2.5.1.1 Determining the Schedule of a
Work-sharing Loop .......................... 38

2.5.2 sections Construct .......................... 39

2.5.3 single Construct .......................... 42

2.5.4 workshare Construct ......................... 44

2.6 Combined Parallel Work-sharing Constructs ......................... 46

2.6.1 Parallel loop construct ........................ 47

2.6.2 parallel sections Construct ...................... 48

2.6.3 parallel workshare Construct ..................... 50

2.7 Master and Synchronization Constructs ......................... 51

2.7.1 master Construct .......................... 51

2.7.2 critical Construct .......................... 52

2.7.3 barrier Construct .......................... 54

2.7.4 atomic Construct .......................... 55

2.7.5 flush Construct ............................ 58

2.7.6 ordered Construct .......................... 61

2.8 Data Environment ................................ 63

2.8.1 Sharing Attribute Rules ......................... 63

2.8.1.1 Sharing Attribute Rules for Variables Referenced
in a Construct ................................. 63

2.8.1.2 Sharing Attribute Rules for Variables Referenced
in a Region, but not in a Construct ................. 65

2.8.2 threadprivate Directive ........................ 66

2.8.3 Data-Sharing Attribute Clauses .................... 70

2.8.3.1 default clause .......................... 71

2.8.3.2 shared clause .......................... 72

2.8.3.3 private clause .......................... 73

2.8.3.4 firstprivate clause ........................ 75

2.8.3.5 lastprivate clause ........................ 77

2.8.3.6 reduction clause ........................ 79
2.8.4 Data Copying Clauses ........................................... 83
  2.8.4.1 copyin clause ............................................. 84
  2.8.4.2 copyprivate clause ...................................... 85
2.9 Nesting of Regions ................................................. 87

3. Runtime Library Routines ........................................... 89
  3.1 Runtime Library Definitions .................................... 90
  3.2 Execution Environment Routines .................................. 91
    3.2.1 omp_set_num_threads ..................................... 91
    3.2.2 omp_get_num_threads ..................................... 93
    3.2.3 omp_get_max_threads ..................................... 94
    3.2.4 omp_get_thread_num ..................................... 95
    3.2.5 omp_get_num_procs ....................................... 96
    3.2.6 omp_in_parallel ......................................... 96
    3.2.7 omp_set_dynamic ......................................... 97
    3.2.8 omp_get_dynamic ......................................... 99
    3.2.9 omp_set_nested .......................................... 100
    3.2.10 omp_get_nested ......................................... 101
  3.3 Lock Routines .................................................. 102
    3.3.1 omp_init_lock and omp_init_nest_lock .................. 104
    3.3.2 omp_destroy_lock and omp_destroy_nest_lock .......... 105
    3.3.3 omp_set_lock and omp_set_nest_lock .................. 105
    3.3.4 omp_unset_lock and omp_unset_nest_lock ............ 106
    3.3.5 omp_test_lock and omp_test_nest_lock ............... 107
  3.4 Timing Routines ................................................ 108
    3.4.1 omp_get_wtime .......................................... 109
    3.4.2 omp_get_wtick .......................................... 110

4. Environment Variables ............................................. 113
  4.1 OMP_SCHEDULE ................................................. 114
  4.2 OMP_NUM_THREADS .............................................. 115
A.26 The private Clause ........................................ 168
A.27 Reprivatization ........................................ 170
A.28 Fortran Restrictions on Storage Association with the
        private Clause ........................................ 171
A.29 C/C++ Arrays in a firstprivate Clause .............. 174
A.30 The lastprivate Clause ................................ 175
A.31 The reduction Clause .................................... 176
A.32 The copyin Clause ....................................... 180
A.33 The copyprivate Clause ................................ 181
A.34 Nested Loop Constructs ................................ 185
A.35 Restrictions on Nesting of Regions .................... 187
A.36 The omp_set_dynamic and omp_set_num_threads
        Routines .............................................. 193
A.37 The omp_get_num_threads Routine ..................... 195
A.38 The omp_init_lock Routine ............................. 197
A.39 Simple Lock Routines .................................. 198
A.40 Nestable Lock Routines ................................ 200

B. Stubs for Runtime Library Routines ....................... 203
B.1 C/C++ Stub routines ...................................... 204
B.2 Fortran Stub Routines .................................... 209

C. OpenMP C and C++ Grammar ................................ 215
C.1 Notation .................................................. 215
C.2 Rules ..................................................... 216

D. Interface Declarations ..................................... 223
D.1 Example of the omp.h Header File ....................... 223
D.2 Example of an Interface Declaration include File .... 225
D.3 Example of a Fortran 90 Interface Declaration module . 227
D.4 Example of a Generic Interface for a Library Routine . 232
E. Implementation Defined Behaviors in OpenMP .................................. 233

F. Changes from Version 2.0 to Version 2.5 ............................................. 237
CHAPTER 1

Introduction

This document specifies a collection of compiler directives, library routines, and
environment variables that can be used to specify shared-memory parallelism in C, C++
and Fortran programs. This functionality collectively defines the specification of the
OpenMP Application Program Interface (OpenMP API). This specification provides a
model for parallel programming that is portable across shared memory architectures
from different vendors. Compilers from numerous vendors support the OpenMP API.
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
allow users to create and manage parallel programs while permitting portability. The
directives extend the C, C++ and Fortran base languages with single program multiple
data (SPMD) constructs, work-sharing constructs, and synchronization constructs, and
they provide support for the sharing and privatization of data. The functionality to
control the runtime environment is provided by library routines and environment
variables. Compilers that support the OpenMP API often include a command line option
to the compiler that activates and allows interpretation of all OpenMP directives.

1.1 Scope

The OpenMP API covers only user-directed parallelization, wherein the user explicitly
specifies the actions to be taken by the compiler and runtime system in order to execute
the program in parallel. OpenMP-compliant implementations are not required to check
for dependencies, conflicts, deadlocks, race conditions, or other problems that result
from non-conforming programs. The user is responsible for using OpenMP in his
application to produce a conforming program. OpenMP does not cover compiler-
generated automatic parallelization and directives to the compiler to assist such
parallelization.
1.2 Glossary

1.2.1 Threading Concepts

thread An execution entity having a serial flow of control and an associated stack.

thread-safe routine A routine that performs the intended function even when executed concurrently (by more than one thread).

1.2.2 OpenMP language terminology

base language A programming language that serves as the foundation of the OpenMP specification.

COMMENT: Current base languages for OpenMP are C90, C99, C++, Fortran 77, Fortran 90, and Fortran 95.

original program A program written in a base language.

structured block For C/C++, an executable statement, possibly compound, with a single entry at the top and a single exit at the bottom.

For Fortran, a block of executable statements with a single entry at the top and a single exit at the bottom.

COMMENTS: For both languages, the point of entry cannot be a labeled statement and the point of exit cannot be a branch of any type.

For C/C++:

• The point of entry cannot be a call to setjmp().
• longjmp() and throw() must not violate the entry/exit criteria.
• Calls to exit() are allowed in a structured block.
• An expression statement, iteration statement, selection statement, or try block is considered to be a structured block if the corresponding compound statement obtained by enclosing it in { and } would be a structured block.
For Fortran:

- **STOP** statements are allowed in a *structured block*.

**OpenMP directive**  
In C/C++, a `#pragma` and in Fortran, a comment, that specifies OpenMP program behavior.

COMMENT: See Section 2.1 on page 18 for a description of OpenMP directive syntax.

**white space**  
A non-empty sequence of space and/or horizontal tab characters.

**OpenMP program**  
A program that consists of an *original program*, annotated with *OpenMP directives*.

**declarative directive**  
An *OpenMP directive* that may only be placed in a declarative context. A declarative directive has no associated executable user code, but instead has one or more associated user declarations.

COMMENT: Only the `threadprivate` directive is a declarative directive.

**executable directive**  
An *OpenMP directive* that is not declarative, i.e., it may be placed in an executable context.

COMMENT: All directives except the `threadprivate` directive are executable directives.

**standalone directive**  
An OpenMP executable directive that has no associated executable user code.

COMMENT: Only the `barrier` and `flush` directives are standalone directives.

**simple directive**  
An OpenMP executable directive whose associated user code must be a simple (single, non-compound) executable statement.

COMMENT: Only the `atomic` directive is a simple directive.

**loop directive**  
An OpenMP executable directive whose associated user code must be a loop that is a *structured block*.

COMMENTS:

For C/C++, only the `for` directive is a loop directive.

For Fortran, only the `do` directive and the optional `end do` directive are loop directives.
structured directive

An OpenMP executable directive that is neither a standalone directive, a simple directive nor a loop directive.

For C/C++, all structured directives have associated user code that is the following structured block.

For Fortran, all structured directives are paired with an associated end directive except section, whose end is marked either by the next section or by the end sections. These structured directives bracket the associated user code that forms a structured block.

construct

An OpenMP executable directive (and for Fortran, the paired end directive, if any) and the associated statement, loop or structured block, if any, not including the code in any called routines, i.e., the lexical extent of an executable directive.

region

All code encountered during a specific instance of the execution of a given construct or OpenMP library routine. A region includes any code in called routines as well as any implicit code introduced by the OpenMP implementation.

COMMENTS:

A region may also be thought of as the dynamic or runtime extent of a construct or OpenMP library routine.

During the execution of an OpenMP program, a construct may give rise to many regions.

sequential part

All code encountered during the execution of an OpenMP program that is not enclosed by a parallel region corresponding to an explicit parallel construct.

COMMENTS:

The sequential part executes as if it were enclosed by an inactive parallel region called the implicit parallel region.

Executable statements in called routines may be in both the sequential part and any number of explicit parallel regions at different points in the program execution.

nested construct

A construct (lexically) enclosed by another construct.

nested region

A region (dynamically) enclosed by another region, i.e., a region executed in its entirety during another region.

COMMENT: Some nestings are conforming and some are not. See Section 2.9 on page 87 for the rules specifying the conforming nestings.
closely nested region  A region nested inside another region with no parallel region nested between them.

binding thread set  The set of threads that are affected by, or provide the context for, the execution of a region.

The binding thread set for a given region can be one of the following:

- all threads - all threads participating in the OpenMP program.
- current team - all the threads in the team executing the innermost enclosing parallel region.
- encountering thread - the thread whose execution encountered the construct giving rise to the region.

COMMENTS:

- The binding thread set for critical and atomic regions is all threads.
- The binding thread set for parallel and flush regions is the encountering thread.
- The binding thread set for all other regions arising from a construct is the current team.
- The binding thread set for the OpenMP library routines omp_get_num_threads and omp_get_thread_num is the current team.
- The binding thread set for the OpenMP library routines omp_set_num_threads, omp_get_max_threads, omp_set_nested, omp_get_nested, omp_set_dynamic, omp_get_dynamic, omp_get_wtime and omp_get_wtick is the encountering thread.
- The binding thread set for all other OpenMP library routines is all threads.
binding region  For a region whose binding thread set is the current team, the enclosing region
that determines the execution context and limits the scope of the effects of the
bound region.

Binding region is not defined for regions whose binding thread set is all
threads or the encountering thread.

COMMENTS:

The binding region for an ordered region is the innermost enclosing
loop region.

For all other regions with whose binding thread set is the current team,
the binding region is the innermost enclosing parallel region.
When such a region is encountered outside of any explicit parallel
region, the binding region is the implicit parallel region enclosing
the sequential part.

A parallel region need not be active to be a binding region.

A region never binds to any region outside of the innermost enclosing
parallel region.

orphaned construct  A construct that gives rise to a region whose binding thread set is the current
team, but that is not nested within another construct giving rise to the binding
region.

worksharing construct  A construct that defines units of work, each of which is executed exactly once
by a thread in the team executing the construct.

For C, worksharing constructs are for, sections, and single.

For Fortran, worksharing constructs are do, sections, single and
workshare.

active parallel region  A parallel region whose if clause evaluates to true.

COMMENT: A missing if clause is equivalent to an if clause that evaluates
to true.

inactive parallel region  A parallel region that is not an active parallel region, i.e., a serialized
parallel region.

An inactive parallel region is always executed by a team of only one thread.

implicit parallel region  The inactive parallel region that encloses the sequential part of an OpenMP
program.
**initial thread**  The thread that executes the sequential part.

**master thread**  A thread that encounters (the start of) a parallel region and creates a team.

**team**  A set of one or more threads participating in the execution of a parallel region.

For an active parallel region, the team comprises the master thread and additional threads that may be launched.

For an inactive parallel region, the team only includes the master thread.

**barrier**  A point in the execution of a program encountered by a team, beyond which no thread in the team may execute until all threads in the team have reached that point.

## 1.2.3 Data Terminology

**variable**  A named data object, whose value can be defined and redefined during the execution of a program.

Only an object that is not part of another object is considered a variable. For example, array elements, structure components, array sections and substrings are not considered variables.

**private variable**  A variable whose name provides access to a different block of storage for each thread in a team.

**shared variable**  A variable whose name provides access to the same block of storage for all threads in a team.

**global-lifetime memory**  Memory locations that persist during the entire execution of the original program, according to the base language specification.

**threadprivate memory**  Global-lifetime memory locations that are replicated, one per thread, by the OpenMP implementation.
defined For variables, the property of having a valid value.

For C:

For the contents of variables, the property of having a valid value.

For C++:

For the contents of variables of POD (plain old data) type, the property of having a valid value.

For variables of non-POD class type, the property of having been constructed but not subsequently destructed.

For Fortran:

For the contents of variables, the property of having a valid value. For the allocation or association status of variables, the property of having a valid status.

COMMENT: Programs that rely upon variables that are not defined are non-conforming programs.

1.2.4 Implementation Terminology

supporting n levels of parallelism Implies allowing an active parallel region to be enclosed by \( n-1 \) active parallel regions, where the team associated with each active parallel region has more than one thread.

supporting OpenMP Supporting at least one level of parallelism.

supporting nested parallelism Supporting more than one level of parallelism.

conforming program An OpenMP program that follows all the rules and restrictions of the OpenMP specification.

compliant implementation An implementation of the OpenMP specification that compiles and executes any conforming program as defined by the specification.

COMMENT: A compliant implementation may exhibit unspecified behavior when compiling or executing a non-conforming program.
unspecified behavior A behavior or result that is not specified by the OpenMP specification or not known prior to the compilation or execution of an OpenMP program.

Such unspecified behavior may result from:

- Issues documented by the OpenMP specification as having unspecified behavior.
- A non-conforming program.
- A conforming program exhibiting an implementation defined behavior.

implementation defined Behavior that is allowed to vary among different compliant implementations, but must be documented by the implementation. An implementation is allowed to define this behavior as unspecified.

COMMENT: All such features are documented in Appendix E.

1.3 Execution Model

The OpenMP API uses the fork-join model of parallel execution. Although this fork-join model can be useful for solving a variety of problems, it is somewhat tailored for large array-based applications. OpenMP is intended to support programs that will execute correctly both as parallel programs (multiple threads of execution and a full OpenMP support library) and as sequential programs (directives ignored and a simple OpenMP stubs library). However, it is possible and permitted to develop a program that executes correctly as a parallel program but not as a sequential program, or that produces different results when executed as a parallel program, compared to when it is executed as a sequential program. Furthermore, using different numbers of threads may result in different numeric results because of changes in the association of numeric operations. For example, a serial addition reduction may have a different pattern of addition associations than a parallel reduction. These different associations may change the results of floating-point addition.

An OpenMP program begins as a single thread of execution, called the initial thread. The initial thread executes sequentially, as if enclosed in an implicit inactive parallel region surrounding the whole program.

When any thread encounters a parallel construct, the thread creates a team of itself and zero or more additional threads and becomes the master of the new team. All members of the new team execute the code inside the parallel construct. There is an implicit barrier at the end of the parallel construct. Only the master thread continues execution of user code beyond the end of the parallel construct. Any number of parallel constructs can be specified in a single program.
parallel regions may be arbitrarily nested inside each other. If nested parallelism is
disabled, or is not supported by the OpenMP implementation, then the new team that is
created by a thread encountering a parallel construct inside a parallel region
will consist only of the encountering thread. However, if nested parallelism is supported
and enabled, then the new team can consist of more than one thread.

When any team encounters a work-sharing construct, the work inside the construct is
divided among the members of the team and executed co-operatively instead of being
executed by every thread. There is an optional barrier at the end of work-sharing
constructs. Execution of code by every thread in the team resumes after the end of the
work-sharing construct.

Synchronization constructs and library routines are available in OpenMP to co-ordinate
threads and data in parallel and work-sharing constructs. In addition, library
routines and environment variables are available to control or query the runtime
environment of OpenMP programs.

OpenMP makes no guarantee that input or output to the same file is synchronous when
executed in parallel. In this case, the programmer is responsible for synchronizing input
and output statements (or routines) using the provided synchronization constructs or
library routines. For the case where each thread accesses a different file, no
synchronization by the programmer is necessary.

1.4 Memory Model

1.4.1 Structure of the OpenMP Memory Model

OpenMP provides a relaxed-consistency, shared-memory model. All OpenMP threads
have access to a place to store and retrieve variables, called the memory. In addition,
each thread is allowed to have its own temporary view of the memory. The temporary
view of memory for each thread is not a required part of the OpenMP memory model,
but can represent any kind of intervening structure, such as machine registers, cache, or
other local storage, between the thread and the memory. The temporary view of memory
allows the thread to cache variables and thereby avoid going to memory for every
reference to a variable. Each thread also has access to another type of memory that must
not be accessed by other threads, called threadprivate memory.

A parallel directive determines two kinds of access to variables used in the
associated structured block: shared and private. Each variable referenced in the
structured block has an original variable, which is the variable by the same name that
exists in the program immediately outside the parallel construct. Each reference to a
shared variable in the structured block becomes a reference to the original variable. For
each private variable referenced in the structured block, a new version of the original
variable (of the same type and size) is created in memory for each thread of the team
formed to execute the parallel region associated with the parallel directive,
except possibly for the master thread of the team. References to a private variable in the
structured block refer to the current thread’s private version of the original variable.

If multiple threads write to the same shared variable without synchronization, the
resulting value of the variable in memory is unspecified. If at least one thread reads from
a shared variable and at least one thread writes to it without synchronization, the value
seen by any reading thread is unspecified.

It is implementation defined as to whether, and in what sizes, memory accesses by
multiple threads to the same variable without synchronization are atomic with respect to
each other.

A private variable in an outer parallel region belonging to, or accessible from, a
thread that eventually becomes the master thread of an inner nested parallel region,
is permitted to be accessed by any of the threads of the team executing the inner
parallel region, unless the variable is also private with respect to the inner
parallel region. Any other access by one thread to the private variables of another
thread results in unspecified behavior.

1.4.2 The Flush Operation

The memory model has relaxed-consistency because a thread’s temporary view of
memory is not required to be consistent with memory at all times. A value written to a
variable can remain in the thread’s temporary view until it is forced to memory at a later
time. Likewise, a read from a variable may retrieve the value from the thread’s
temporary view, unless it is forced to read from memory. The OpenMP flush operation
enforces consistency between the temporary view and memory.

The flush operation is applied to a set of variables called the flush-set. The flush
operation restricts reordering of memory operations that an implementation might
otherwise do. Implementations must not reorder the code for a memory operation for a
given variable, or the code for a flush operation for the variable, with respect to a flush
operation that refers to the same variable.

If a thread has captured the value of a write in its temporary view of a variable since its
last flush of that variable, then when it executes another flush of the variable, the flush
does not complete until the value of the variable has been written to the variable in
memory. A flush of a variable executed by a thread also causes its temporary view of the
variable to be discarded, so that if its next memory operation for that variable is a read,
then the thread will read from memory and may capture the value in the temporary view.
When a thread executes a flush, no later memory operation by that thread for a variable involved in that flush is allowed to start until the flush completes. The completion of a flush of a set of variables executed by a thread is defined as the point at which all writes to those variables done by that thread are visible in memory to all other threads and the temporary view, for that thread, of all variables involved, is discarded.

The flush operation provides a guarantee of consistency between a thread’s temporary view and memory. Therefore, the flush operation can be used to guarantee that a value written to a variable by one thread may be read by a second thread. To accomplish this, the programmer must ensure that the second thread has not written to the variable since its last flush of the variable, and that the following sequence of events happens in the specified order:

1. The value is written to the variable by the first thread.
2. The variable is flushed by the first thread.
3. The variable is flushed by the second thread.
4. The value is read from the variable by the second thread.

