    function omp_in_parallel ()
    use omp_lib_kinds
    logical ( kind=omp_logical_kind ) :: omp_in_parallel
    end function omp_in_parallel
  end interface

  interface
    subroutine omp_set_dynamic ( enable_expr )
    use omp_lib_kinds
    logical ( kind=omp_logical_kind ), intent(in) :: enable_expr
    end subroutine omp_set_dynamic
  end interface

  interface
    subroutine omp_set_lock ( var )
    use omp_lib_kinds
    integer ( kind=omp_lock_kind ), intent(inout) :: var
   end subroutine omp_set_lock
  end interface

  interface
    subroutine omp_set_nest_lock ( var )
    use omp_lib_kinds
    integer ( kind=omp_nest_lock_kind ), intent(inout) :: var
    end subroutine omp_set_nest_lock
  end interface

  interface
    subroutine omp_set_nested ( enable_expr )
    use omp_lib_kinds
    logical ( kind=omp_logical_kind ), intent(in) :: &
```

```
     &                                                     enable_expr
          end subroutine omp_set_nested
        end interface

        interface
          subroutine omp_set_num_threads ( number_of_threads_expr )
          use omp_lib_kinds
          integer ( kind=omp_integer_kind ), intent(in) :: &
     &                                            number_of_threads_expr
          end subroutine omp_set_num_threads
        end interface

        interface
          function omp_test_lock ( var )
          use omp_lib_kinds
          logical ( kind=omp_logical_kind ) :: omp_test_lock
          integer ( kind=omp_lock_kind ), intent(inout) :: var
          end function omp_test_lock
        end interface

        interface
          function omp_test_nest_lock ( var )
          use omp_lib_kinds
          integer ( kind=omp_integer_kind ) :: omp_test_nest_lock
          integer ( kind=omp_nest_lock_kind ), intent(inout) :: var
          end function omp_test_nest_lock
        end interface

        interface
          subroutine omp_unset_lock ( var )
          use omp_lib_kinds
          integer ( kind=omp_lock_kind ), intent(inout) :: var
          end subroutine omp_unset_lock
        end interface

        interface
          subroutine omp_unset_nest_lock ( var )
          use omp_lib_kinds
          integer ( kind=omp_nest_lock_kind ), intent(inout) :: var
          end subroutine omp_unset_nest_lock
        end interface
       end module omp_lib
```

D.3 Example of a Generic Interface for a Library Routine

Any of the OMP runtime library routines that take an argument may be extended with a generic interface so arguments of different KIND type can be accomodated.

Assume an implementation supports both default INTEGER as KIND = OMP_INTEGER_KIND and another INTEGER KIND, KIND = SHORT_INT. Then OMP_SET_NUM_THREADS could be specified in the omp_lib module as the following:

```
!     the "!" of this comment starts in column 1
      interface omp_set_num_threads
       subroutine omp_set_num_threads_1 ( number_of_threads_expr )
       use omp_lib_kinds
       integer ( kind=omp_integer_kind ), intent(in) :: &
     &                                         number_of_threads_expr
       end subroutine omp_set_num_threads_1
        subroutine omp_set_num_threads_2 ( number_of_threads_expr )
       use omp_lib_kinds
       integer ( kind=short_int ), intent(in) :: &
     &                                         number_of_threads_expr
       end subroutine omp_set_num_threads_2
     end interface omp_set_num_threads
```

Implementation-Dependent Behaviors in OpenMP Fortran [E]

This appendix sumarizes the behaviors that are described as “implementation dependent” in this API. Each behavior is cross-referenced back to its description in the main specification. An implementation is required to define and document its behavior in these cases.

- `SCHEDULE(GUIDED,`*chunk*`)`: *chunk* specifies the size of the smallest piece, except possibly the last. The size of the initial piece is implementation dependent (Table 1, page 17).
- 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 (Table 1, page 17).
- In the absence of the `SCHEDULE` clause, the default schedule is implementation-dependent (Section 2.3.1, page 15).
- `OMP_GET_NUM_THREADS`: If the number of threads has not been explicitly set by the user, the default is implementation-dependent (Section 3.1.2, page 48).
- `OMP_SET_DYNAMIC`: The default for dynamic thread adjustment is implementation-dependent (Section 3.1.7, page 51).
- `OMP_SET_NESTED`: When nested parallelism is enabled, the number of threads used to execute nested parallel regions is implementation-dependent (Section 3.1.9, page 52).
- `OMP_SCHEDULE` environment variable: The default value for this environment variable is implementation-dependent (Section 4.1, page 59).
- `OMP_NUM_THREADS` environment variable: The default value is implementation-dependent (Section 4.2, page 60).
- `OMP_DYNAMIC` environment variable: The default value is implementation-dependent (Section 4.3, page 60).
- An implementation can replace all `ATOMIC` directives by enclosing the statement in a critical section (Section 2.5.4, page 27).
- If the dynamic threads mechanism is enabled on entering a parallel region, the allocation status of an allocatable array that is not affected by a `COPYIN` clause that appears on the region is implementation-dependent (Section 2.6.1, page 32).

- Due to resource constraints, it is not possible for an implementation to document the maximum number of threads that can be created successfully during a program's execution. This number is dependent upon the load on the system, the amount of memory allocated by the program, and the amount of implementation dependent stack space allocated to each thread. If the dynamic threads mechanism is disabled, the behavior of the program is implementation-dependent when more threads are requested than can be successfully created. If the dynamic threads mechanism is enabled, requests for more threads than an implementation can support are satisfied by a smaller number of threads (Section 2.3.1, page 15).

- If an OMP runtime library routine interface is defined to be generic by an implementation, use of arguments of kind other than those specified by the `OMP_*_KIND` constants is implementation-dependent (Section D.3, page 111).

New Features in OpenMP Fortran version 2.0 [F]

This appendix summarizes the key changes made to the OpenMP Fortran specification in moving from version 1.1 to version 2.0. The following items are new features added to the specification:

- The FORTRAN 77 standard does not require that initialized data have the `SAVE` attribute but Fortran 95 does require this. OpenMP Fortran version 2.0 requires this. See Section 1.4, page 4.
- An OpenMP compliant implementation must document its implementation-defined behaviors. See Appendix E, page 113.
- Directives may contain end-of-line comments starting with an exclamation point. See Section 2.1.2, page 10.
- The `_OPENMP` preprocessor macro is defined to be an integer of the form `YYYYMM` where `YYYY` and `MM` are the year and month of the version of the OpenMP Fortran specification supported by the implementation. See Section 2.1.3, page 10.
- `COPYPRIVATE` is a new modifier on `END SINGLE`. See Section 2.6.2.8, page 41.
- `THREADPRIVATE` may now be applied to variables as well as `COMMON` blocks. See Section 2.6.1, page 32.
- `REDUCTION` is now allowed on an array name. See Section 2.6.2.6, page 38.
- `COPYIN` now works on variables as well as `COMMON` blocks. See Section 2.6.2.7, page 41.
- Reprivatization of variables is now allowed. See Section 2.6.3, page 42.
- Nested lock routines consistent with those defined in the C/C++ specification have been added. See Section 3.2, page 52.
- Wallclock timers have been added. See Section 3.3, page 56.
- An example of `INTERFACE` definitions for all of the OpenMP runtime routines has been added to the specification. See Appendix D, page 105.
- The `NUM_THREADS` clause on parallel regions defines the number of threads to be used to execute that region. See Section 2.2, page 12.
- The `WORKSHARE` directive allows parallelization of array expressions in Fortran statements. See Section 2.3.4, page 20.

The following items list changes that served to clarify features or to correct errors within the OpenMP Fortran specification:

- Under the right circumstances, subsequent parallel regions use the same threads with the same thread numbers as previous regions. See Section 2.2, page 12.
- It is implementation-defined whether global variable references in statement functions refer to `SHARED` or `PRIVATE` copies of those variables. See Section 2.6.2, page 34
- Exceptional values (such as negative infinity) may affect the behavior of a program. This can occur with `REDUCTION`, `FIRSTPRIVATE`, `LASTPRIVATE`, `COPYPRIVATE`, or `COPYIN`. See Section 2.6.3, page 42.
- Additional examples have been added. See Appendix A, page 63.