The volatile keyword in the C and C++ languages specifies a consistency mechanism that is related to the OpenMP memory consistency mechanism in the following way: a reference that reads the value of an object with a volatile-qualified type behaves as if there were a flush operation on that object at the previous sequence point, while a reference that modifies the value of an object with a volatile-qualified type behaves as if there were a flush operation on that object at the next sequence point.

### 1.4.3 OpenMP Memory Consistency

The type of relaxed memory consistency provided by OpenMP is similar to weak ordering. OpenMP does not apply restrictions to the reordering of memory operations executed by a single thread except for those related to a flush operation.

The restrictions in Section 1.4.2 on page 11 on reordering with respect to flush operations guarantee the following:

- If the intersection of the flush-sets of two flushes performed by two different threads is non-empty, then the two flushes must be completed as if in some sequential order, seen by all threads.

---

1. Weak ordering is described in S. V. Adve and K. Gharachorloo, “Shared Memory Consistency Models: A Tutorial”, IEEE Computer, 29(12), pp 86-76, December 1996. Weak ordering requires that some memory operations be defined as synchronization operations and that these be ordered with respect to each other. In the context of OpenMP, two flushes of the same variable are synchronization operations. The OpenMP memory model is slightly weaker than weak ordering, however, because flushes whose flush-sets have an empty intersection are not ordered with respect to each other.
13

- If the intersection of the flush-sets of two flushes performed by one thread is non-empty, then the two flushes must appear to be completed in that thread’s program order.

- If the intersection of the flush-sets of two flushes is empty, the threads can observe these flushes in any order.

The flush operation can be specified using the flush directive, and is also implied at various locations in an OpenMP program: see Section 2.7.5 on page 58 for details. For an example illustrating the memory model, see Section A.2 on page 120.

1.5 OpenMP Compliance

An implementation of the OpenMP API is compliant if and only if it compiles and executes all conforming programs according to the syntax and semantics laid out in Chapters 1, 2, 3 and 4. Appendices A, B, C, D, E and F and sections designated as Notes (see Section 1.7 on page 14) are for information purposes only and are not part of the specification.

The OpenMP API defines constructs that operate in the context of the base language that is supported by an implementation. If the base language does not support a language construct that appears in this document, a compliant OpenMP implementation is not required to support it, with the exception that for Fortran, the implementation must allow case insensitivity for directive and API routines names, and must allow identifiers of more than six characters.

All library, intrinsic and built-in routines provided by the base language must be thread-safe in a compliant implementation. In addition, the implementation of the base language must also be thread-safe (e.g., ALLOCATE and DEALLOCATE statements must be thread-safe in Fortran). Unsynchronized concurrent use of such routines by different threads must produce correct results (though not necessarily the same as serial execution results, as in the case of random number generation routines).

In both Fortran 90 and Fortran 95, variables with explicit initialization have the SAVE attribute implicitly. This is not the case in Fortran 77. However, a compliant OpenMP Fortran implementation must give such a variable the SAVE attribute, regardless of the underlying base language version.

Appendix E lists certain aspects of the OpenMP API that are implementation-defined. A compliant implementation is required to define and document its behavior for each of the items in Appendix E.
1.6 Normative References

  This OpenMP API specification refers to ISO/IEC 9899:1990 as C90.

  This OpenMP API specification refers to ISO/IEC 9899:1999 as C99.

  This OpenMP API specification refers to ISO/IEC 14882:1998 as C++.

  This OpenMP API specification refers to ISO/IEC 1539:1980 as Fortran 77.

  This OpenMP API specification refers to ISO/IEC 1539:1991 as Fortran 90.

  This OpenMP API specification refers to ISO/IEC 1539-1:1997 as Fortran 95.

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

1.7 Organization of this document

The remainder of this document is structured as follows:

- Chapter 2: Directives
- Chapter 3: Runtime Library Routines
• Chapter 4: Environment Variables
• Appendix A: Examples
• Appendix B: Stubs for Runtime Library Routines
• Appendix C: OpenMP C and C++ Grammar
• Appendix D: Interface Declarations
• Appendix E: Implementation Defined Behaviors in OpenMP
• Appendix F: Changes from Version 2.0 to Version 2.5

Some sections of this document only apply to programs written in a certain base language. Text that applies only to programs whose base language is C or C++ is shown as follows:

\[ \text{\textbackslash end{C/C++} specific text} \]

Text that applies only to programs whose base language is Fortran is shown as follows:

\[ \text{\textbackslash end{Fortran} specific text} \]

Where an entire page consists of, for example, Fortran specific text, a marker is shown at the top of the page like this:

\[ \text{\textbackslash end{Fortran} (cont.)} \]

Some text is for information only, and is not part of the normative specification. Such text is designated as a note, like this:

\[ \text{\textbackslash end{Note} – Non-normative text} \]
CHAPTER 2

Directives

This chapter describes the syntax and behavior of OpenMP directives, and is divided into the following sections:

- The language-specific directive format (Section 2.1 on page 18)
- Mechanisms to control conditional compilation (Section 2.2 on page 21)
- Control of OpenMP API internal control variables (Section 2.3 on page 24)
- Details of each OpenMP directive (Section 2.4 on page 26 to Section 2.9 on page 87)

In C/C++, OpenMP directives are specified by using the `#pragma` mechanism provided by the C and C++ standards.

In Fortran, OpenMP directives are specified by using special comments that are identified by unique sentinels. Also, a special comment form is available for conditional compilation.

Compilers can therefore ignore OpenMP directives and conditionally compiled code if support of OpenMP is not provided or enabled. A compliant implementation must provide an option or interface that ensures that underlying support of all OpenMP directives and OpenMP conditional compilation mechanisms is enabled. In the remainder of this document, the phrase OpenMP compilation is used to mean a compilation with these OpenMP features enabled.

Restrictions

The following restriction applies to all OpenMP directives:
2.1 Directive Format

C/C++

OpenMP directives for C/C++ are specified with the `#pragma` preprocessing directive. The syntax of an OpenMP directive is formally specified by the grammar in Appendix C, and informally as follows:

```
#pragma omp directive-name [clause[ [,] clause]...] new-line
```

Each directive starts with `#pragma omp`. The remainder of the directive follows the conventions of the C and C++ standards for compiler directives. In particular, white space can be used before and after the `#`, and sometimes white space must be used to separate the words in a directive. Preprocessing tokens following the `#pragma omp` are subject to macro replacement.

Directives are case-sensitive.

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

Fortran

OpenMP directives for Fortran are specified as follows:

```
sentinel directive-name [clause[ [,] clause]...] new-line
```

All OpenMP compiler directives must begin with a directive `sentinel`. The format of a sentinel differs between fixed and free-form source files, as described in Section 2.1.1 on page 19 and Section 2.1.2 on page 20.

Directives are case-insensitive. Directives cannot be embedded within continued statements, and statements cannot be embedded within directives.

In order to simplify the presentation, free form is used for the syntax of OpenMP directives for Fortran in the remainder of this document, except as noted.
Only one directive-name can be specified per directive (note that this includes combined directives, see Section 2.6 on page 46). The order in which clauses appear on directives is not significant. Clauses on directives may be repeated as needed, subject to the restrictions listed in the description of each clause.

Some data-sharing attribute clauses (Section 2.8.3 on page 70), data copying clauses (Section 2.8.4 on page 83), the threadprivate directive (Section 2.8.2 on page 66), and the flush directive (Section 2.7.5 on page 58) accept a list. A list consists of a comma-separated collection of one or more list items.

A list item is a variable name, subject to the restrictions specified in each of the sections describing clauses and directives for which a list appears.

A list item is a variable name or common block name (enclosed in slashes), subject to the restrictions specified in each of the sections describing clauses and directives for which a list appears.

### 2.1.1 Fixed Source Form Directives

The following sentinels are recognized in fixed form source files:

!$omp | c$omp | *$omp

Sentinels must start in column 1 and appear as a single word with no intervening characters. Fortran fixed form line length, white space, continuation, and column rules apply to the directive line. Initial directive lines must have a space or zero in column 6, and continuation directive lines must have a character other than a space or a zero in column 6.

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 non-blank character after the directive sentinel of an initial or continuation directive line is an exclamation point, the line is ignored.

Note – in the following example, the three formats for specifying the directive are equivalent (the first line represents the position of the first 9 columns):
2.1.2 Free Source Form Directives

The following sentinel is recognized in free form source 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 character. Fortran free form line length, 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. Continued directive lines can have an ampersand after the directive sentinel with optional white space before and after the ampersand.

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.

One or more blanks or horizontal 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
```
Note – in the following example the three formats for specifying the directive are equivalent (the first line represents the position of the first 9 columns):

```fortran
!23456789
!$omp parallel do &
    !$omp shared(a,b,c)

!$omp parallel &
    !$omp&do shared(a,b,c)

!$omp paralleldo shared(a,b,c)
```

2.2 Conditional Compilation

In implementations that support a preprocessor, the `_OPENMP` macro name is defined to have the decimal value `yyyyymm` where `yyyy` and `mm` are the year and month designations of the version of the OpenMP API that the implementation supports.

If this macro is the subject of a `#define` or a `#undef` preprocessing directive, the behavior is unspecified.

For examples of conditional compilation, see Section A.3 on page 122.
The OpenMP API requires Fortran lines to be compiled conditionally, as described in the following sections.

### 2.2.1 Fixed Source Form Conditional Compilation Sentinels

The following conditional compilation sentinels are recognized in fixed form source files:

```fortran
!$ 10 iam = omp_get_thread_num() + $c$
```

To enable conditional compilation, a line with a conditional compilation sentinel must satisfy the following criteria:

- The sentinel must start in column 1 and appear as a single word with no intervening white space.
- 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 met, the sentinel is replaced by two spaces. If these criteria are not met, the line is left unchanged.

---

**Note** — in the following example, the two forms for specifying conditional compilation in fixed source form are equivalent (the first line represents the position of the first 9 columns):

```
c23456789
!$ 10 iam = omp_get_thread_num() +
!$   &          index
```

```c
#define _OPENMP

10 iam = omp_get_thread_num() +
   & index
```
2.2.2 Free Source Form Conditional Compilation Sentinel

The following conditional compilation sentinel is recognized in free form source files:

```
!$
```

To enable conditional compilation, a line with a conditional compilation sentinel must satisfy the following criteria:

- The sentinel can appear in any column but must be preceded only by white space.
- The sentinel must appear as a single word with no intervening white space.
- 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. (Continued lines can have an ampersand after the sentinel, with optional white space before and after the ampersand.)

If these criteria are met, the sentinel is replaced by two spaces. If these criteria are not met, the line is left unchanged.

**Note** – in the following example, the two forms for specifying conditional compilation in free source form are equivalent (the first line represents the position of the first 9 columns):

```
c23456789
!$ iam = omp_get_thread_num() + &
!$& index

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

Fortran
2.3 Internal Control Variables

An OpenMP implementation must act as if there were internal control variables that store the information for determining the number of threads to use for a parallel region and how to schedule a work-sharing loop. The control variables are given values at various times (described below) during execution of an OpenMP program. They are initialized by the implementation itself and may be given values by using OpenMP environment variables, and by calls to OpenMP API routines. The only way for the program to retrieve the values of these control variables is by calling OpenMP API routines.

For purposes of exposition, this document refers to the control variables by certain names (below), but an implementation is not required to use these names or to offer any way to access the variables other than through the ways shown in Table 2.1.

The following control variables store values that affect the operation of parallel regions:

- $nthreads-var$ - stores the number of threads requested for future parallel regions.
- $dyn-var$ - controls whether dynamic adjustment of the number of threads to be used for future parallel regions is enabled.
- $nest-var$ - controls whether nested parallelism is enabled for future parallel regions.

The following control variables store values that affect the operation of loop regions:

- $run-sched-var$ - stores scheduling information to be used for loop regions using the runtime schedule clause.
- $def-sched-var$ - stores implementation defined default scheduling information for loop regions.

Table 2-1 shows the methods for modifying and retrieving the values of each control variable, as well as their initial values.

<table>
<thead>
<tr>
<th>Control variable</th>
<th>Ways to modify value</th>
<th>Way to retrieve value</th>
<th>Initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$nthreads-var$</td>
<td>OMP_NUM_THREADS</td>
<td>omp_get_max_threads()</td>
<td>Implementation defined</td>
</tr>
<tr>
<td></td>
<td>omp_set_num_threads()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$dyn-var$</td>
<td>OMP_DYNAMIC</td>
<td>omp_get_dynamic()</td>
<td>Implementation defined</td>
</tr>
<tr>
<td></td>
<td>omp_set_dynamic()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$nest-var$</td>
<td>OMP_NESTED</td>
<td>omp_get_nested()</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>omp_set_nested()</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$run-sched-var$</td>
<td>OMP_SCHEDULE</td>
<td>(none)</td>
<td>Implementation defined</td>
</tr>
<tr>
<td></td>
<td>(none)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$def-sched-var$</td>
<td></td>
<td>(none)</td>
<td>Implementation defined</td>
</tr>
</tbody>
</table>

Table 2-1  Control variables
The effect of the API routines in Table 2-1 on the internal control variables described in this specification applies only during the execution of the sequential part of the program. During execution of the sequential part, only one copy of each internal control variable may exist. The effect of these API routines on the internal control variables is implementation defined when the API routines are executed from within any explicit parallel region. Additionally, the number of copies of the internal control variables, and their effects, during the execution of any explicit parallel region are implementation defined.

The internal control variables are each given values before any OpenMP construct or OpenMP API routine executes. The initial values of nthreads-var, dyn-var, run-sched-var, and def-sched-var are implementation defined. The initial value of nest-var is false. After the initial values are assigned, but also before any OpenMP construct or OpenMP API routine executes, the values of any OpenMP environment variables that were set by the user are read and the associated control variables are modified accordingly. After this point, no changes to any OpenMP environment variables will be reflected in the control variables. During execution of the user’s code, certain control variables can be further modified by certain OpenMP API routine calls. An OpenMP construct clause does not modify the value of any of these control variables.

Table 2-2 shows the override relationships between various construct clauses, OpenMP API routines, environment variables, and initial values.

**TABLE 2-2** Override relationships

<table>
<thead>
<tr>
<th>construct clause, if used</th>
<th>...overrides previous call to OpenMP API routine</th>
<th>...overrides environment variable, if set</th>
<th>...overrides initial value</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_threads clause</td>
<td><code>omp_set_num_threads()</code></td>
<td><code>OMP_NUM_THREADS</code></td>
<td>initial value of nthreads-var</td>
</tr>
<tr>
<td>(none)</td>
<td><code>omp_set_dynamic()</code></td>
<td><code>OMP_DYNAMIC</code></td>
<td>initial value of dyn-var</td>
</tr>
<tr>
<td>(none)</td>
<td><code>omp_set_nested()</code></td>
<td><code>OMP_NESTED</code></td>
<td>initial value of nest-var</td>
</tr>
<tr>
<td>(none)</td>
<td>(none)</td>
<td><code>OMP_SCHEDULE</code> (only used when schedule kind is runtime)</td>
<td>initial value of run-sched-var</td>
</tr>
<tr>
<td>schedule clause</td>
<td>(none)</td>
<td>(none)</td>
<td>initial value of def-sched-var</td>
</tr>
</tbody>
</table>

**Cross References:**
- `parallel` construct, see Section 2.4 on page 26.
- Loop construct, see Section 2.5.1 on page 33.
- `omp_set_num_threads` routine, see Section 3.2.1 on page 91.
- `omp_set_dynamic` routine, see Section 3.2.7 on page 97.
parallel Construct

Summary
This is the fundamental construct that starts parallel execution. See Section 1.3 on page 9 for a general description of the OpenMP execution model.

Syntax

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

where clause is one of the following:

```
        if ( scalar-expression )
        private ( list )
        firstprivate ( list )
        default ( shared | none )
        shared ( list )
        copyin ( list )
```
The syntax of the parallel construct is as follows:

```c
reduction (operator : list)
num_threads (integer-expression)
```

The parallel construct is as follows:

```fortran
!$omp parallel [clause[,...] clause]...
structured-block
!$omp end parallel
```

where clause is one of the following:

```c
if (scalar-logical-expression)
private (list)
firstprivate (list)
default (private | shared | none)
shared (list)
copyin (list)
reduction ({operator | intrinsic_procedure_name} : list)
num_threads (scalar-integer-expression)
```

The end parallel directive denotes the end of the parallel construct.

### Binding

The binding thread set for a parallel region is the encountering thread. The
encountering thread becomes the master thread of the new team.

### Description

When a thread encounters a parallel construct, a team of threads is created to
execute the parallel region (see Section 2.4.1 on page 29 for more information about
how the number of threads in the team is determined, including the evaluation of the if
and num_threads clauses). The thread which encountered the parallel construct
becomes the master thread of the new team, with a thread number of zero for the duration of the `parallel` region. All threads in the new team, including the master thread, execute the region. Once the team is created, the number of threads in the team remains constant for the duration of that `parallel` region.

Within 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. A thread may obtain its own thread number by a call to the `omp_get_thread_num` library routine.

The structured block of the `parallel` construct is executed by each thread, although each thread can execute a path of statements that is different from the other threads.

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

If a thread in a team executing a `parallel` region encounters another `parallel` directive, it creates a new team, according to the rules in Section 2.4.1 on page 29, and it becomes the master of that new team.

If execution of a thread terminates while inside a `parallel` region, execution of all threads in all teams terminates. The order of termination of threads is unspecified. All the work done by a team prior to any barrier which the team has passed in the program is guaranteed to be complete. The amount of work done by each thread after the last barrier that it passed and before it terminates is unspecified.

For an example of the `parallel` construct, see Section A.4 on page 123. For an example of the `num_threads` clause, see Section A.5 on page 125.

**Restrictions**

Restrictions to the `parallel` construct are as follows:

- A program which branches into or out of a `parallel` region is non-conforming.
- A program must not depend on any ordering of the evaluations of the clauses of the `parallel` directive, or on any side effects of the evaluations of the clauses.
- At most one `if` clause can appear on the directive.
- At most one `num_threads` clause can appear on the directive. The `num_threads` expression must evaluate to a positive integer value.

- A `throw` executed inside a `parallel` region must cause execution to resume within the same `parallel` region, and it must be caught by the same thread that threw the exception.
Chapter 2  Directives

2.4.1 Determining the Number of Threads for a parallel Region

When execution encounters a parallel directive, the value of the if clause or num_threads clause (if any) on the directive, the current parallel context, the number of levels of parallelism supported, and the values of the nthreads-var, dyn-var and nest-var internal control variables are used to determine the number of threads to use in the region. Figure 2-1 describes how the number of threads is determined. The if clause expression and the num_threads clause expression are evaluated in the context outside of the parallel construct, and no ordering of those evaluations is specified. It is also unspecified whether, in what order, or how many times any side-effects of the evaluation of the num_threads or if clause expressions occur.

When a thread executing inside an active parallel region encounters a parallel construct, the new team which is created will consist of only the encountering thread, when any of the following conditions hold:

• nested parallelism is disabled,
• the if clause expression evaluates to false, or
• no further levels of parallelism are supported by the OpenMP implementation.

However, if nested parallelism is enabled and additional levels of parallelism are supported, then the new team can consist of more than one thread.

The number of levels of parallelism supported is implementation defined. If only one level of parallelism is supported (i.e. nested parallelism is not supported) then the value of the nest-var internal control variable is always false.

If dynamic adjustment of the number of threads is enabled, the number of threads that are used for executing subsequent parallel regions may be adjusted automatically by the implementation. Once the number of threads is determined, it remains fixed for the
duration of that `parallel` region. If dynamic adjustment of the number of threads is
disabled, the number of threads that are used for executing subsequent `parallel`
regions may not be adjusted by the implementation.

It is implementation defined whether the ability to dynamically adjust the number of
threads is provided. If this ability is not provided, then the value of the `dyn-var` internal
control variable is always `false`.

Implementations may deliver fewer threads than indicated in Figure 2-1, in exceptional
situations, such as when there is a lack of resources, even if dynamic adjustment is
disabled. In these exceptional situations the behavior of the program is implementation
defined: this may, for example, include interrupting program execution.

**Note** – Since the initial value of the `dyn-var` internal control variable is implementation
defined, programs that depend on a specific number of threads for correct execution
should explicitly disable dynamic adjustment of the number of threads.
FIGURE 2-1 Determining the number of threads for a parallel region. Note that no ordering of evaluation of the if and num_threads clauses is implied.
2.5 Work-sharing Constructs

A work-sharing construct distributes the execution of the associated region among the members of the team that encounters it. A work-sharing region must bind to an active parallel region in order for the work-sharing region to execute in parallel. If execution encounters a work-sharing region in the sequential part, it is executed by the initial thread.

A work-sharing construct does not launch new threads, and a work-sharing region has no barrier on entry. However, an implied barrier exists at the end of the work-sharing region, unless a nowait clause is specified. If a nowait clause is present, an implementation may omit code to synchronize the threads at the end of the work-sharing region. In this case, threads that finish early may proceed straight to the instructions following the work-sharing region without waiting for the other members of the team to finish the work-sharing region, and without performing a flush operation (see Section A.8 on page 128 for an example.)

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

- loop construct
- sections construct
- single construct
- workshare construct

Restrictions

The following restrictions apply to work-sharing constructs:

- Each work-sharing region must be encountered by all threads in a team or by none at all.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.
2.5.1 Loop Construct

Summary

The loop construct specifies that the iterations of the associated loop will be executed in parallel. The iterations of the loop are distributed across threads that already exist in the team executing the parallel region to which the loop region binds.

Syntax

C/C++

The syntax of the loop construct is as follows:

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

The clause is one of the following:

private(list)
firstprivate(list)
lastprivate(list)
reduction(operator: list)
ordered
schedule(kind[, chunk_size])
nowait
The `for` directive places restrictions on the structure of the corresponding `for-loop`. Specifically, the corresponding `for-loop` must have the following canonical form:

```
for (init-expr; var relational-op b; incr-expr) statement
```

- `init-expr`: One of the following:
  - `var = lb`
  - `integer-type var = lb`
- `incr-expr`: One of the following:
  - `++var`
  - `var++`
  - `--var`
  - `var--`
  - `var += incr`
  - `var -= incr`
  - `var = var + incr`
  - `var = incr + var`
  - `var = var - incr`
- `var`: A signed integer variable, of type `integer-type`, as defined in the base language. If this variable would otherwise be shared, it is implicitly made private on the loop construct. This variable must not be modified during the execution of the `for-loop` other than in `incr-expr`. Unless the variable is specified `lastprivate` on the loop construct, its value after the loop is undefined.
- `relational-op`: One of the following:
  - `<`
  - `<=`
  - `>`
  - `>=`
- `lb`, `b`, and `incr`: Loop invariant integer expressions. There is no implied synchronization during the evaluation of these expressions. It is unspecified whether, in what order, or how many times any side effects within the `lb`, `b`, or `incr` expressions occur.
- `statement`: Defined according to the base language.

Note that the canonical form allows the number of loop iterations to be computed on entry to the loop. This computation is performed with values in the type of `var`, after integral promotions. In particular, if the value of `b - lb + incr`, or any intermediate result required to compute this value, cannot be represented in that type, the behavior is unspecified.
The syntax of the loop construct is as follows:

```
$omp do [clause[[, clause] ... ]
  do-loop
  !$omp end do [nowait ]
```

The `clause` is one of the following:

- `private( list )`
- `firstprivate( list )`
- `lastprivate( list )`
- `reduction( { operator | intrinsic_procedure_name } : list )`
- `ordered`
- `schedule( kind[ , chunk_size] )`

If an `end do` directive is not specified, an `end do` directive is assumed at the end of the `do-loop`.

The `do-loop` must be a `do-construct` as defined in Section 8.1.4.1 of the Fortran 95 standard. If an `end do` directive follows a `do-construct` in which several `DO` statements share a `DO` termination statement, then a `do` directive can only be specified for the first (i.e. outermost) of these `DO` statements. See Section A.6 on page 125 for examples.

If the loop iteration variable would otherwise be shared, it is implicitly made private on the loop construct. See Section A.7 on page 127 for examples. Unless the variable is specified `lastprivate` on the loop construct, its value after the loop is undefined.

### Binding

The binding thread set for a loop region is the current team. A loop region binds to the innermost enclosing `parallel` region. Only the threads of the team executing the binding `parallel` region participate in the execution of the loop iterations and (optional) implicit barrier of the loop region.

### Description

There is an implicit barrier at the end of a loop construct unless a `nowait` clause is specified.
The `schedule` clause specifies how iterations of the loop are divided into contiguous non-empty subsets, called chunks, and how these chunks are assigned among threads of the team. Programs which depend on which thread executes a particular iteration are non-conforming. The `chunk_size` expression is evaluated using the original list items of any variables that are made private for the duration of the loop construct. It is unspecified whether, in what order, or how many times, any side-effects of the evaluation of this expression occur.

See Section 2.5.1.1 on page 38 for details of how the schedule for a work-sharing loop is determined.

The schedule *kind* can be one of those specified in Table 2-3.

<table>
<thead>
<tr>
<th>schedule</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>static</td>
<td>When <code>schedule(static, chunk_size)</code> is specified, iterations are divided into chunks of size <code>chunk_size</code>, and the chunks are statically assigned to threads in the team in a round-robin fashion in the order of the thread number. Note that the last chunk to be assigned may have a smaller number of iterations. When no <code>chunk_size</code> is specified, the iteration space is divided into chunks which are approximately equal in size, and each thread is assigned at most one chunk.</td>
</tr>
<tr>
<td>dynamic</td>
<td>When <code>schedule(dynamic, chunk_size)</code> is specified, the iterations are assigned to threads in chunks as the threads request them. The thread executes the chunk of iterations, then requests another chunk, until no chunks remain to be assigned. Each chunk contains <code>chunk_size</code> iterations, except for the last chunk to be assigned, which may have fewer iterations. When no <code>chunk_size</code> is specified, it defaults to 1.</td>
</tr>
<tr>
<td>guided</td>
<td>When <code>schedule(guided, chunk_size)</code> is specified, the iterations are assigned to threads in chunks as the threads request them. The thread executes the chunk of iterations, then requests another chunk, until no chunks remain to be assigned. For a <code>chunk_size</code> of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a <code>chunk_size</code> with value <code>k</code> (greater than 1), the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than <code>k</code> iterations (except for the last chunk to be assigned, which may have fewer than <code>k</code> iterations). When no <code>chunk_size</code> is specified, it defaults to 1.</td>
</tr>
</tbody>
</table>
Note – For a team of $p$ threads and a loop of $n$ iterations, let $q$ be the integer which satisfies $n = p \times q - r$, with $0 \leq r < p$. One compliant implementation of the **static** schedule (with no specified **chunk_size**) would behave as though **chunk_size** had been specified with value $q$. Another compliant implementation would assign $q$ iterations to the first $p-r$ threads, and $q-1$ iterations to the remaining $r$ threads. This illustrates why a conforming program must not rely on the details of a particular implementation.

A compliant implementation of the **guided** schedule with a **chunk_size** value of $k$ would assign $q = \lceil n/p \rceil$ iterations to the first available thread and set $n$ to the larger of $n-q$ and $p*k$. It would then repeat this process until $q$ is greater than or equal to the number of remaining iterations, at which time the remaining iterations form the final chunk. Another compliant implementation could use the same method, except with $q = \lceil n/(2p) \rceil$, and set $n$ to the larger of $n-q$ and $2*p*k$.

Restrictions

Restrictions to the loop construct are as follows:

- The values of the loop control expressions of the loop associated with the loop directive must be the same for all the threads in the team.
- Only a single **schedule** clause can appear on a loop directive.
- **chunk_size** must be a loop invariant integer expression with a positive value.
- The value of the **chunk_size** expression must be the same for all threads in the team.
- When **schedule(runtime)** is specified, **chunk_size** must not be specified.
- Only a single **ordered** clause can appear on a loop directive.
- The **ordered** clause must be present on the loop construct if any **ordered** region ever binds to a loop region arising from the loop construct.
- The loop iteration variable may not appear in a **threadprivate** directive.

**C/C++**

- The **for-loop** must be a structured block, and in addition, its execution must not be terminated by a **break** statement.
- The **for-loop** iteration variable **var** must have a signed integer type.
- Only a single **nowait** clause can appear on a **for** directive.
• If relational-op is < or <= then incr-exp must cause var to increase on each iteration of the loop. Conversely, if relational-op is > or >= then incr-exp must cause var to decrease on each iteration of the loop.

C/C++

Fortran

• The do-loop must be a structured block, and in addition, its execution must not be terminated by an EXIT statement.

• The do-loop iteration variable must be of type integer.

• The do-loop cannot be a DO WHILE or a DO loop without loop control.

Cross References

• private, firstprivate, lastprivate, and reduction clauses, see Section 2.8.3 on page 70.

• OMP_SCHEDULE environment variable, see Section 4.1 on page 114.

• ordered construct, see Section 2.7.6 on page 61.

2.5.1.1 Determining the Schedule of a Work-sharing Loop

When execution encounters a loop directive, the schedule clause (if any) on the directive, and the run-sched-var and def-sched-var internal control variables are used to determine how loop iterations are assigned to threads. See Section 2.3 on page 24 for details of how the values of the internal control variables are determined. If no schedule clause is used on the work-sharing loop directive, then the schedule is taken from the current value of def-sched-var. If the schedule clause is used and specifies the runtime schedule kind, then the schedule is taken from the run-sched-var control variable. Otherwise, the schedule is taken from the value of the schedule clause. Figure 2-2 describes how the schedule for a work-sharing loop is determined.

Cross References

• Internal control variables, see Section 2.3 on page 24.
### 2.5.2 sections Construct

**Summary**

The `sections` construct is a noniterative work-sharing construct that contains a set of structured blocks that are to be divided among, and executed by, the threads in a team. Each structured block is executed once by one of the threads in the team.

---

**FIGURE 2-2** Determining the schedule for a work-sharing loop.
Syntax

C/C++

The syntax of the `sections` construct is as follows:

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

The `clause` is one of the following:

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

Fortran

The syntax of the `sections` construct is as follows:

```
!$omp sections [clause[[,] clause] ...]
  !$omp section/
  structured-block
  !$omp section
  structured-block
...
!$omp end sections [nowait/]
```
The clause is one of the following:

\begin{verbatim}
private(list)
firstprivate(list)
lastprivate(list)
reduction(operator | intrinsic_procedure_name : list)
\end{verbatim}

---

**Fortran**

**Binding**

The binding thread set for a sections region is the current team. A sections region binds to the innermost enclosing parallel region. Only the threads of the team executing the binding parallel region participate in the execution of the structured blocks and (optional) implicit barrier of the sections region.

**Description**

Each structured block in the sections construct is preceded by a section directive except possibly the first block, for which a preceding section directive is optional.

The method of scheduling the structured blocks among threads in the team is implementation defined.

There is an implicit barrier at the end of a sections construct, unless a nowait clause is specified.

**Restrictions**

Restrictions to the sections construct are as follows:

- The section directives must appear within the sections construct and may not be encountered elsewhere in the sections region.

- The code enclosed in a sections construct must be a structured block.

- Only a single nowait clause can appear on a sections directive.

---

C/C++
Cross References

- `private`, `firstprivate`, `lastprivate`, and `reduction` clauses, see Section 2.8.3 on page 70.

2.5.3 single Construct

Summary

The `single` construct specifies that the associated structured block is executed by only one thread in the team (not necessarily the master thread). The other threads in the team do not execute the block, and wait at an implicit barrier at the end of `single` construct, unless a `nowait` clause is specified.

Syntax

C/C++

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

The `clause` is one of the following:

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

Fortran

```fortran
!$omp single [clause[,[ clause] ...]
structured-block
!$omp end single [end_clause[,[ end_clause] ...]
```

The `clause` is one of the following:
The clause is one of the following:

\[ \text{private}(\text{list}) \]
\[ \text{firstprivate}(\text{list}) \]

and end_clause is one of the following:

\[ \text{copyprivate}(\text{list}) \]
\[ \text{nowait} \]

**Binding**

The binding thread set for a single region is the current team. A single region binds to the innermost enclosing parallel region. Only the threads of the team executing the binding parallel region participate in the execution of the structured block and (optional) implicit barrier of the single region.

**Description**

The method of choosing a thread to execute the structured block is implementation defined. There is an implicit barrier after the single construct unless a nowait clause is specified.

For an example of the single construct, see Section A.10 on page 130.

**Restrictions**

Restrictions to the single construct are as follows:

- The copyprivate clause must not be used with the nowait clause.
- At most one nowait clause can appear on a single construct.

**Cross References**

- private and firstprivate clauses, see Section 2.8.3 on page 70.
- copyprivate clause, see Section 2.8.4.2 on page 85.
2.5.4 workshare Construct

Summary

The workshare construct divides the execution of the enclosed structured block into separate units of work, and causes the threads of the team to share the work such that each unit is executed only once.

Syntax

The syntax of the workshare construct is as follows:

```fortran
!$omp workshare
    structured-block
!$omp end workshare [nowait]
```

The enclosed structured block must consist of only the following:

- array assignments
- scalar assignments
- FORALL statements
- FORALL constructs
- WHERE statements
- WHERE constructs
- atomic constructs
- critical constructs
- parallel constructs

Statements contained in any enclosed critical construct are also subject to these restrictions. Statements in any enclosed parallel construct are not restricted.

Binding

The binding thread set for a workshare region is the current team. A workshare region binds to the innermost enclosing parallel region. Only the threads of the team executing the binding parallel region participate in the execution of the units of work and (optional) implicit barrier of the workshare region.
Description

There is an implicit barrier at the end of a workshare construct unless a nowait clause is specified.

An implementation of the workshare construct must insert any synchronization that is required to maintain standard Fortran semantics. For example, the effects of one statement within the structured 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 the workshare construct 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, including any references to ELEMENTAL functions, 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 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 constructs, 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 construct.
- If none of the rules above apply to a portion of a statement in the structured 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.

It is unspecified how the units of work are assigned to the threads executing a workshare region.
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 workshare construct, and not in the remainder of the workshare region.

For examples of the workshare construct, see Section A.11 on page 132.

Restrictions

The following restrictions apply to the workshare directive:

• The construct must not contain any user defined function calls unless the function is ELEMENTAL.

2.6 Combined Parallel Work-sharing Constructs

Combined parallel work-sharing constructs are shortcuts for specifying a work-sharing construct nested immediately inside a parallel construct. The semantics of these directives are identical to that of explicitly specifying a parallel construct containing one work-sharing construct and no other statements.

The combined parallel work-sharing constructs allow certain clauses which are permitted on both parallel constructs and on work-sharing constructs. If a program would have different behavior depending on whether the clause were applied to the parallel construct or to the work-sharing construct, then the program’s behavior is unspecified.

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

• The parallel loop construct.

• The parallel sections construct.

• The parallel workshare construct.
### 2.6.1 Parallel loop construct

#### Summary

The parallel loop construct is a shortcut for specifying a `parallel` construct containing one loop construct and no other statements.

#### Syntax

**C/C++**

The syntax of the parallel loop construct is as follows:

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

The clause can be any of the clauses accepted by the `parallel` or `for` directives, except the `nowait` clause, with identical meanings and restrictions.

**Fortran**

The syntax of the parallel loop construct is as follows:

```fortran
!$omp parallel do [clause[,...] clause] ...
```

The `clause` can be any of the clauses accepted by the `parallel` or `do` directives, with identical meanings and restrictions. However, `nowait` may not be specified on an `end parallel do` directive.

If an `end parallel do` directive is not specified, an `end parallel do` directive is assumed at the end of the `do-loop`.

#### Description

**C/C++**

The semantics are identical to explicitly specifying a `parallel` directive immediately followed by a `for` directive.

**Fortran**

...
Fortran

The semantics are identical to explicitly specifying a `parallel` directive immediately followed by a `do` directive, and an `end do` directive immediately followed by an `end parallel` directive.

Restrictions

The restrictions for the `parallel` construct and the loop construct apply.

Cross References

- `parallel` construct, see Section 2.4 on page 26.
- `loop` construct, see Section 2.5.1 on page 33.
- Data attribute clauses, see Section 2.8.3 on page 70.

2.6.2 `parallel sections` Construct

Summary

The `parallel sections` construct is a shortcut for specifying a `parallel` construct containing one `sections` construct and no other statements.

Syntax

```
#pragma omp parallel sections [clause[[,] clause] ...] new-line
{
  /#pragma omp section new-line
  structured-block
  /#pragma omp section new-line
  structured-block }
... }
```
The clause can be any of the clauses accepted by the `parallel` or `sections` directives, except the `nowait` clause, with identical meanings and restrictions.

C/C++

Fortran

The syntax of the `parallel sections` construct is as follows:

```c
#pragma omp parallel sections [clause[,, clause] ...]
  /#pragma omp section
  structured-block
  /#pragma omp section
  structured-block
  ...
#pragma omp end parallel sections
```

The clause can be any of the clauses accepted by the `parallel` or `sections` directives, with identical meanings and restrictions. However, `nowait` cannot be specified on an `end parallel sections` directive.

The last section ends at the `end parallel sections` directive.

Description

C/C++

Fortran

The semantics are identical to explicitly specifying a `parallel` directive immediately followed by a `sections` directive.

C/C++

Fortran

The semantics are identical to explicitly specifying a `parallel` directive immediately followed by a `sections` directive, and an `end sections` directive immediately followed by an `end parallel` directive.

Fortran

For an example of the parallel sections construct, see Section A.9 on page 129.

Restrictions

The restrictions for the `parallel` construct and the `sections` construct apply.
Cross References:

• parallel construct, see Section 2.4 on page 26.
• sections construct, see Section 2.5.2 on page 39.
• Data attribute clauses, see Section 2.8.3 on page 70.

2.6.3 parallel workshare Construct

Summary

The parallel workshare construct is a shortcut for specifying a parallel construct containing one workshare construct and no other statements.

Syntax

The syntax of the parallel workshare construct is as follows:

```fortran
!$omp parallel workshare [clause[,...] clause] ...
structured-block
!$omp end parallel workshare
```

The clause can be any of the clauses accepted by the parallel directive, with identical meanings and restrictions. However, nowait may not be specified on an end parallel workshare directive.

Description

The semantics are identical to explicitly specifying a parallel directive immediately followed by a workshare directive, and an end workshare directive immediately followed by an end parallel directive.

Restrictions

The restrictions for the parallel construct and the workshare construct apply.

Cross References

• parallel construct, see Section 2.4 on page 26.
• workshare construct, see Section 2.5.4 on page 44.
2.7 Master and Synchronization Constructs

The following sections describe:

- the master construct.
- the critical construct.
- the barrier construct.
- the atomic construct.
- the flush construct.
- the ordered construct.

2.7.1 master Construct

Summary

The master construct specifies a structured block that is executed by the master thread of the team.

Syntax

The syntax of the master construct is as follows:

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

• Data attribute clauses, see Section 2.8.3 on page 70.
The syntax of the `master` construct is as follows:

```
!$omp master
  structured-block
!$omp end master
```

**Binding**

The binding thread set for a `master` region is the current team. A `master` region binds to the innermost enclosing `parallel` region. Only the master thread of the team executing the binding `parallel` region participates in the execution of the structured block of the `master` region.

**Description**

Other threads in the team do not execute the associated structured block. There is no implied barrier either on entry to, or exit from, the `master` construct.

For an example of the `master` construct, see Section A.12 on page 136.

### 2.7.2 `critical` Construct

**Summary**

The `critical` construct restricts execution of the associated structured block to a single thread at a time.

**Syntax**

```
#pragma omp critical [(name)] new-line
  structured-block
```

C/C++
The syntax of the *critical* construct is as follows:

```fortran
!$omp critical [(name)]
  structured-block
!$omp end critical [(name)]
```

### Binding

The binding thread set for a *critical* region is all threads. Region execution is restricted to a single thread at a time among all the threads in the program, without regard to the team(s) to which the threads belong.

### Description

An optional *name* may be used to identify the *critical* construct. All *critical* constructs without a name are considered to have the same unspecified name. A thread waits at the beginning of a *critical* region until no other thread is executing a *critical* region with the same name. The *critical* construct enforces exclusive access with respect to all *critical* constructs with the same name in all threads, not just in the current team.

Identifiers used to identify a *critical* construct have external linkage and are in a name space which is separate from the name spaces used by labels, tags, members, and ordinary identifiers.

The names of *critical* constructs are global entities of the program. If a name conflicts with any other entity, the behavior of the program is unspecified.

For an example of the *critical* construct, see Section A.13 on page 138.

### Restrictions

The following restrictions apply to the *critical* construct:
• If a name is specified on a critical directive, the same name must also be specified on the end critical directive.

• If no name appears on the critical directive, no name can appear on the end critical directive.

### 2.7.3 barrier Construct

#### Summary

The barrier construct specifies an explicit barrier at the point at which the construct appears.

#### Syntax

```c
#include <omp.h>

#pragma omp barrier
```

The syntax of the barrier construct is as follows:

Note that because the barrier construct does not have a C language statement as part of its syntax, there are some restrictions on its placement within a program. The barrier directive may only be placed in the program at a position where ignoring or deleting the directive would result in a program with correct syntax. See Appendix C for the formal grammar. The examples in Section A.20 on page 153 illustrate these restrictions.

```fortran
&omp barrier
```

The syntax of the barrier construct is as follows:
**Binding**

The binding thread set for a barrier region is the current team. A barrier region binds to the innermost enclosing parallel region. See Section A.15 on page 140 for examples.

**Description**

All of the threads of the team executing the binding parallel region must execute the barrier region before any are allowed to continue execution beyond the barrier.

**Restrictions**

The following restrictions apply to the barrier construct:

- Each barrier region must be encountered by all threads in a team or by none at all.
- The sequence of work-sharing regions and barrier regions encountered must be the same for every thread in a team.

### 2.7.4 atomic Construct

**Summary**

The atomic construct ensures that a specific storage location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

**Syntax**

```c/c++
#pragma omp atomic new-line
expression-stmt
```

expression-stmt is an expression statement with one of the following forms:

- `x binop= expr`
- `x++`
In the preceding expressions:

- $x$ is an lvalue expression with scalar type.
- $expr$ is an expression with scalar type, and it does not reference the object designated by $x$.
- $binop$ is not an overloaded operator and is one of $+$, $*$, $-$, $/$, $\&$, $^\wedge$, $\mid$, $\ll$, or $\gg$.

The syntax of the `atomic` construct is as follows:

```c
!$omp atomic
statement
```

where `statement` has one of the following forms:

- $x = x \text{ operator } expr$
- $x = expr \text{ operator } x$
- $x = \text{ intrinsic\_procedure\_name} (x, expr\_list)$
- $x = \text{ intrinsic\_procedure\_name} (expr\_list, x)$

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 $(expr)\_operator x$.
- `intrinsic\_procedure\_name` must refer to the intrinsic procedure name and not to other program entities.
• *operator* must refer to the intrinsic operator and not to a user-defined operator.

• The assignment must be intrinsic assignment.

---

**Fortran**

**Binding**

The binding thread set for an *atomic* region is all threads. *atomic* regions enforce exclusive access with respect to other *atomic* regions that update the same storage location *x* among all the threads in the program without regard to the team(s) to which the threads belong.

**Description**

Only the load and store of the object designated by *x* are atomic; the evaluation of *expr* is not atomic. To avoid race conditions, all updates of the location which could potentially occur in parallel must be protected with an *atomic* directive. *atomic* regions do not enforce exclusive access with respect to any *critical* or *ordered* regions which access the same storage location *x*.

A compliant implementation may enforce exclusive access between *atomic* regions which update different storage locations. The circumstances under which this occurs are implementation defined.

For an example of the *atomic* construct, see Section A.16 on page 142.

**Restrictions**

---

**C/C++**

The following restriction applies to the *atomic* construct:

• All atomic references to the storage location throughout the program are required to have a compatible type. See Section A.17 on page 144 for examples.

---

**Fortran**

The following restriction applies to the *atomic* construct:

• All atomic references to the storage location of variable *x* throughout the program are required to have the same type and type parameters. See Section A.17 on page 144 for examples.
Cross References

- critical construct, see Section 2.7.2 on page 52.

2.7.5 flush Construct

Summary

The flush construct executes the OpenMP flush operation. This operation makes a thread’s temporary view of memory consistent with memory, and enforces an order on the memory operations of the variables explicitly specified or implied. See the memory model description in Section 1.4 on page 10 for more details.

Syntax

The syntax of the flush construct is as follows:

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

Binding

The binding thread set for a flush region is the encountering thread. Execution of a flush region only affects the view of memory from the thread which executes the region. Other threads must themselves execute a flush operation in order to observe the effects of the encountering thread’s flush operation.
Description

A **flush** construct with a list applies the flush operation to the items in the list, and does not return until the operation is complete for all specified list items. A **flush** construct without a list, executed on a given thread, operates as if the whole thread-visible data state of the program, as defined by the base language, is flushed.

If a pointer is present in the list, the pointer itself is flushed, not the object to which the pointer refers.

If the list item or a subobject of the list item has the **POINTER** attribute, the allocation or association status of the **POINTER** item is flushed, but the pointer target is not. If the list item is a Cray pointer, the pointer is flushed, but the object to which it points is not. If the list item has the **ALLOCATABLE** attribute and the list item is allocated, the allocated array is flushed; otherwise the allocation status is flushed.

For examples of the **flush** construct, see Section A.18 on page 147 and Section A.19 on page 150.

**Note** – the following examples illustrate the ordering properties of the flush operation. In the following incorrect pseudocode example, the programmer intends to prevent simultaneous execution of the critical section by the two threads, but the program does not work properly because it does not enforce the proper ordering of the operations on variables \(a\) and \(b\).

**Incorrect example:**

\[
\begin{align*}
\text{thread 1} & \quad \text{thread 2} \\
\text{a} = \text{b} = 0 & \\
\text{b} = 1 & \quad \text{a} = 1 \\
\text{flush (b)} & \quad \text{flush (a)} \\
\text{flush (a)} & \quad \text{flush (b)} \\
\text{if (a == 0) then} & \quad \text{if (b == 0) then} \\
\text{critical section} & \quad \text{critical section} \\
\text{end if} & \quad \text{end if}
\end{align*}
\]
The problem with this example is that operations on variables \(a\) and \(b\) are not ordered with respect to each other. For instance, nothing prevents the compiler from moving the flush of \(b\) on thread 1 or the flush of \(a\) on thread 2 to a position completely after the critical section (assuming that the critical section on thread 1 does not reference \(b\) and the critical section on thread 2 does not reference \(a\)). If either re-ordering happens, the critical section can be active on both threads simultaneously.

The following correct pseudocode example correctly ensures that the critical section is executed by not more than one of the two threads at any one time. Notice that execution of the critical section by neither thread is considered correct in this example.

```
Correct example:

\[
\begin{align*}
&a = b = 0 \\
\text{thread 1} & \quad \text{thread 2} \\
&b = 1 \quad a = 1 \\
&\text{flush}(a,b) \quad \text{flush}(a,b) \\
&\text{if } (a == 0) \text{ then} \\
&\quad \text{critical section} \\
&\text{end if} \\
&\text{critical section} \\
&\text{end if}
\end{align*}
\]
```

The compiler is prohibited from moving the flush at all for either thread, ensuring that the respective assignment is complete and the data is flushed before the if statement is executed.

---

C/C++

Note that because the `flush` construct does not have a C language statement as part of its syntax, there are some restrictions on its placement within a program. The `flush` directive may only be placed in the program at a position where ignoring or deleting the directive would result in a program with correct syntax. See Appendix C for the formal grammar. See Section A.20 on page 153 for an example that illustrates these placement restrictions.

---

C/C++

A `flush` region without a list is implied at the following locations:

- During a `barrier` region.
- At entry to and exit from `parallel`, `critical` and `ordered` regions.
- At exit from work-sharing regions, unless a `nowait` is present.
- At entry to and exit from combined parallel work-sharing regions.
• During `omp_set_lock` and `omp_unset_lock` regions.

• During `omp_test_lock`, `omp_set_nest_lock`, `omp_unset_nest_lock` and `omp_test_nest_lock` regions, if the region causes the lock to be set or unset.

A flush region with a list is implied at the following locations:

• At entry to and exit from `atomic` regions, where the list contains only the object updated in the `atomic` construct.

---

Note – A flush region is not implied at the following locations:

• At entry to work-sharing regions.

• At entry to or exit from a `master` region.

### 2.7.6 ordered Construct

#### Summary

The ordered construct specifies a structured block in a loop region which will be executed in the order of the loop iterations. This sequentializes and orders the code within an ordered region while allowing code outside the region to run in parallel.

#### Syntax

```c/c++
#pragma omp ordered new-line
structured-block
```

The syntax of the ordered construct is as follows:
The syntax of the `ordered` construct is as follows:

```fortran
!$omp ordered
   structured-block
!$omp end ordered
```

**Binding**

The binding thread set for an `ordered` region is the current team. An `ordered` region binds to the innermost enclosing loop region. `ordered` regions that bind to different loop regions execute completely independently of each other.

**Description**

The threads in the team executing the loop region execute `ordered` regions sequentially in the order of the loop iterations. When the thread executing the first iteration of the loop encounters an `ordered` construct, it can enter the `ordered` region without waiting. When a thread executing any subsequent iteration encounters an `ordered` region, it waits at the beginning of that `ordered` region until each of the previous iterations that contains an `ordered` region has completed the `ordered` region.

For examples of the `ordered` construct, see Section A.21 on page 154.

**Restrictions**

Restrictions to the `ordered` construct are as follows:

- The loop region to which an `ordered` region binds must have an `ordered` clause specified on the corresponding loop (or parallel loop) construct.
- During execution of an iteration of a loop within a loop region, the executing thread must not execute more than one `ordered` region which binds to the same loop region.

**Cross References**

- `loop construct`, see Section 2.5.1 on page 33.
- `parallel loop construct`, see Section 2.6.1 on page 47.
2.8 Data Environment

This section presents a directive and several clauses for controlling the data environment during the execution of parallel regions.

- Section 2.8.1 on page 63 describes how the sharing attributes of variables referenced in parallel regions are determined.
- The threadprivate directive, which is provided to create threadprivate memory, is described in Section 2.8.2 on page 66.
- Clauses that may be specified on directives to control the sharing attributes of variables referenced in parallel or work-sharing constructs are described in Section 2.8.3 on page 70.
- Clauses that may be specified on directives to copy data values from private or threadprivate objects on one thread to the corresponding objects on other threads in the team are described in Section 2.8.4 on page 83.

2.8.1 Sharing Attribute Rules

This section describes how the sharing attributes of variables referenced in parallel regions are determined. The following two cases are described separately:

- Section 2.8.1.1 on page 63 describes the sharing attribute rules for variables referenced in a construct.
- Section 2.8.1.2 on page 65 describes the sharing attribute rules for variables referenced in a region, but outside any construct.

2.8.1.1 Sharing Attribute Rules for Variables Referenced in a Construct

The sharing attributes of variables which are referenced in a construct may be one of the following: predetermined, explicitly determined, or implicitly determined.

Note that specifying a variable on a firstprivate, lastprivate, or reduction clause of an enclosed construct causes an implicit reference to the variable in the enclosing construct. Such implicit references are also subject to the following rules.

The following variables have predetermined sharing attributes:

- Variables appearing in threadprivate directives are threadprivate.
• Variables with automatic storage duration which are declared in a scope inside the
construct are private.
• Variables with heap allocated storage are shared.
• Static data members are shared.
• The loop iteration variable in the *for-loop* of a *for* or *parallel for* construct is
private in that construct.
• Variables with const-qualified type having no mutable member are shared.

\[\text{C/C++}\]

\[\text{Fortran}\]

• Variables and common blocks appearing in *threadprivate* directives are
threadprivate.
• The loop iteration variable in the *do-loop* of a *do* or *parallel do* construct is
private in that construct.
• Variables used as loop iteration variables in sequential loops in a *parallel*
construct are private in the *parallel* construct.
• *implied-do* and *forall* indices are private.
• Cray pointees inherit the sharing attribute of the storage with which their Cray
pointers are associated.

\[\text{Fortran}\]

Variables with predetermined sharing attributes may not be listed in data-sharing
attribute clauses, with the following exceptions:

\[\text{C/C++}\]

• The loop iteration variable in the *for-loop* of a *for* or *parallel for* construct
may be listed in a *private* or *lastprivate* clause.

\[\text{C/C++}\]

\[\text{Fortran}\]

• The loop iteration variable in the *do-loop* of a *do* or *parallel do* construct may be
listed in a *private* or *lastprivate* clause.
• Variables used as loop iteration variables in sequential loops in a *parallel*
construct may be listed in *private, firstprivate, lastprivate, shared,*
*or* *reduction* clauses.

\[\text{Fortran}\]

Additional restrictions on the variables which may appear in individual clauses are
described with each clause in Section 2.8.3 on page 70.

Variables referenced in the construct are said to have an explicitly determined sharing
attribute if they are listed in a data-sharing attribute clause on the construct.
Variables referenced in the construct whose sharing attribute is not predetermined or explicitly determined will have their sharing attribute implicitly determined. In a parallel construct, the sharing attributes of these variables is determined by the default clause, if present (see Section 2.8.3.1 on page 71). If no default clause is present, variables with implicitly determined sharing attributes are shared. For other constructs, variables with implicitly determined sharing attributes inherit their sharing attributes from the enclosing context.

2.8.1.2 Sharing Attribute Rules for Variables Referenced in a Region, but not in a Construct

The sharing attributes of variables which are referenced in a region, but not in a construct, are determined as follows:

C/C++
- Static variables declared in called routines in the region are shared.
- Variables with const-qualified type having no mutable member, and that are declared in called routines, are shared.
- File-scope or namespace-scope variables referenced in called routines in the region are shared unless they appear in a threadprivate directive.
- Variables with heap-allocated storage are shared.
- Static data members are shared.
- Formal arguments of called routines in the region that are passed by reference inherit the data-sharing attributes of the associated actual argument.
- Other variables declared in called routines in the region are private.

Fortran
- Local variables declared in called routines in the region and that have the save attribute, or that are data initialized, are shared unless they appear in a threadprivate directive.
- Variables belonging to common blocks, or declared in modules, and referenced in called routines in the region are shared unless they appear in a threadprivate directive.
- Dummy arguments of called routines in the region that are passed by reference inherit the data-sharing attributes of the associated actual argument.
- implied-do and forall indices are private.
- Cray pointees inherit the sharing attribute of the storage with which their Cray pointers are associated.
• Other local variables declared in called routines in the region are private.

### 2.8.2 threadprivate Directive

**Summary**

The `threadprivate` directive specifies that named global-lifetime objects are replicated, with each thread having its own copy.

**Syntax**

```c
#pragma omp threadprivate(list) new-line
```

where `list` is a comma-separated list of file-scope, namespace-scope, or static block-scope variables that do not have incomplete types.

```fortran
!$omp threadprivate(list)
```

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

**Description**

Each copy of a threadprivate object is initialized once, in the manner specified by the program, but at an unspecified point in the program prior to the first reference to that copy.

A thread may not reference another thread’s copy of a threadprivate object.
During the sequential part, and in non-nested inactive parallel regions, references
will be to the initial thread’s copy of the object. In parallel regions, references by
the master thread will be to the copy of the object in the thread which encountered the
parallel region.

The values of data in the initial thread’s copy of a threadprivate object are guaranteed to
persist between any two consecutive references to the object in the program.

The values of data in the threadprivate objects of threads other than the initial thread are
guaranteed to persist between two consecutive active parallel regions only if all the
following conditions hold:

- Neither parallel region is nested inside another parallel region.
- The number of threads used to execute both parallel regions is the same.
- The value of the dyn-var internal control variable is false at entry to the first
  parallel region and remains false until entry to the second parallel region.
- The value of the nthreads-var internal control variable is the same at entry to both
  parallel regions and has not been modified between these points.

If these conditions all hold, and if a threadprivate object is referenced in both regions,
then threads with the same thread number in their respective regions will reference the
same copy of that variable.

C/C++

If the above conditions hold, the storage duration, lifetime, and value of a thread’s copy
of a threadprivate variable that does not appear in any copyin clause on the second
region will be retained. Otherwise, the storage duration, lifetime, and value of a thread’s
copy of the variable in the second region is undefined.

If an object is referenced in an explicit initializer of a threadprivate variable, and the
value of the object is modified prior to the first reference to a copy of the variable, then
the behavior is unspecified.

C/C++

Fortran

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 the above conditions hold, 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 second region, will be retained.
Otherwise, the definition and association status of a thread’s copy of the variable in the
second region is undefined, and the allocation status of an allocatable array will be
implementation defined.
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.

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 which it is referenced, 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.

For examples of the threadprivate directive, see Section A.22 on page 158.

Restrictions

The restrictions to the threadprivate directive are as follows:

- A threadprivate object must not appear in any clause except the copyin, copyprivate, schedule, num_threads, and if clauses.

- A threadprivate directive for file-scope variables must appear outside any definition or declaration, and must lexically precede all references to any of the variables in its list.

- A threadprivate directive for namespace-scope variables must appear outside any definition or declaration other than the namespace definition itself, and must lexically precede all references to any of the variables in its list.

- Each variable in the list of a threadprivate directive at file or namespace scope must refer to a variable declaration at file or namespace scope that lexically precedes the directive.

- A threadprivate directive for static block-scope variables must appear in the scope of the variable and not in a nested scope. The directive must lexically precede all references to any of the variables in its list.
• Each variable in the list of a `threadprivate` directive in block scope must refer to
  a variable declaration in the same scope that lexically precedes the directive. The
  variable declaration must use the static storage-class specifier.

• If a variable is specified in a `threadprivate` directive in one translation unit, it
  must be specified in a `threadprivate` directive in every translation unit in which
  it is declared.

• The address of a threadprivate variable is not an address constant.

• A threadprivate variable must not have an incomplete type or a reference type.

• A threadprivate variable with non-POD class type must have an accessible,
  unambiguous copy constructor if it is declared with an explicit initializer.

### C/C++

### Fortran

• The `threadprivate` 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.

• If a `threadprivate` directive specifying a common block name appears in one
  program unit, then such a directive must also appear in every other program unit that
  contains a `COMMON` statement specifying the same name. It must appear after the last
  such `COMMON` statement in the program unit.

• A blank common block cannot appear in a `threadprivate` directive.

• 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.

### Cross References:

• Dynamic adjustment of threads, see Section 2.4.1 on page 29.

• `copyin` clause, see Section 2.8.4.1 on page 84.

• Internal control variables, see Section 2.3 on page 24.
2.8.3 Data-Sharing Attribute Clauses

Several constructs accept clauses that allow a user to control the sharing attributes of variables for the duration of the construct. Data-sharing attribute clauses apply only to variables whose names are visible in the construct on which the clause appears, except that formal arguments that are passed by reference inherit the data-sharing attributes of the associated actual argument.

Not all of the clauses listed in this section are valid on all directives. The set of clauses that is valid on a particular directive is described with the directive.

Most of the clauses accept a comma-separated list of list items (see Section 2.1 on page 18). All list items appearing in a clause must be visible, according to the scoping rules of the base language. With the exception of the default clause, clauses may be repeated as needed. A list item that specifies a given variable may not appear in more than one clause on the same directive, except that a variable may be specified in both firstprivate and lastprivate clauses.

C/C++

If a variable referenced in a data-sharing attribute clause has a type derived from a template, and there are no other references to that variable in the program, then any behavior related to that variable is undefined.

Fortran

A named common block may be specified in a list by enclosing the name in 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. An explicit member of a common block is a variable that is named in a COMMON statement that specifies the common block name and is 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-sharing attribute clause must be declared to be a common block in the same scoping unit in which the data-sharing attribute clause appears.

When a named common block appears in a private, firstprivate, lastprivate, or shared clause of a directive, none of its members may be declared in another data-sharing attribute clause in that directive (see Section A.23 on page 163 for examples). 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.28 on page 171 for examples).
2.8.3.1 default clause

Summary

The default clause allows the user to control the sharing attributes of variables which are referenced in a parallel construct, and whose sharing attributes are implicitly determined (see Section 2.8.1.1 on page 63).

Syntax

C/C++

The syntax of the default clause is as follows:

default(shared|none)

default(private|shared|none)

Fortran

The syntax of the default clause is as follows:

Description

The default(shared) clause causes all variables referenced in the construct which have implicitly determined sharing attributes to be shared.

Fortran

The default(private) clause causes all variables referenced in the construct which have implicitly determined sharing attributes to be private.

Fortran

The default(none) clause requires that each variable which is referenced in the construct, and that does not have a predetermined sharing attribute, must have its sharing attribute explicitly determined by being listed in a data-sharing attribute clause. See Section A.24 on page 165 for examples.
Restrictions

The restrictions to the default clause are as follows:

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

2.8.3.2 shared clause

Summary

The shared clause declares one or more list items to be shared among all the threads in a team.

Syntax

The syntax of the shared clause is as follows:

shared(list)

Description

All threads within a team access the same storage area for each shared object.

Fortran

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.

Under certain conditions, passing a shared variable to a non-intrinsic procedure 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 situation will occur when the following three conditions hold regarding an actual argument in a reference to a non-intrinsic procedure:

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.
Chapter 2 Directives

• An array section with a vector subscript.
• 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.

This effectively results in references to, and definitions of, the storage during the procedure reference. 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.

It is implementation defined whether this situation might occur under other conditions. See Section A.25 on page 167 for an example of this behavior.

2.8.3.3 private clause

Summary

The private clause declares one or more list items to be private to a thread.

Syntax

The syntax of the private clause is as follows:

```
private(list)
```

Description

Each thread in the team that references a list item that appears in a private clause in any statement in the construct receives a new list item whose language-specific attributes are derived from the original list item. Inside the construct, all references to the original list item are replaced by references to the new list item. If a thread does not reference a list item that appears in a private clause, it is unspecified whether that thread receives a new list item.

The value of the original list item is not defined upon entry to the region. The original list item must not be referenced within the region. The value of the original list item is not defined upon exit from the region.
List items that are privatized in a **parallel** region may be privatized again in an enclosed **parallel** or work-sharing construct. As a result, list items that appear in a **private** clause on a **parallel** or work-sharing construct may be either shared or private in the enclosing context. See Section A.27 on page 170 for an example.

---

**C/C++**

A new list item of the same type, with automatic storage duration, is allocated for the construct. The size and alignment of the new list item are determined by the type of the variable. This allocation occurs once for each thread in the team that references the list item in any statement in the construct.

The new list item is initialized, or has an undefined initial value, as if it had been locally declared without an initializer. The order in which any default constructors for different private objects are called is unspecified.

---

**Fortran**

A new list item of the same type is declared once for each thread in the team that references the list item in any statement in the construct. The initial value of the new list item is undefined. Within a **parallel** region, the initial status of a **private** pointer is undefined.

A list item that appears in a **private** clause 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 is storage-associated with $A$, then:

- The contents, allocation, and association status of $B$ are undefined on entry to the **parallel** region.
- Any definition of $A$, or of its allocation or association status, causes the contents, allocation, and association status of $B$ to become undefined.
- Any definition of $B$, or of its allocation or association status, causes the contents, allocation, and association status of $A$ to become undefined.

For examples, see Section A.28 on page 171.

---

**Fortran**

For examples of the **private** clause, see Section A.26 on page 168.

**Restrictions**

The restrictions to the **private** clause are as follows:
• A list item that appears in the reduction clause of a parallel construct must not appear in a private clause on a work-sharing construct if any of the work-sharing regions arising from the work-sharing construct ever bind to any of the parallel regions arising from the parallel construct.

  C/C++

• A variable of class type (or array thereof) that appears in a private clause requires an accessible, unambiguous default constructor for the class type.

• A variable that appears in a private clause must not have a const-qualified type unless it is of class type with a mutable member.

• A variable that appears in a private clause must not have an incomplete type or a reference type.

  C/C++

  Fortran

• A variable that appears in a private clause must either be definable, or an allocatable array.

• An allocatable array that appears in a private clause must have an allocation status of “not currently allocated” on entry to and on exit from the construct.

• Assumed-size arrays may not appear in a private clause.

• Variables that appear in namelist statements, in variable format expressions, and in expressions for statement function definitions, may not appear in a private clause.

  Fortran

2.8.3.4 firstprivate clause

Summary

The firstprivate clause declares one or more list items to be private to a thread, and initializes each of them with the value that the corresponding original item has when the construct is encountered.

Syntax

The syntax of the firstprivate clause is as follows:

```
firstprivate(list)
```
Description

The `firstprivate` clause provides a superset of the functionality provided by the `private` clause.

A list item that appears in a `firstprivate` clause is subject to the `private` clause semantics described in Section 2.8.3.3 on page 73. In addition, the new list item is initialized from the original list item existing before the construct. The initialization of the new list item is done once for each thread in the team that references the list item in any statement in the construct. The initialization is done prior to the thread’s execution of the construct.

For a `firstprivate` clause on a `parallel` construct, the initial value of the new list item is the value of the original list item that exists immediately prior to the `parallel` construct for the thread that encounters the construct. For a `firstprivate` clause on a work-sharing construct, the initial value of the new list item for a thread that executes the work-sharing construct is the value of the original list item that exists immediately prior to the point in time that the thread encounters the work-sharing construct.

If a list item appears in both `firstprivate` and `lastprivate` clauses, the update required for `lastprivate` occurs after all the initializations for `firstprivate`.

---

C/C++

For variables of non-array type, the initialization occurs as if by assignment. For a (possibly multi-dimensional) array of objects of non-array type, each element is initialized as if by assignment from an element of the original array to the corresponding element of the new array. For class types, a copy constructor is invoked to perform the initialization. The order in which copy constructors for different objects are called is unspecified.

---

Fortran

The initialization of the new list items occurs as if by assignment.

---

Restrictions

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

- A list item that is private within a `parallel` region, or that appears in the `reduction` clause of a `parallel` construct, must not appear in a `firstprivate` clause on a work-sharing construct if any of the work-sharing regions arising from the work-sharing construct ever bind to any of the `parallel` regions arising from the `parallel` construct.
Chapter 2 Directives

2.8.3.5 lastprivate clause

Summary

The lastprivate clause declares one or more list items to be private to a thread, and causes the corresponding original list item to be updated after the end of the region.

Syntax

The syntax of the lastprivate clause is as follows:

```
lastprivate (list)
```

Description

The lastprivate clause provides a superset of the functionality provided by the private clause.
A list item that appears in a `lastprivate` clause is subject to the `private` clause semantics described in Section 2.8.3.3 on page 73. In addition, when a `lastprivate` clause appears on the directive that identifies a work-sharing construct, the value of each new list item from the sequentially last iteration of the associated loop, or the lexically last `section` construct, is assigned to the original list item.

For a (possibly multi-dimensional) array of objects of non-array type, each element is assigned to the corresponding element of the original array.

List items that are not assigned a value by the sequentially last iteration of the loop, or by the lexically last `section` construct, have unspecified values after the construct. Unassigned subobjects also have an unspecified value after the construct.

The original list item becomes defined at the end of the construct if there is an implicit barrier at that point. Any concurrent uses or definitions of the original list item must be synchronized with the definition that occurs at the end of the construct to avoid race conditions.

If the `lastprivate` clause is used on a construct to which `nowait` is also applied, the original list item remains undefined until a barrier synchronization has been performed to ensure that the thread that executed the sequentially last iteration, or the lexically last `section` construct, has stored that list item.

If a list item appears in both `firstprivate` and `lastprivate` clauses, the update required for `lastprivate` occurs after all initializations for `firstprivate`.

For an example of the `lastprivate` clause, see Section A.30 on page 175.

**Restrictions**

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

- A list item that is private within a `parallel` region, or that appears in the `reduction` clause of a `parallel` construct, must not appear in a `lastprivate` clause on a work-sharing construct if any of the corresponding work-sharing regions ever binds to any of the corresponding `parallel` regions.

- A variable of class type (or array thereof) that appears in a `lastprivate` clause requires an accessible, unambiguous default constructor for the class type, unless the list item is also specified in a `firstprivate` clause.

- A variable of class type (or array thereof) that appears in a `lastprivate` clause requires an accessible, unambiguous copy assignment operator for the class type. The order in which copy assignment operators for different objects are called is unspecified.
• A variable that appears in a `lastprivate` clause must not have a `const`-qualified type unless it is of class type with a `mutable` member.
• A variable that appears in a `lastprivate` clause must not have an incomplete type or a reference type.

2.8.3.6 reduction clause

Summary

The `reduction` clause specifies an operator and one or more list items. For each list item, a private copy is created on each thread, and is initialized appropriately for the operator. After the end of the region, the original list item is updated with the values of the private copies using the specified operator.

Syntax

The syntax of the `reduction` clause is as follows:

```c
reduction(operator:list)
```

The following table lists the operators that are valid and their initialization values. The actual initialization value depends on the data type of the reduction variable.
The syntax of the `reduction` clause is as follows:

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

The following table lists the operators and intrinsic_procedure_names that are valid and their initialization values. The actual initialization value depends on the data type of the reduction variable.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initialization value</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>0</td>
</tr>
<tr>
<td>*</td>
<td>1</td>
</tr>
<tr>
<td>−</td>
<td>0</td>
</tr>
<tr>
<td>&amp;</td>
<td>~0</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>0</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>.and.</td>
<td>.true.</td>
</tr>
<tr>
<td>.or.</td>
<td>.false.</td>
</tr>
<tr>
<td>.eqv.</td>
<td>.true.</td>
</tr>
<tr>
<td>.neqv.</td>
<td>.false.</td>
</tr>
<tr>
<td>max</td>
<td>Most negative representable number in the reduction variable type</td>
</tr>
</tbody>
</table>
Fortran

### Description

The **reduction** clause can be used to perform some forms of recurrence calculations (involving mathematically associative and commutative operators) in parallel.

A private copy of each list item is created, one for each thread, as if the **private** clause had been used. The private copy is then initialized to the initialization value for the operator, as specified above. At the end of the region for which the **reduction** clause was specified, the original list item is updated by combining its original value with the final value of each of the private copies, using the operator specified. (The partial results of a subtraction reduction are added to form the final value.)

The value of the original list item becomes undefined when the first thread reaches the construct that specifies the clause and remains so until the reduction computation is complete. Normally, the computation will be complete at the end of the construct; however, if the **reduction** clause is used on a construct to which **nowait** is also applied, the value of the original list item remains undefined until a barrier synchronization has been performed to ensure that all threads have completed the reduction. Any concurrent uses or definitions of the original list item must be synchronized with the definition that occurs at the end of the construct, or at the subsequent barrier, to avoid race conditions.

The order in which the values are combined is unspecified. Therefore, comparing sequential and parallel runs, or comparing one parallel run to another (even if the number of threads used is the same), there is no guarantee that bit-identical results will be obtained or that side effects (such as floating point exceptions) will be identical.

\[ \text{min} \quad \text{Largest representable number in the reduction variable type} \]
\[ \text{iand} \quad \text{All bits on} \]
\[ \text{ior} \quad 0 \]
\[ \text{ieor} \quad 0 \]

**Note** – List items specified in a **reduction** clause are typically used in the enclosed region in certain forms.
A reduction is typically specified for statements of the form:

C/C++

\[
\begin{align*}
  x &= x \, \text{op} \, expr \\
  x &= \text{binop} \, expr \\
  x &= expr \, \text{op} \, x \quad \text{(except for subtraction)} \\
  x &= ++x \\
  x &= --x
\end{align*}
\]

where \( expr \) has scalar type and does not reference \( x \), \( \text{op} \) is not an overloaded operator, but one of \(+, \ast, -, \&\&, |\mid, \), and \( \text{binop} \) is not an overloaded operator, but one of \(+, \ast, -, \&\&, |\mid, \).

Fortran

A reduction using an operator is typically specified for statements of the form:

\[
\begin{align*}
  x &= x \, \text{op} \, expr \\
  x &= expr \, \text{op} \, x \quad \text{(except for subtraction)}
\end{align*}
\]

where \( \text{op} \) is \(+, \ast, -, \&\&, \mid\mid, \). the expression does not involve \( x \), and the reduction \( \text{op} \) is the last operation performed on the right hand side.

A reduction using an intrinsic is typically specified for statements of the form:

\[
\begin{align*}
  x &= \text{intr}(x, expr\_list) \\
  x &= \text{intr}(expr\_list, x)
\end{align*}
\]

where \( \text{intr} \) is \( \text{max, min, iand, ior, } \) or \( \text{ieor} \) and \( \text{expr\_list} \) is a comma separated list of expressions not involving \( x \).

For examples, see Section A.31 on page 176.

Restrictions

The restrictions to the \textbf{reduction} clause are as follows:
• A list item that appears in a reduction clause of a work-sharing construct must be shared in the parallel regions to which any of the work-sharing regions arising from the work-sharing construct bind.

• A list item that appears in a reduction clause of a parallel construct must not be privatized on any enclosed work-sharing construct if any of the work-sharing regions arising from the work-sharing construct bind to any of the parallel regions arising from the parallel construct.

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

C/C++
• The type of a list item that appears in a reduction clause must be valid for the reduction operator.

• Aggregate types (including arrays), pointer types and reference types may not appear in a reduction clause.

• A variable that appears in a reduction clause must not be const-qualified.

• The operator specified in a reduction clause cannot be overloaded with respect to the variables that appear in that clause.

Fortran
• The type of a list item that appears in a reduction clause must be valid for the reduction operator or intrinsic.

• A variable that appears in a reduction clause must be definable.

• A list item that appears in a reduction clause must be a named variable of intrinsic type.

• Fortran pointers, Cray pointers, assumed-size arrays and allocatable arrays may not appear in a reduction clause.

• Operators specified must be intrinsic operators and any intrinsic_procedure_name must refer to one of the allowed intrinsic procedures. Assignment to the reduction variables must be via intrinsic assignment. See Section A.31 on page 176 for examples.

2.8.4 Data Copying Clauses
This section describes the copyin clause (valid on the parallel directive and combined parallel work-sharing directives) and the copyprivate clause (valid on the single directive).
These clauses support the copying of data values from private or threadprivate objects on one thread, to the corresponding objects on other threads in the team.

The clauses accept a comma-separated list of list items (see Section 2.1 on page 18). All list items appearing in a clause must be visible, according to the scoping rules of the base language. Clauses may be repeated as needed, but a list item that specifies a given variable may not appear in more than one clause on the same directive.

### 2.8.4.1 copyin clause

**Summary**

The `copyin` clause provides a mechanism to copy the value of the master thread’s threadprivate variable to the threadprivate variable of each other member of the team executing the `parallel` region.

**Syntax**

The syntax of the `copyin` clause is as follows:

```
copyin(list)
```

**Description**

C/C++

The copy is done after the team is formed and prior to the start of execution of the `parallel` region. For variables of non-array type, the copy occurs as if by assignment. For a (possibly multi-dimensional) array of objects of non-array type, each element is copied as if by assignment from an element of the master thread’s array to the corresponding element of the other thread’s array. For class types, the copy assignment operator is invoked. The order in which copy assignment operators for different objects are called is unspecified.

---

Fortran

The copy is done, as if by assignment, after the team is formed and prior to the start of execution of the `parallel` region.

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, and 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 \texttt{POINTER} attribute, each copy becomes defined with the value
  of the master thread’s copy as if by intrinsic assignment.

For an example of the \texttt{copyin} clause, see Section A.32 on page 180.

\section*{Restrictions}

The restrictions on the \texttt{copyin} clause are as follows:

\begin{itemize}
  \item A list item that appears in a \texttt{copyin} clause must be threadprivate.
  \item A variable of class type (or array thereof) that appears in a \texttt{copyin} clause requires
    an accessible, unambiguous copy assignment operator for the class type.
  \item A list item that appears in a \texttt{copyin} clause must be threadprivate. Named variables
    appearing in a threadprivate common block may be specified: it is not necessary to
    specify the whole common block.
  \item A common block name that appears in a \texttt{copyin} clause must be declared to be a
    common block in the same scoping unit in which the \texttt{copyin} clause appears.
  \item Allocatable arrays may not appear in a \texttt{copyin} clause.
\end{itemize}

\subsection{copyprivate clause}

\section*{Summary}

The \texttt{copyprivate} clause provides a mechanism to use a private variable to broadcast
a value from one member of a team to the other members of the team.
Syntax

The syntax of the `copyprivate` clause is as follows:

```plaintext
copyprivate(list)
```

Description

The effect of the `copyprivate` clause on the specified list items occurs after the execution of the structured block associated with the `single` construct (see Section 2.5.3 on page 42), and before any of the threads in the team have left the barrier at the end of the construct.

**C/C++**

In all other threads in the team, each specified list item becomes defined with the value of the corresponding list item in the thread that executed the structured block. For variables of non-array type, the definition occurs as if by copy assignment. For a (possibly multi-dimensional) array of objects of non-array type, each element is copied as if by copy assignment from an element of the array in the thread that executed the structured block to the corresponding element of the array in the other threads. For class types, a copy assignment operator is invoked. The order in which copy assignment operators for different objects are called is unspecified.

**Fortran**

If a list item is not a pointer, then in all other threads in the team, the list item becomes defined (as if by assignment) with the value of the corresponding list item in the thread that executed the structured block. If the list item is a pointer, then in all other threads in the team, the list item becomes pointer associated (as if by pointer assignment) with the corresponding list item in the thread that executed the structured block.

For examples of the `copyprivate` clause, see Section A.33 on page 181.

**Note** – The `copyprivate` clause is an alternative to using a shared variable for the value when providing such a shared variable would be difficult (for example, in a recursion requiring a different variable at each level).

Restrictions

The restrictions to the `copyprivate` clause are as follows:
• All list items that appear in the `copyprivate` clause must be either threadprivate, or private in the enclosing context.

• A list item that appears in a `copyprivate` clause may not appear in a `private` or `firstprivate` clause on the `single` construct.

`C/C++`

• A variable of class type (or array thereof) that appears in a `copyprivate` clause requires an accessible unambiguous copy assignment operator for the class type.

`Fortran`

• A common block that appears in a `copyprivate` clause must be threadprivate.

• Allocatable arrays and assumed-size arrays may not appear in a `copyprivate` clause.

`Fortran`

2.9 Nesting of Regions

This section describes a set of restrictions on the nesting of regions. The restrictions on nesting are as follows:

• A work-sharing region may not be closely nested inside a work-sharing, `critical`, `ordered`, or `master` region.

• A `barrier` region may not be closely nested inside a work-sharing, `critical`, `ordered`, or `master` region.

• A `master` region may not be closely nested inside a work-sharing region.

• An `ordered` region may not be closely nested inside a `critical` region.

• An `ordered` region must be closely nested inside a loop region (or parallel loop region) with an `ordered` clause.

• A `critical` region may not be nested (closely or otherwise) inside a `critical` region with the same name. Note that this restriction is not sufficient to prevent deadlock.

For examples illustrating these rules, see Section A.14 on page 139, Section A.34 on page 185 and Section A.35 on page 187.
CHAPTER 3

Runtime Library Routines

This chapter describes the OpenMP API runtime library routines and is divided into the following sections:

- Runtime library definitions (Section 3.1 on page 90).
- Execution environment routines that can be used to control and query the parallel execution environment (Section 3.2 on page 91).
- Lock routines that can be used to synchronize access to data (Section 3.3 on page 102).
- Portable timer routines (Section 3.4 on page 108).

Throughout this chapter, true and false are used as generic terms to simplify the description of the routines.

C/C++

true means a nonzero integer value and false means an integer value of zero.

Fortran

true means a logical value of .TRUE. and false means a logical value of .FALSE..

Restrictions

The following restriction applies to all OpenMP runtime library routines:

- OpenMP runtime library routines may not be called from PURE or ELEMENTAL procedures.
## 3.1 Runtime Library Definitions

For each base language, a compliant implementation must supply a set of definitions for the OpenMP API runtime library routines and the special data types of their parameters. The set of definitions must contain a declaration for each OpenMP API runtime library routine and a declaration for the `simple lock` and `nestable lock` data types. In addition, each set of definitions may specify other implementation specific values.

---

### C/C++

The library routines are external functions with “C” linkage.

Prototypes for the C/C++ runtime library routines described in this chapter shall be provided in a header file named `omp.h`. This file defines the following:
- The prototypes of all the routines in the chapter.
- The type `omp_lock_t`.
- The type `omp_nest_lock_t`.

See Section D.1 on page 223 for an example of this file.

---

### Fortran

The OpenMP Fortran API runtime library routines are external procedures. The return values of these routines are of default kind, unless otherwise specified.

Interface declarations for the OpenMP Fortran runtime library routines described in this chapter shall be provided in the form of a Fortran `include` file named `omp_lib.h` or a Fortran 90 `module` named `omp_lib`. It is implementation defined whether the `include` file or the `module` file (or both) is provided.

These files define the following:
- The interfaces of all of the routines in this chapter.
- The `integer parameter omp_lock_kind`.
- The `integer parameter omp_nest_lock_kind`.
- The `integer parameter openmp_version` with a value `yyyymm` where `yyyy` and `mm` are the year and month designations of the version of the OpenMP Fortran API that the implementation supports. This value matches that of the C preprocessor macro `_OPENMP`, when a macro preprocessor is supported (see Section 2.2 on page 21).

See Section D.2 on page 225 and Section D.3 on page 227 for examples of these files.
It is implementation defined whether any of the OpenMP runtime library routines that take an argument are extended with a generic interface so arguments of different `KIND` type can be accommodated. See Appendix D.4 for an example of such an extension.

### 3.2 Execution Environment Routines

The routines described in this section affect and monitor threads, processors, and the parallel environment.

- the `omp_set_num_threads` routine.
- the `omp_get_num_threads` routine.
- the `omp_get_max_threads` routine.
- the `omp_get_thread_num` routine.
- the `omp_get_num_procs` routine.
- the `omp_in_parallel` routine.
- the `omp_set_dynamic` routine.
- the `omp_get_dynamic` routine.
- the `omp_set_nested` routine.
- the `omp_get_nested` routine.

#### 3.2.1 omp_set_num_threads

**Summary**

The `omp_set_num_threads` routine affects the number of threads to be used for subsequent `parallel` regions that do not specify a `num_threads` clause, by setting the value of the `nthreads-var` internal control variable.
Constraints on Arguments

The value of the argument passed to this routine must evaluate to a positive integer.

Binding

When called from the sequential part of the program, the binding thread set for an `omp_set_num_threads` region is the encountering thread. When called from within any explicit `parallel` region, the binding thread set (and binding region, if required) for the `omp_set_num_threads` region is implementation defined.

Effect

The effect of this routine is to set the value of the `nthreads-var` internal control variable to the value specified in the argument.

See Section 2.4.1 on page 29 for the rules governing the number of threads used to execute a `parallel` region.

If the number of threads requested exceeds the number the implementation can support, or is not a positive integer, the behavior of this routine is implementation defined.

For an example of the `omp_set_num_threads` routine, see Section A.36 on page 193.

Calling Context Rules

This routine has the described effect only when called from the sequential part of the program. If it is called from any `parallel` region, the behavior of this routine is implementation defined.
**3.2.2 omp_get_num_threads**

**Summary**

The `omp_get_num_threads` routine returns the number of threads in the current team.

**Format**

```c
int omp_get_num_threads(void);
```

**Binding**

The binding thread set for an `omp_get_num_threads` region is the current team. The binding region for an `omp_get_num_threads` region is the innermost enclosing parallel region. The return value of this routine depends on the characteristics of the team executing the binding parallel region.

**Effect**

The `omp_get_num_threads` routine returns the number of threads in the team executing the parallel region to which the routine region binds. If called from the sequential part of a program, this routine returns 1. For examples, see Section A.37 on page 195.

See Section 2.4.1 on page 29 for the rules governing the number of threads used to execute a parallel region.

**Cross References**

- parallel construct, see Section 2.4 on page 26.
3.2.3 omp_get_max_threads

Summary

The **omp_get_max_threads** routine returns the value of the *nthreads-var* internal control variable, which is used to determine the number of threads that would form the new team, if an active **parallel** region without a **num_threads** clause were to be encountered at that point in the program.

Format

```
int omp_get_max_threads(void);
```

```
integer function omp_get_max_threads()
```

Binding

When called from the sequential part of the program, the binding thread set for an **omp_get_max_threads** region is the encountering thread. When called from within any explicit **parallel** region, the binding thread set (and binding region, if required) for the **omp_get_max_threads** region is implementation defined.

Effect

The following expresses a lower bound on the value of **omp_get_max_threads**: the number of threads that would be used to form a team if an active **parallel** region without a **num_threads** clause were to be encountered at that point in the program is less than or equal to the value returned by **omp_get_max_threads**.

See Section 2.4.1 on page 29 for the rules governing the number of threads used to execute a **parallel** region.

Note – The return value of **omp_get_max_threads** routine can be used to dynamically allocate sufficient storage for all threads in the team formed at the subsequent active **parallel** region.
Cross References

• Internal control variables, see Section 2.3 on page 24.
• parallel construct, see Section 2.4 on page 26.
• num_threads clause, see Section 2.4 on page 26.

3.2.4 omp_get_thread_num

Summary

The omp_get_thread_num routine returns the thread number, within the team, of the thread executing the parallel region from which omp_get_thread_num is called.

Format

C/C++

int omp_get_thread_num(void);

Fortran

integer function omp_get_thread_num()

Binding

The binding thread set for an omp_get_thread_num region is the current team. The binding region for an omp_get_thread_num region is the innermost enclosing parallel region. The return value of this routine depends on the characteristics of the team executing the binding parallel region.

Effect

The omp_get_thread_num routine returns the thread number of the current thread, within the team executing the parallel region to which the routine region binds. The thread number is an integer between 0 and one less than the value returned by omp_get_num_threads, inclusive. The thread number of the master thread of the team is 0. The routine returns 0 if it is called from the sequential part of a program.
Cross References

- `omp_get_num_threads` routine, see Section 3.2.2 on page 93.

### 3.2.5 omp_get_num_procs

#### Summary

The `omp_get_num_procs` routine returns the number of processors available to the program.

#### Format

```c
int omp_get_num_procs(void);
```

#### Binding

The binding thread set for an `omp_get_num_procs` region is all threads. The effect of executing this routine is not related to any specific region corresponding to any construct or API routine.

#### Effect

The `omp_get_num_procs` routine returns the number of processors that are available to the program at the time the routine is called.

### 3.2.6 omp_in_parallel

#### Summary

The `omp_in_parallel` routine returns `true` if the call to the routine is enclosed by an active `parallel` region; otherwise, it returns `false`.
Chapter 3 Runtime Library Routines

Format

C/C++

```c
int omp_in_parallel(void);
```

Fortran

```fortran
logical function omp_in_parallel()
```

Binding

The binding thread set for an `omp_in_parallel` region is all threads. The effect of executing this routine is not related to any specific parallel region but instead depends on the state of all enclosing parallel regions.

Effect

`omp_in_parallel` returns the logical OR of the if clauses of all enclosing parallel regions. If a parallel region does not have an if clause, this is equivalent to `if(true)`.

If the routine is called from the sequential part of the program, then `omp_in_parallel` returns `false`.

Cross References

• if clause, see Section 2.4.1 on page 29.

3.2.7 omp_set_dynamic

Summary

The `omp_set_dynamic` routine enables or disables dynamic adjustment of the number of threads available for the execution of parallel regions by setting the value of the `dyn-var` internal control variable.
When called from the sequential part of the program, the binding thread set for an 
`omp_set_dynamic` region is the encountering thread. When called from within any 
explicit `parallel` region, the binding thread set (and binding region, if required) for 
the `omp_set_dynamic` region is implementation defined.

**Effect**

For implementations that provide the ability to dynamically adjust the number of 
threads, if the argument to `omp_set_dynamic` evaluates to `true`, dynamic adjustment 
of the number of threads is enabled; otherwise, dynamic adjustment is disabled.

For implementations that do not provide the ability to dynamically adjust the number of 
threads, this routine has no effect: the value of `dyn-var` remains `false`.

For an example of the `omp_set_dynamic` routine, see Section A.36 on page 193.

See Section 2.4.1 on page 29 for the rules governing the number of threads used to 
execute a `parallel` region.

**Calling Context Rules**

The `omp_set_dynamic` routine has the described effect only when called from the 
sequential part of the program. If called from within any explicit `parallel` region, the 
behavior of this routine is implementation defined.

**Cross References:**

- Internal control variables, see Section 2.3 on page 24.
- `omp_get_num_threads` routine, see Section 3.2.2 on page 93.
3.2.8 \texttt{omp\_get\_dynamic}

Summary

The \texttt{omp\_get\_dynamic} routine returns the value of the \textit{dyn-var} internal control variable, which determines whether dynamic adjustment of the number of threads is enabled or disabled.

Format

\begin{verbatim}
int omp_get_dynamic(void);
\end{verbatim}

Binding

When called from the sequential part of the program, the binding thread set for an \texttt{omp\_get\_dynamic} region is the encountering thread. When called from within any explicit \texttt{parallel} region, the binding thread set (and binding region, if required) for the \texttt{omp\_get\_dynamic} region is implementation defined.

Effect

This routine returns \texttt{true} if dynamic adjustment of the number of threads is enabled; it returns \texttt{false}, otherwise.

If the implementation does not provide the ability to dynamically adjust the number of threads, then this routine always returns \texttt{false}.

See Section 2.4.1 on page 29 for the rules governing the number of threads used to execute a \texttt{parallel} region.

Cross References

- Internal control variables, see Section 2.3 on page 24.
3.2.9  

omp_set_nested

Summary

The `omp_set_nested` routine enables or disables nested parallelism, by setting the `nest-var` internal control variable.

Format

```c
void omp_set_nested(int nested);
```

Binding

When called from the sequential part of the program, the binding thread set for an `omp_set_nested` region is the encountering thread. When called from within any explicit `parallel` region, the binding thread set (and binding region, if required) for the `omp_set_nested` region is implementation defined.

Effect

For implementations that support nested parallelism, if the argument to `omp_set_nested` evaluates to `true`, nested parallelism is enabled; otherwise, nested parallelism is disabled.

For implementations that do not support nested parallelism, this routine has no effect: the value of `nest-var` remains `false`.

See Section 2.4.1 on page 29 for the rules governing the number of threads used to execute a `parallel` region.

Calling Context Rules

The `omp_set_nested` routine has the described effect only when called from the sequential part of the program. If called from within any explicit `parallel` region, the behavior of this routine is implementation defined.
Cross References

- Internal control variables, see Section 2.3 on page 24.

3.2.10 omp_get_nested

Summary

The `omp_get_nested` routine returns the value of the `nest-var` internal control variable, which determines if nested parallelism is enabled or disabled.

Format

```c
int omp_get_nested(void);
```

`Fortran`

```fortran
logical function omp_get_nested()
```

Binding

When called from the sequential part of the program, the binding thread set for an `omp_get_nested` region is the encountering thread. When called from within any explicit `parallel` region, the binding thread set (and binding region, if required) for the `omp_get_nested` region is implementation defined.

Effect

This routine returns `true` if nested parallelism is enabled; it returns `false`, otherwise.

If an implementation does not support nested parallelism, this routine always returns `false`.

See Section 2.4.1 on page 29 for the rules governing the number of threads used to execute a `parallel` region.

Cross References

- Internal control variables, see Section 2.3 on page 24.
3.3 Lock Routines

The OpenMP runtime library includes a set of general-purpose lock routines that can be used for synchronization. These general-purpose lock routines operate on OpenMP locks that are represented by OpenMP lock variables. An OpenMP lock variable must be accessed only through the routines described in this section.

An OpenMP lock may be in one of the following states: uninitialized, unlocked, or locked. If a lock is in the unlocked state, a thread may set the lock, which changes its state to locked. The thread which sets the lock is then said to own the lock. A thread which owns a lock may unset that lock, returning it to the unlocked state. A thread may not set or unset a lock which is owned by another thread.

Two types of locks are supported: simple locks and nestable locks. A nestable lock may be set multiple times by the same thread before being unset; a simple lock may not be set if it is already owned by the thread trying to set it. 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.

Constraints on the state and ownership of the lock accessed by each of the lock routines are described with the routine. If these constraints are not met, the behavior of the routine is unspecified.

The OpenMP lock routines access a lock variable in such a way that they always read and update the most current value of the lock variable. Therefore, it is not necessary for an OpenMP program to include explicit flush directives to ensure that the lock variable’s value is consistent among different threads.

See Section A.39 on page 198 and Section A.40 on page 200, for examples of using the simple and the nestable lock routines, respectively.

Binding

The binding thread set for all lock routine regions is all threads. For each OpenMP lock, the lock routine effects relate to all threads which execute the routines, without regard to which team(s) the threads belong.
Simple Lock Routines

The type `omp_lock_t` is an object type capable of representing a simple lock. For the following routines, a lock variable must be of `omp_lock_t` type. All simple lock routines require an argument that is a pointer to a variable of type `omp_lock_t`.

For the following routines, `svar` must be an integer variable of `kind=omp_lock_kind`.

The simple lock routines are as follows:

- The `omp_init_lock` routine initializes a simple lock.
- The `omp_destroy_lock` routine uninitializes a simple lock.
- The `omp_set_lock` routine waits until a simple lock is available, and then sets it.
- The `omp_unset_lock` routine unsets a simple lock.
- The `omp_test_lock` routine tests a simple lock, and sets it if it is available.

Nestable Lock Routines:

The type `omp_nest_lock_t` is an object type capable of representing a nestable lock. For the following routines, a lock variable must be of `omp_nest_lock_t` type. All nestable lock routines require an argument that is a pointer to a variable of type `omp_nest_lock_t`.

For the following routines, `nvar` must be an integer variable of `kind=omp_nest_lock_kind`.

The nestable lock routines are as follows:

- The `omp_init_nest_lock` routine initializes a nestable lock.
- The `omp_destroy_nest_lock` routine uninitializes a nestable lock.
- The `omp_set_nest_lock` routine waits until a nestable lock is available, and then sets it.
3.3.1 omp_init_lock and omp_init_nest_lock

Summary
These routines provide the only means of initializing an OpenMP lock.

**Format**

**C/C++**
```c
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

**Fortran**
```fortran
subroutine omp_init_lock(svar)
integer (kind=omp_lock_kind) svar

subroutine omp_init_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

**Constraints on Arguments**
A lock accessed by either routine must be in the uninitialized state.

**Effect**
The effect of these routines is to initialize the lock to the unlocked state (that is, no thread owns the lock). In addition, the nesting count for a nestable lock is set to zero.

For an example of the `omp_init_lock` routine, see Section A.38 on page 197.
### 3.3.2 omp_destroy_lock and omp_destroy_nest_lock

#### Summary
These routines ensure that the OpenMP lock is uninitialized.

#### Format

**C/C++**

```c
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

**Fortran**

```fortran
subroutine omp_destroy_lock(svar)
   integer (kind=omp_lock_kind) svar
end subroutine

subroutine omp_destroy_nest_lock(nvar)
   integer (kind=omp_nest_lock_kind) nvar
end subroutine
```

#### Constraints on Arguments
A lock accessed by either routine must be in the unlocked state.

#### Effect
The effect of these routines is to change the state of the lock to uninitialized.

### 3.3.3 omp_set_lock and omp_set_nest_lock

#### Summary
These routines provide a means of setting an OpenMP lock. The calling thread blocks until the lock is set.
Constraints on Arguments

A lock accessed by either routine must not be in the uninitialized state. A simple lock accessed by `omp_set_lock` which is in the locked state must not be owned by the thread executing the routine.

Effect

Each of these routines blocks the thread executing the routine until the specified lock is available and then sets the lock.

A simple lock is available if it is unlocked. Ownership of the lock is granted to the thread executing the routine.

A nestable lock is available if it is unlocked or if it is already owned by the thread executing the routine. The thread executing the routine is granted, or retains, ownership of the lock, and the nesting count for the lock is incremented.

3.3.4 `omp_unset_lock` and `omp_unset_nest_lock`

Summary

These routines provide the means of unsetting an OpenMP lock.
Chapter 3 Runtime Library Routines

3.3.5 \texttt{omp_test_lock} and \texttt{omp_test_nest_lock}

Summary

These routines attempt to set an OpenMP lock but do not block execution of the thread executing the routine.
### Constraints on Arguments

A lock accessed by either routine must not be in the uninitialized state. A simple lock accessed by `omp_test_lock` which is in the locked state must not be owned by the thread executing the routine.

### Effect

These routines attempt to set a lock in the same manner as `omp_set_lock` and `omp_set_nest_lock`, except that they do not block execution of the thread executing the routine.

For a simple lock, the `omp_test_lock` routine returns `true` if the lock is successfully set; otherwise, it returns `false`.

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

### 3.4 Timing Routines

The routines described in this section support a portable wall clock timer.

- the `omp_get_wtime` routine.
- the `omp_get_wtick` routine.
3.4.1 omp_get_wtime

Summary

The omp_get_wtime routine returns elapsed wall clock time in seconds.

Format

C/C++

double omp_get_wtime(void);

Fortran

double precision function omp_get_wtime()

Binding

The binding thread set for an omp_get_wtime region is the encountering thread. The routine’s return value is not guaranteed to be consistent across any set of threads.

Effect

The omp_get_wtime routine returns a value equal to the elapsed wall clock time in seconds since some “time in the past”. The actual “time in the past” is arbitrary, but it is guaranteed not to change during the execution of the application program. The times returned are “per-thread times”, so they are not required to be globally consistent across all the threads participating in an application.
Note – It is anticipated that the routine will be used to measure elapsed times as shown in the following example:

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

```fortran
DOUBLE PRECISION START, END
START = omp_get_wtime()
... work to be timed ...
END = omp_get_wtime()
PRINT *, "Work took", END - START, "seconds"
```

### 3.4.2 `omp_get_wtick`

**Summary**

The `omp_get_wtick` routine returns the precision of the timer used by `omp_get_wtime`. 
Format

```c
double omp_get_wtick(void);
```

Binding
The binding thread set for an `omp_get_wtick` region is the encountering thread. The routine’s return value is not guaranteed to be consistent across any set of threads.

Effect
The `omp_get_wtick` routine returns a value equal to the number of seconds between successive clock ticks of the timer used by `omp_get_wtime`.
This chapter describes the OpenMP environment variables that specify the settings of the internal control variables that affect the execution of OpenMP programs (see Section 2.3 on page 24). The names of the environment variables must be uppercase. The values assigned to the environment variables are case insensitive and may have leading and trailing white space. Modifications to the environment variables after the program has started, even if modified by the program itself, are ignored by the OpenMP implementation. However, the settings of the internal control variables can be modified during the execution of the OpenMP program by the use of the appropriate directive clauses or OpenMP API routines.

The environment variables are as follows:

- **OMP_SCHEDULE** sets the run-sched-var internal control variable for the runtime schedule type and chunk size.
- **OMP_NUM_THREADS** sets the nthreads-var internal control variable for the number of threads to use for parallel regions.
- **OMP_DYNAMIC** sets the dyn-var internal control variable for the dynamic adjustment of threads to use for parallel regions.
- **OMP_NESTED** sets the nest-var internal control variable to enable or disable nested parallelism.

The examples in this chapter only demonstrate how these variables might be set in Unix C shell (csh) environments. In Korn shell (ksh) and DOS environments the actions are similar, as follows:

- **csh:**

  ```
  setenv OMP_SCHEDULE "dynamic"
  ```

- **ksh:**

  ```
  export OMP_SCHEDULE="dynamic"
  ```
4.1 OMP_SCHEDULE

The OMP_SCHEDULE environment variable controls the schedule type and chunk size of all loop directives that have the schedule type runtime, by setting the value of the run-sched-var internal control variable.

The value of this environment variable takes the form:

\[ \text{type}[, \text{chunk}] \]

where

- type is one of static, dynamic or guided
- chunk is an optional positive integer which specifies the chunk size

If chunk is present, there may be white space on either side of the "\,". See Section 2.5.1 on page 33 for a detailed description of the schedule types.

If OMP_SCHEDULE is not set, the initial value of the run-sched-var internal control variable is implementation defined.

Example:

```
setenv OMP_SCHEDULE "guided,4"
setenv OMP_SCHEDULE "dynamic"
```

Cross References:

- Internal control variables, see Section 2.3 on page 24.
- Loop construct, see Section 2.5.1 on page 33.
- Parallel loop construct, see Section 2.6.1 on page 47.
4.2 OMP_NUM_THREADS

The OMP_NUM_THREADS environment variable sets the number of threads to use for parallel regions by setting the initial value of the nthreads-var internal control variable. See Section 2.3 for a comprehensive set of rules about the interaction between the OMP_NUM_THREADS environment variable, the num_threads clause, the omp_set_num_threads library routine and dynamic adjustment of threads.

The value of this environment variable must be a positive integer. The behavior of the program is implementation defined if the requested value of OMP_NUM_THREADS is greater than the number of threads an implementation can support, or if the value is not a positive integer.

If the OMP_NUM_THREADS environment variable is not set, the initial value of the nthreads-var internal control variable is implementation defined.

The nthreads-var internal control variable can be modified using the omp_set_num_threads library routine. The number of threads in the current team can be queried using the omp_get_num_threads library routine. The maximum number of threads in future teams can be queried using the omp_get_max_threads library routine.

Example:

```
setenv OMP_NUM_THREADS 16
```

Cross References:

- Internal control variables, see Section 2.3 on page 24.
- num_threads clause, Section 2.4 on page 26.
- omp_set_num_threads routine, see Section 3.2.1 on page 91.
- omp_get_num_threads routine, see Section 3.2.2 on page 93.
- omp_get_max_threads routine, see Section 3.2.3 on page 94.
- omp_get_dynamic routine, see Section 3.2.8 on page 99.
4.3 OMP_DYNAMIC

The OMP_DYNAMIC environment variable controls dynamic adjustment of the number of threads to use for executing parallel regions by setting the initial value of the dyn-var internal control variable. The value of this environment variable must be true or false. If the environment variable is set to true, the OpenMP implementation may adjust the number of threads to use for executing parallel regions in order to optimize the use of system resources. If the environment variable is set to false, the dynamic adjustment of the number of threads is disabled.

If the OMP_DYNAMIC environment variable is not set, the initial value of the dyn-var internal control variable is implementation defined.

The dyn-var internal control variable can be modified by calling the omp_set_dynamic library routine. The current value of dyn-var can be queried using the omp_get_dynamic library routine.

Example:

```
setenv OMP_DYNAMIC true
```

Cross References:
- Internal control variables, see Section 2.3 on page 24.
- omp_get_num_threads routine, see Section 3.2.2 on page 93.
- omp_set_dynamic routine, see Section 3.2.7 on page 97.
- omp_get_dynamic routine, see Section 3.2.8 on page 99.

4.4 OMP_NESTED

The OMP_NESTED environment variable controls nested parallelism by setting the initial value of the nest-var internal control variable. The value of this environment variable must be true or false. If the environment variable is set to true, nested parallelism is enabled; if set to false, nested parallelism is disabled.

If the OMP_NESTED environment variable is not set, the initial value of the nest-var internal control variable is false.
The *nest-var* internal control variable can be modified by calling the
`omp_set_nested` library routine. The current value of *nest-var* can be queried using
the `omp_get_nested` library routine.

Example:

```
setenv OMP_NESTED false
```

Cross References:
- Internal control variables, see Section 2.3 on page 24.
- `omp_set_nested` routine, see Section 3.2.9 on page 100.
Examples

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

C/C++

A statement following a directive is compound only when necessary, and a non-
compound statement is indented with respect to a directive preceding it.

A.1 A Simple Parallel Loop

The following example demonstrates how to parallelize a simple loop using the parallel
loop construct (Section 2.6.1 on page 47). The loop iteration variable is private by
default, so it is not necessary to specify it explicitly in a private clause.

Example A.1.1c

void a1(int n, float *a, float *b)
{
    int i;

    #pragma omp parallel for
    for (i=1; i<n; i++) /* i is private by default */
        b[i] = (a[i] + a[i-1]) / 2.0;
}

Example A.1.1f

SUBROUTINE A1(N, A, B)
A.2 The OpenMP Memory Model

In the following example, at Print 1, the value of \( x \) could be either 2 or 5, depending on the timing of the threads, and the implementation of the assignment to \( x \). There are two reasons that the value at Print 1 might not be 5. First, Print 1 might be executed before the assignment to \( x \) is executed. Second, even if Print 1 is executed after the assignment, the value 5 is not guaranteed to be seen by thread 1 because a flush may not have been executed by thread 0 since the assignment.

The barrier after Print 1 contains implicit flushes on all threads, as well as a thread synchronization, so the programmer is guaranteed that the value 5 will be printed by both Print 2 and Print 3.

```c
#include <stdio.h>
#include <omp.h>

int main()
{
    int x;
    x = 2;
    #pragma omp parallel num_threads(2) shared(x)
    {
        if (omp_get_thread_num() == 0) {
            x = 5;
        } else {
            /* Print 1: the following read of x has a race */
            printf("1: Thread# %d: x = %d\n", omp_get_thread_num(), x);
        }
    }
```
```c
#pragma omp barrier

if (omp_get_thread_num() == 0) {
    /* Print 2 */
    printf("2: Thread\%d: x = \%d\n", omp_get_thread_num(), x);
} else {
    /* Print 3 */
    printf("3: Thread\%d: x = \%d\n", omp_get_thread_num(), x);
}

return 0;
```

```fortran

Example A.2.1f

PROGRAM A2
    INCLUDE "omp_lib.h" ! or USE OMP_LIB
    INTEGER X

    X = 2
    !$OMP PARALLEL NUM_THREADS(2) SHARED(X)

    IF (OMP_GET_THREAD_NUM() .EQ. 0) THEN
        X = 5
    ELSE
        ! PRINT 1: The following read of x has a race
        PRINT *, "1: THREAD\", OMP_GET_THREAD_NUM(), "X = ", X
    ENDIF

    !$OMP BARRIER

    IF (OMP_GET_THREAD_NUM() .EQ. 0) THEN
        ! PRINT 2
        PRINT *, "2: THREAD\", OMP_GET_THREAD_NUM(), "X = ", X
    ELSE
        ! PRINT 3
        PRINT *, "3: THREAD\", OMP_GET_THREAD_NUM(), "X = ", X
    ENDIF

    !$OMP END PARALLEL

END PROGRAM A2
```
A.3 Conditional Compilation

The following example illustrates the use of conditional compilation using the OpenMP macro `_OPENMP` (Section 2.2 on page 21). With OpenMP compilation, the `_OPENMP` macro becomes defined.

**Example A.3.1c**
```
#include <stdio.h>

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

        return 0;
}
```

The following example illustrates the use of the conditional compilation sentinel (see Section 2.2 on page 21). With OpenMP compilation, the conditional compilation sentinel `!$` is recognized and treated as two spaces. In fixed form source, statements guarded by the sentinel must start after column 6.

**Example A.3.1f**
```
PROGRAM A3
    C234567890
    !$    PRINT *, "Compiled by an OpenMP-compliant implementation."
END PROGRAM A3
```

A.4 The parallel Construct

The parallel construct (Section 2.4 on page 26) 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:

```c
#include <omp.h>

void subdomain(float *x, int istart, int ipoints)
{
    int i;

    for (i = 0; i < ipoints; i++)
        x[istart+i] = 123.456;
}

void sub(float *x, int npoints)
{
    int iam, nt, ipoints, istart;

#pragma omp parallel default(shared) private(iam,nt,ipoints,istart)
{
    iam = omp_get_thread_num();
    nt = omp_get_num_threads();
    ipoints = npoints / nt;  /* size of partition */
    istart = iam * ipoints;  /* starting array index */
    if (iam == nt-1)          /* last thread may do more */
        ipoints = npoints - istart;
    subdomain(x, istart, ipoints);
}

int main()
{
    float array[10000];

    sub(array, 10000);

    return 0;
}
```
Example A.4.1f

SUBROUTINE SUBDOMAIN(X, ISTART, IPOINTS)
   INTEGER ISTART, IPOINTS
   REAL X(*)

   INTEGER I

   DO 100 I=1,IPOINTS
      X(ISTART+I) = 123.456
   100      CONTINUE

END SUBROUTINE SUBDOMAIN

SUBROUTINE SUB(X, NPOINTS)
   INCLUDE "omp_lib.h"     ! or USE OMP_LIB

   REAL X(*)
   INTEGER NPOINTS

   INTEGER IAM, NT, IPOINTS, ISTART

   !$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,NPOINTS)
      IAM = OMP_GET_THREAD_NUM()
      NT = OMP_GET_NUM_THREADS()
      IPOINTS = NPOINTS/NT
      ISTART = IAM * IPOINTS
      IF (IAM .EQ. NT-1) THEN
         IPOINTS = NPOINTS - ISTART
      ENDIF
      CALL SUBDOMAIN(X,ISTART,IPOINTS)
   !$OMP END PARALLEL

END SUBROUTINE SUB

PROGRAM A4

   REAL ARRAY(10000)

   CALL SUB(ARRAY, 10000)

END PROGRAM A4
A.5 The **num_threads** Clause

The following example demonstrates the **num_threads** clause (Section 2.4 on page 26). The parallel region is executed with a maximum of 10 threads.

```c
#include <omp.h>
int main()
{
    omp_set_dynamic(1);
    #pragma omp parallel num_threads(10)
    {
        /* do work here */
    }
    return 0;
}
```

```fortran
PROGRAM A5
    INCLUDE "omp_lib.h" ! or USE OMP_LIB
    CALL OMP_SET_DYNAMIC(.TRUE.)

    !$OMP PARALLEL NUM_THREADS(10)
    ! do work here
    !$OMP END PARALLEL
END PROGRAM A5
```

A.6 Fortran Restrictions on the **do** Construct

If an **end do** directive follows a **do-construct** in which several **DO** statements share a **DO** termination statement, then a **do** directive can only be specified for the first (i.e. outermost) of these **DO** statements. For more information, see Section 2.5.1 on page 33. The following example contains correct usages of loop constructs:
Example A.6.1f

```fortran
SUBROUTINE WORK(I, J)
   INTEGER I, J
END SUBROUTINE WORK

SUBROUTINE A6_GOOD()
   INTEGER I, J
   REAL A(1000)
   DO 100 I = 1,10
      !$OMP DO
      DO 100 J = 1,10
         CALL WORK(I,J)
      100 CONTINUE ! !$OMP ENDDO implied here
   !$OMP DO
   DO 200 J = 1,10
      200       A(I) = I + 1
   !$OMP ENDDO
   !$OMP DO
   DO 300 I = 1,10
      DO 300 J = 1,10
         CALL WORK(I,J)
      300     CONTINUE
   !$OMP ENDDO
END SUBROUTINE A6_GOOD
```

The following example is non-conforming because the matching `do` directive for the `end do` does not precede the outermost loop:

Example A.6.2f

```fortran
SUBROUTINE WORK(I, J)
   INTEGER I, J
END SUBROUTINE WORK

SUBROUTINE A6_WRONG
   INTEGER I, J
   DO 100 I = 1,10
      !$OMP DO
      DO 100 J = 1,10
         CALL WORK(I,J)
      100 CONTINUE
```

126  OpenMP API • Version 2.5  May 2005
A.7 Fortran Private Loop Iteration Variables

In general loop iteration variables will be private, when used in the do-loop of a do and parallel do construct or in sequential loops in a parallel construct (see Section 2.5.1 on page 33 and Section 2.8.1 on page 63). In the following example of a sequential loop in a parallel construct the loop iteration variable \( I \) will be private.

Example A.7.1f

```fortran
SUBROUTINE A7_1(A,N)
   INCLUDE "omp_lib.h" ! or USE OMP_LIB
   REAL A(*)
   INTEGER I, MYOFFSET, N
   !$OMP PARALLEL PRIVATE(MYOFFSET)
   MYOFFSET = OMP_GET_THREAD_NUM()*N
   DO I = 1, N
      A(MYOFFSET+I) = FLOAT(I)
   ENDDO
   !$OMP END PARALLEL
END SUBROUTINE A7_1
```

In exceptional cases, loop iteration variables can be made shared, as in the following example:

Example A.7.2f

```fortran
SUBROUTINE A7_2(A,B,N,I1,I2)
   REAL A(*), B(*)
   INTEGER I1, I2, N
   !$OMP PARALLEL SHARED(A,B,I1,I2)
   !$OMP SECTIONS
   !$OMP SECTION
      DO I1 = I1, N
      !...
   !$OMP END SECTION
   !$OMP SECTIONS
   !$OMP SECTION
      DO I2 = I2, N
      !...
   !$OMP END SECTION
   !$OMP END PARALLEL
END SUBROUTINE A7_2
```
IF (A(I1).NE.0.0) EXIT
ENDDO
!$OMP SECTION
DO I2 = I2, N
   IF (B(I2).NE.0.0) EXIT
ENDDO
!$OMP END SECTIONS
!$OMP SINGLE
   IF (I1.LE.N) PRINT *, 'ITEMS IN A UP TO ', I1, ' ARE ALL ZERO.'
   IF (I2.LE.N) PRINT *, 'ITEMS IN B UP TO ', I2, ' ARE ALL ZERO.'
!$OMP END SINGLE
!$OMP END PARALLEL
END SUBROUTINE A7_2

Note however that the use of shared loop iteration variables can easily lead to race
conditions.

A.8 The nowait clause

If there are multiple independent loops within a parallel region, you can use the
nowait clause (see Section 2.5.1 on page 33) to avoid the implied barrier at the end of
the loop construct, as follows:

--- C/C++ ---

#include <math.h>

void a8(int n, int m, float *a, float *b, float *y, float *z)
{
    int i;
    #pragma omp parallel
    {
        #pragma omp for nowait
        for (i=1; i<n; i++)
            b[i] = (a[i] + a[i-1]) / 2.0;

        #pragma omp for nowait
        for (i=0; i<m; i++)
            y[i] = sqrt(z[i]);
    }
}
\[ \text{Fortran} \]

\textbf{Example A.8.1f}

\begin{verbatim}
SUBROUTINE A8(N, M, A, B, Y, Z)
    INTEGER N, M
    REAL A(*), B(*), Y(*), Z(*)
    INTEGER I

    !$OMP PARALLEL
    !$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

END SUBROUTINE A8
\end{verbatim}

\[ \text{Fortran} \]

\[ \text{C/C++} \]

\textbf{A.9 The parallel sections Construct}

In the following example (for Section 2.5.2 on page 39) routines \textit{xaxis}, \textit{yaxis}, and \textit{zaxis} can be executed concurrently. The first \texttt{section} directive is optional. Note that all \texttt{section} directives need to appear in the \texttt{parallel sections} construct.

\[ \text{C/C++} \]
A.10 The single Construct

The following example demonstrates the single construct (Section 2.5.3 on page 42). In the example, only one thread prints each of the progress messages. All other threads will skip the single region and stop at the barrier at the end of the single construct until all threads in the team have reached the barrier. If other threads can proceed without waiting for the thread executing the single region, a nowait clause can be specified, as is done in the third single construct in this example. The user must not make any assumptions as to which thread will execute a single region.
Example A.10.1c

```c
#include <stdio.h>

void work1() {}
void work2() {}

void a10()
{
    #pragma omp parallel
    {
        #pragma omp single
        printf("Beginning work1.\n");
        work1();

        #pragma omp single
        printf("Finishing work1.\n");

        #pragma omp single nowait
        printf("Finished work1 and beginning work2.\n");

        work2();
    }
}
```

Example A.10.1f

```fortran
SUBROUTINE WORK1()
END SUBROUTINE WORK1

SUBROUTINE WORK2()
END SUBROUTINE WORK2

PROGRAM A10
!
$OMP PARALLEL
!
$OMP SINGLE
    print *, "Beginning work1."
$OMP END SINGLE
!
CALL WORK1()
!
$OMP SINGLE
    print *, "Finishing work1."
$OMP END SINGLE
```
A.11 The workshare Construct

The following are examples of the workshare construct (see Section 2.5.4 on page 44).

In the following example, workshare spreads work across the threads executing the parallel region, and there is a barrier after the last statement. Implementations must enforce Fortran execution rules inside of the workshare block.

Example A.11.1f

```fortran
SUBROUTINE A11_1(AA, BB, CC, DD, EE, FF, N)
    INTEGER N
    REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N), EE(N,N), FF(N,N)
!
    !$OMP PARALLEL
    !$OMP WORKSHARE
    AA = BB
    CC = DD
    EE = FF
    !$OMP END WORKSHARE
    !$OMP END PARALLEL

END SUBROUTINE A11_1
```

In the following example, the barrier at the end of the first workshare region is eliminated with a nowait clause. Threads doing CC = DD immediately begin work on EE = FF when they are done with CC = DD.
Fortran (cont.)

Example A.11.2f

```fortran
SUBROUTINE A11_2(AA, BB, CC, DD, EE, FF, N)
  INTEGER N
  REAL AA(N,N), BB(N,N), CC(N,N)
  REAL DD(N,N), EE(N,N), FF(N,N)

  !$OMP PARALLEL
  !$OMP WORKSHARE
  AA = BB
  CC = DD
  !$OMP END WORKSHARE NOWAIT
  !$OMP WORKSHARE
  EE = FF
  !$OMP END WORKSHARE
  !$OMP END PARALLEL
END SUBROUTINE A11_2
```

The following example shows the use of an atomic directive inside a workshare construct. The computation of $\text{SUM}(\text{AA})$ is workshared, but the update to $I$ is atomic.

Example A.11.3f

```fortran
SUBROUTINE A11_3(AA, BB, CC, DD, N)
  INTEGER N
  REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
  REAL R

  R=0
  !$OMP PARALLEL
  !$OMP WORKSHARE
  AA = BB
  !$OMP ATOMIC
  R = R + SUM(AA)
  CC = DD
  !$OMP END WORKSHARE
  !$OMP END PARALLEL
END SUBROUTINE A11_3
```

Fortran WHERE and FORALL statements are compound statements, 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. The following example shows the use of a WHERE statement in a workshare construct.
Each task gets worked on in order by the threads:

\[
\begin{align*}
\text{AA} &= \text{BB} \quad \text{then} \\
\text{CC} &= \text{DD} \quad \text{then} \\
\text{EE} &= \text{.ne. 0} \quad \text{then} \\
\text{FF} &= 1 / \text{EE} \quad \text{then} \\
\text{GG} &= \text{HH}
\end{align*}
\]

**Example A.11.4f**

```fortran
SUBROUTINE A11_4(AA, BB, CC, DD, EE, FF, GG, HH, N)
INTEGER N
REAL AA(N,N), BB(N,N), CC(N,N)
REAL DD(N,N), EE(N,N), FF(N,N)
REAL GG(N,N), HH(N,N)

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

END SUBROUTINE A11_4
```

In the following example, an assignment to a shared scalar variable is performed by one thread in a workshare while all other threads in the team wait.

**Example A.11.5f**

```fortran
SUBROUTINE A11_5(AA, BB, CC, DD, N)
INTEGER N
REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
INTEGER SHR

!$OMP PARALLEL SHARED(SHR)
!$OMP WORKSHARE
AA = BB
SHR = 1
CC = DD * SHR
```

134 OpenMP API • Version 2.5 May 2005
Fortran (cont.)

The following example contains an assignment to a private scalar variable, which is performed by one thread in a workshare while all other threads wait. It is non-conforming because the private scalar variable is undefined after the assignment statement.

**Example A.11.6f**

```fortran
SUBROUTINE A11_6_WRONG(AA, BB, CC, DD, N)
INTEGER N
REAL AA(N,N), BB(N,N), CC(N,N), DD(N,N)
INTEGER PRI

!$OMP PARALLEL PRIVATE(PRI)
!$OMP WORKSHARE
AA = BB
PRI = 1
CC = DD * PRI
!$OMP END WORKSHARE
!$OMP END PARALLEL
END SUBROUTINE A11_6_WRONG
```

Fortran execution rules must be enforced inside a workshare construct. In the following example, 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:

**Example A.11.7f**

```fortran
SUBROUTINE A11_7(AA, BB, CC, N)
INTEGER N
REAL AA(N), BB(N), CC(N)

!$OMP PARALLEL
!$OMP WORKSHARE
AA(1:50) = BB(11:60)
CC(11:20) = AA(1:10)
```

Appendix A  Examples  135
A.12 The master Construct

The following example demonstrates the master construct (Section 2.7.1 on page 51). In
the example, the master keeps track of how many iterations have been executed and
prints out a progress report. The other threads skip the master region without waiting.

Example A.12.1c
#include <stdio.h>

extern float average(float, float, float);

void a12( float* x, float* xold, int n, float tol )
{
    int c, i, toobig;
    float error, y;
    c = 0;
    #pragma omp parallel
    {
        do{
            #pragma omp for private(i)
            for( i = 1; i < n-1; ++i ){
                xold[i] = x[i];
            }
            #pragma omp single
            {
                toobig = 0;
            }
            #pragma omp for private(i,y,error) reduction(+:toobig)
            for( i = 1; i < n-1; ++i ){
                y = x[i];
                x[i] = average( xold[i-1], x[i], xold[i+1] );
                error = y - x[i];
                if( error > tol || error < -tol ) ++toobig;
            }
            #pragma omp master
            {
                ++c;
                printf( "iteration %d, toobig=%d\n", c, toobig );
            }
        }
    }
}
Example A.12.1f

```fortran
SUBROUTINE A12( X, XOLD, N, TOL )
REAL X(*), XOLD(*), TOL
INTEGER N
INTEGER C, I, TOOBIG
REAL ERROR, Y, AVERAGE
EXTERNAL AVERAGE
C = 0
TOOBIG = 1
!$OMP PARALLEL
   DO WHILE( TOOBIG > 0 )
      !$OMP DO PRIVATE(I)
      DO I = 2, N-1
         XOLD(I) = X(I)
      ENDDO
      !$OMP SINGLE
      TOOBIG = 0
      !$OMP END SINGLE
      !$OMP DO PRIVATE(I,Y,ERROR), REDUCTION(+:TOOBIG)
      DO I = 2, N-1
         Y = X(I)
         X(I) = AVERAGE( XOLD(I-1), X(I), XOLD(I+1) )
         ERROR = Y-X(I)
         IF( ERROR > TOL .OR. ERROR < -TOL ) TOOBIG = TOOBIG+1
      ENDDO
      !$OMP MASTER
      C = C + 1
      PRINT *, 'Iteration ', C, ' TOOBIG=', TOOBIG
      !$OMP END MASTER
   ENDDO
!$OMP END PARALLEL
END SUBROUTINE A12
```
A.13 The critical Construct

The following example includes several critical constructs (Section 2.7.2 on page 52). 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 region. Because the two queues in this example are independent, they are protected by critical constructs with different names, xaxis and yaxis.

Example A.13.1c

```c/c++
int dequeue(float *a);
void work(int i, float *a);

void a13(float *x, float *y)
{
    int ix_next, iy_next;

    #pragma omp parallel shared(x, y) private(ix_next, iy_next)
    {
        #pragma omp critical (xaxis)
        ix_next = dequeue(x);
        work(ix_next, x);

        #pragma omp critical (yaxis)
        iy_next = dequeue(y);
        work(iy_next, y);
    }
}
```

Example A.13.1f

```fortran
SUBROUTINE A13(X, Y)
    REAL X(*), Y(*)
    INTEGER IX_NEXT, IY_NEXT

    !$OMP PARALLEL SHARED(X, Y) PRIVATE(IX_NEXT, IY_NEXT)

    !$OMP CRITICAL(XAXIS)
    CALL DEQUEUE(IX_NEXT, X)
    !$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)
```
A.14 Work-Sharing Constructs Inside a critical Construct

The following example demonstrates using a work-sharing construct inside a critical construct (see Section 2.7.2 on page 52). This example is conforming because the single region and the critical region are not closely nested (see Section 2.9 on page 87).

Example A.14.1c

```c
void a14()
{
    int i = 1;
    #pragma omp parallel sections
    {
        #pragma omp section
        {
            #pragma omp critical (name)
            {
                #pragma omp parallel
                {
                    #pragma omp single
                    {
                        i++;
                    }
                }
            }
        }
    }

    // C/C++
```

```fortran
!$OMP CRITICAL(YAXIS)
   CALL DEQUEUE(IY_NEXT,Y)
!$OMP END CRITICAL(YAXIS)
   CALL WORK(IY_NEXT, Y)
!$OMP END PARALLEL
END SUBROUTINE A13
```

```c
// C/C++
```
A.15 Binding of barrier Regions

The binding rules call for a barrier region to bind to the closest enclosing parallel region (see Section 2.7.3 on page 54).

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

The call from the main program to sub3 is conforming because the barrier region binds to the implicit inactive parallel region enclosing the sequential part. Also note that the barrier region in sub3 when called from sub2 only synchronizes the team of threads in the enclosing parallel region and not all the threads created in sub1.

Example A.15.1c

```c
void work(int n) {}

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

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

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

int main()
{
    sub1(2);
    sub2(2);
    sub3(2);
    return 0;
}
```

---

**Example A.15.1f**

```fortran
SUBROUTINE WORK(N)
    INTEGER N
END SUBROUTINE WORK

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

SUBROUTINE SUB2(K)
    INTEGER K
    !$OMP PARALLEL SHARED(K)
    CALL SUB3(K)
    !$OMP END PARALLEL
END SUBROUTINE SUB2
```
A.16  The atomic Construct

The following example avoids race conditions (simultaneous updates of an element of \( x \) by multiple threads) by using the `atomic` construct (Section 2.7.4 on page 55).

The advantage of using the `atomic` construct in this example is that it allows updates of two different elements of \( x \) to occur in parallel. If a `critical` construct (see Section 2.7.2 on page 52) were used instead, then all updates to elements of \( x \) would be executed serially (though not in any guaranteed order).

Note that the `atomic` directive applies only to the statement immediately following it. As a result, elements of \( y \) are not updated atomically in this example.

---

**Example A.16.1c**

```c
float work1(int i)
{
    return 1.0 * i;
}

float work2(int i)
{
    return 2.0 * i;
}
```

---

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

PROGRAM A15
 CALL SUB1(2)
 CALL SUB2(2)
 CALL SUB3(2)
END PROGRAM A15

void a16(float *x, float *y, int *index, int n)
{
    int i;

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

int main()
{
    float x[1000];
    float y[10000];
    int index[10000];
    int i;

    for (i = 0; i < 10000; i++) {
        index[i] = i % 1000;
        y[i]=0.0;
    }

    for (i = 0; i < 1000; i++)
        x[i] = 0.0;

    a16(x, y, index, 10000);
    return 0;
}

Example A.16.1f

REAL FUNCTION WORK1(I)
    INTEGER I
    WORK1 = 1.0 * I
    RETURN
END FUNCTION WORK1

REAL FUNCTION WORK2(I)
    INTEGER I
    WORK2 = 2.0 * I
    RETURN
END FUNCTION WORK2

SUBROUTINE SUBA16(X, Y, INDEX, N)
    REAL X(*), Y(*)
    }
### A.17 Restrictions on the atomic Construct

The following examples illustrate the restrictions on the atomic construct. For more information, see Section 2.7.4 on page 55.

**C/C++**

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 a compatible type.
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.

The following are some non-conforming examples:

**Example A.17.1c**

```c
void a17_1_wrong ()
{
    union {int n; float x;} u;

#pragma omp parallel
{
    #pragma omp atomic
    u.n++;

    #pragma omp atomic
    u.x += 1.0;

    /* Incorrect because the atomic constructs reference the same location
    through incompatible types */
}
```

**Example A.17.1f**

```fortran
SUBROUTINE A17_1_WRONG()
    INTEGER:: I
    REAL:: R
    EQUIVALENCE(I,R)

!$OMP PARALLEL
!$OMP ATOMIC
    I = I + 1
!$OMP ATOMIC
    R = R + 1.0
! incorrect because I and R reference the same location
! but have different types
!$OMP END PARALLEL
```
END SUBROUTINE A17_1_WRONG

---

Example A.17.2c

```c
void a17_2_wrong ()
{
    int  x;
    int *i;
    float  *r;

    i = &x;
    r = (float *)&x;

    #pragma omp parallel
    {
        #pragma omp atomic
        *i += 1;

        #pragma omp atomic
        *r += 1.0;

        /* Incorrect because the atomic constructs reference the same location
           through incompatible types */
    }
}
```

---

The following example is non-conforming because `I` and `R` reference the same location but have different types.

Example A.17.2f

```fortran
SUBROUTINE SUB()
    COMMON /BLK/ R
    REAL R

    !$OMP ATOMIC
    R = R + 1.0
END SUBROUTINE SUB

SUBROUTINE A17_2_WRONG()
    COMMON /BLK/ I
    INTEGER I
```
Although the following example might work on some implementations, this is also non-conforming:

Example A.17.3f

```fortran
SUBROUTINE A17_3_WRONG
    INTEGER:: I
    REAL:: R
    EQUIVALENCE(I,R)
    !$OMP PARALLEL
    !$OMP ATOMIC
        I = I + 1
        ! incorrect because I and R reference the same location
        ! but have different types
        !$OMP END PARALLEL
    !$OMP PARALLEL
    !$OMP ATOMIC
        R = R + 1.0
        ! incorrect because I and R reference the same location
        ! but have different types
        !$OMP END PARALLEL
END SUBROUTINE A17_3_WRONG
```

A.18 The flush Construct with a List

The following example uses the flush construct (see Section 2.7.5 on page 58) for point-to-point synchronization of specific objects between pairs of threads:

Example A.18.1c

```
#include <omp.h>
```
#define NUMBER_OF_THREADS 256

int synch[NUMBER_OF_THREADS];
float work[NUMBER_OF_THREADS];
float result[NUMBER_OF_THREADS];

float fn1(int i)
{
    return i*2.0;
}

float fn2(float a, float b)
{
    return a + b;
}

int main()
{
    int iam, neighbor;

    #pragma omp parallel private(iam, neighbor) shared(work, synch)
    {
        iam = omp_get_thread_num();
        synch[iam] = 0;

        #pragma omp barrier
        /*Do computation into my portion of work array */
        work[iam] = fn1(iam);

        /* Announce that I am done with my work. The first flush
         * ensures that my work is made visible before synch.
         * The second flush ensures that synch is made visible.
         */

        #pragma omp flush(work, synch)
        synch[iam] = 1;
        #pragma omp flush(synch)

        /* Wait for neighbor. The first flush ensures that synch is read
         * from memory, rather than from the temporary view of memory.
         * The second flush ensures that work is read from memory, and
         * is done so after the while loop exits.
         */

        neighbor = (iam>0 ? iam : omp_get_num_threads()) - 1;
        while (synch[neighbor] == 0) {
            #pragma omp flush(synch)
Example A.18.1f

```c
#include "omp_lib.h"    // or USE OMP_LIB

int isync[256];
real work[256];
real result[256];
int iam, neighbor;
!
int omp parallel private(iam, neighbor) shared(work, isync)
!
iam = omp_get_thread_num() + 1
isync(iam) = 0
!
omp barrier
!
do computation into my portion of work array
!
work(iam) = fn1(iam)
!
do computation into my portion of work array
!
result[iam] = fn2(work[neighbor], work[iam]);
!
omp barrier
!
/* output result here */
!
return 0;
```
The first flush ensures that my work is made visible before
synch. The second flush ensures that synch is made visible.

```fortran
!$OMP FLUSH(WORK, ISYNC)
ISYNC(IAM) = 1
!$OMP FLUSH(ISYNC)

C Wait until neighbor is done. The first flush ensures that
C synch is read from memory, rather than from the temporary
C view of memory. The second flush ensures that work is read
C from memory, and is done so after the while loop exits.

IF (IAM .EQ. 1) THEN
  NEIGHBOR = OMP_GET_NUM_THREADS()
ELSE
  NEIGHBOR = IAM - 1
ENDIF

DO WHILE (ISYNC(NEIGHBOR) .EQ. 0)
  !$OMP FLUSH(ISYNC)
END DO

!$OMP FLUSH(WORK, ISYNC)
RESULT(IAM) = FN2(WORK(NEIGHBOR), WORK(IAM))
!$OMP END PARALLEL
```

```fortran
END PROGRAM A18
```

---

A.19 The flush Construct without a List

The following example (for Section 2.7.5 on page 58) distinguishes the shared objects
affected by a `flush` construct with no list from the shared objects that are not affected:

```
Example A.19.1c
int x, *p = &x;

void f1(int *q)
{
  *q = 1;
  //pragma omp flush
  /* x, p, and *q are flushed */
  /* because they are shared and accessible */
  /* q is not flushed because it is not shared. */
```
void f2(int *q)
{
    #pragma omp barrier
    *q = 2;
    #pragma omp barrier

    /* a barrier implies a flush */
    /* x, p, and *q are flushed */
    /* because they are shared and accessible */
    /* q is not flushed because it is not shared. */
}

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

        f2(&j);
        /* i, n, and sum were not flushed */
        /* because they were not accessible in f2 */
        /* j was flushed because it was accessible */
        sum += i + j + *p + n;
    }

    return sum;
}

int main()
{
    int result = g(7);
    return result;
}

--- C/C++

--- Fortran

Example A.19.1f

SUBROUTINE F1(Q)
COMMON /DATA/ X, P
INTEGER, TARGET :: X
INTEGER, POINTER :: P
INTEGER Q

Q = 1
!$OMP FLUSH
! X, P and Q are flushed
! because they are shared and accessible
END SUBROUTINE F1

SUBROUTINE F2(Q)
COMMON /DATA/ X, P
INTEGER, TARGET :: X
INTEGER, POINTER :: P
INTEGER Q

!$OMP BARRIER
Q = 2
!$OMP BARRIER
! a barrier implies a flush
! X, P and Q are flushed
! because they are shared and accessible
END SUBROUTINE F2

INTEGER FUNCTION G(N)
COMMON /DATA/ X, P
INTEGER, TARGET :: X
INTEGER, POINTER :: P
INTEGER N
INTEGER I, J, SUM

I = 1
SUM = 0
P = 1
!$OMP PARALLEL REDUCTION(+: SUM) NUM_THREADS(10)
CALL F1(J)
! I, N and SUM were not flushed
! because they were not accessible in F1
! J was flushed because it was accessible
SUM = SUM + J

CALL F2(J)
! I, N, and SUM were not flushed
! because they were not accessible in F2
! J was flushed because it was accessible
SUM = SUM + I + J + P + N
A.20 Placement of \texttt{flush} and \texttt{barrier} Directives

The following example is non-conforming, because the \texttt{flush} and \texttt{barrier} directives cannot be the immediate substatement of an \texttt{if} statement. See Section 2.7.3 on page 54 and Section 2.7.5 on page 58.

\textit{Example A.20.1c}

```c
void a20_wrong()
{
    int a = 1;

    #pragma omp parallel
    {
        if (a != 0)
            #pragma omp flush(a)
            /* incorrect as flush cannot be immediate substatement
            of if statement */

        if (a != 0)
            #pragma omp barrier
            /* incorrect as barrier cannot be immediate substatement
            of if statement */
```
The following version of the above example is conforming because the \texttt{flush} and \texttt{barrier} directives are enclosed in a compound statement.

\textit{Example A.20.2c}

\begin{verbatim}
void a20()
{
  int a = 1;

  #pragma omp parallel
  {
    if (a != 0) {
      #pragma omp flush(a)
    }
    if (a != 0) {
      #pragma omp barrier
    }
  }
}
\end{verbatim}

\section*{A.21 The \texttt{ordered} Clause and the \texttt{ordered} Construct}

Ordered constructs (Section 2.7.6 on page 61) are useful for sequentially ordering the output from work that is done in parallel. The following program prints out the indices in sequential order:

\begin{verbatim}
#include <stdio.h>

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

void a21(int lb, int ub, int stride)
{


}\end{verbatim}
```
int i;

#pragma omp parallel for ordered schedule(dynamic)
for (i=lb; i<ub; i+=stride)
    work(i);
}

int main()
{
    a21(0, 100, 5);
    return 0;
}

```C/C++
```
Example A.21.1f

```Fortran
SUBROUTINE WORK(K)
    INTEGER k

    !$OMP ORDERED
    WRITE(*,*) K
    !$OMP END ORDERED

    END SUBROUTINE WORK

SUBROUTINE SUBA21(LB, UB, STRIDE)
    INTEGER LB, UB, STRIDE
    INTEGER I

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

    END SUBROUTINE SUBA21

PROGRAM A21
    CALL SUBA21(1,100,5)
    END PROGRAM A21
```

Fortran

It is possible to have multiple `ordered` constructs within a loop region with the `ordered` clause specified. The first example is non-conforming because all iterations execute two `ordered` regions. An iteration of a loop must not execute more than one `ordered` region:
Example A.21.2c

```c
void work(int i) {}

void a21_wrong(int n)
{
    int i;
    #pragma omp for ordered
    for (i=0; i<n; i++) {
        /* incorrect because an iteration may not execute more than one
           ordered region */
        #pragma omp ordered
        work(i);
        #pragma omp ordered
        work(i+1);
    }
}
```

Example A.21.2f

```fortran
SUBROUTINE WORK(I)
   INTEGER I
END SUBROUTINE WORK

SUBROUTINE A21_WRONG(N)
   INTEGER N

   INTEGER I

   !$OMP   DO ORDERED
   DO I = 1, N
       !$omp ordered
       CALL WORK(I)
   END ORDERED

   !$OMP   DO ORDERED
   CALL WORK(I+1)
   END ORDERED
END DO

END SUBROUTINE A21_WRONG
```

The following is a conforming example with more than one `ordered` construct. Each iteration will execute only one `ordered` region:
Example A.21.3c

```c
void a21_good(int n)
{
    int i;

    //pragma omp for ordered
    for (i=0; i<n; i++) {
        if (i <= 10) {
            //pragma omp ordered
            work(i);
        }
        if (i > 10) {
            //pragma omp ordered
            work(i+1);
        }
    }
}
```

Example A.21.3f

```fortran
SUBROUTINE A21_GOOD(N)
    INTEGER N

    !$OMP DO ORDERED
    DO I = 1,N
        IF (I <= 10) THEN
            !$OMP ORDERED
            CALL WORK(I)
            !$OMP END ORDERED
        ENDIF
    IF (I > 10) THEN
        !$OMP ORDERED
        CALL WORK(I+1)
        !$OMP END ORDERED
    ENDDO
    END SUBROUTINE A21_GOOD
```


A.22 **The threadprivate Directive**

The following examples demonstrate how to use the `threadprivate` directive (Section 2.8.2 on page 66) to give each thread a separate counter.

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

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

```fortran
INTEGER FUNCTION INCREMENT_COUNTER()
COMMON/A22_COMMON/COUNTER
!$OMP THREADPRIVATE(/A22_COMMON/)
COUNTER = COUNTER +1
INCREMENT_COUNTER = COUNTER
RETURN
END FUNCTION INCREMENT_COUNTER
```

The following example uses `threadprivate` on a static variable:

```c++
int increment_counter_2()
{
    static int counter = 0;
    #pragma omp threadprivate(counter)
    counter++;  
    return(counter);
}
```
The following example illustrates how modifying a variable that appears in an initializer can cause unspecified behavior, and also how to avoid this problem by using an auxiliary object and a copy-constructor.

**Example A.22.3c**

```cpp
class T {
    public:
        int val;
        T (int);
        T (const T&);
}

T :: T (int v){
    val = v;
}

T :: T (const T& t) {
    val = t.val;
}

void g(T a, T b){
    a.val += b.val;
}

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

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

C/C++
The following examples show non-conforming uses and correct uses of the `threadprivate` directive. For more information, see Section 2.8.2 on page 66 and Section 2.8.4.1 on page 84.

The following example is non-conforming because the common block is not declared local to the subroutine that refers to it:

```
Example A.22.4f

```Fortran

```
MODULE A22_MODULE
  COMMON /T/ A
END MODULE A22_MODULE

SUBROUTINE A22_4_WRONG()
  USE A22_MODULE
  !$OMP THREADPRIVATE(/T/)
  !non-conforming because /T/ not declared in A22_4_WRONG
END SUBROUTINE A22_4_WRONG
```

The following example is also non-conforming because the common block is not declared local to the subroutine that refers to it:

```
Example A.22.5f

```Fortran

```
SUBROUTINE A22_5_WRONG()
  COMMON /T/ A
  !$OMP THREADPRIVATE(/T/)
  CONTAINS
    SUBROUTINE A22_5S_WRONG()
      !$OMP PARALLEL COPYIN(/T/)
      !non-conforming because /T/ not declared in A22_5S_WRONG
    !$OMP END PARALLEL
    END SUBROUTINE A22_5S_WRONG
END SUBROUTINE A22_5_WRONG
```

The following example is a correct rewrite of the previous example:

```
Example A.22.6f

```Fortran

```
SUBROUTINE A22_6_GOOD()
  COMMON /T/ A
  !$OMP THREADPRIVATE(/T/)
```

```
The following is an example of the use of `threadprivate` for local variables:

```
Example A.22.7f
```

```
PROGRAM A22_7_GOOD
  INTEGER, ALLOCATABLE, SAVE :: A(:)
  INTEGER, POINTER, SAVE :: PTR
  INTEGER, SAVE :: I
  INTEGER, TARGET :: TARG
  LOGICAL :: FIRSTIN = .TRUE.
  !$OMP THREADPRIVATE(A, 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
```

```
The above program, if executed by two threads, will print one of the following two sets of output:

```
7     a = 11 12 13
8     ptr = 4
9     i = 15
10    A is not allocated
11    ptr = 4
12    i = 5
13    or
14    A is not allocated
15    ptr = 4
16    i = 5
17     a = 1 2 3
18     ptr = 4
19     i = 5
```

The following is an example of the use of `threadprivate` for module variables:

```
Example A.22.8f

22     MODULE A22_MODULE8
23     REAL, POINTER :: WORK(:)
24     SAVE WORK
25     !$OMP THREADPRIVATE(WORK)
26     END MODULE A22_MODULE8
27     SUBROUTINE SUB1(N)
28     USE A22_MODULE8
29     !$OMP PARALLEL PRIVATE(THE_SUM)
30        ALLOCATE(WORK(N))
31        CALL SUB2(THE_SUM)
32        WRITE(*,*), THE_SUM
33     !$OMP END PARALLEL
34     END SUBROUTINE SUB1
```
A.23 Fortran Restrictions on shared and private Clauses with Common Blocks

When a named common block is specified in a private, firstprivate, or lastprivate clause of a construct, none of its members may be declared in another data-sharing attribute clause on that construct. The following examples illustrate this point. For more information, see Section 2.8.3 on page 70.

The following example is conforming:

**Example A.23.1f**

```fortran
SUBROUTINE A23_1_GOOD()
    COMMON /C/ X, Y
    REAL X, Y

    !$OMP PARALLEL PRIVATE (/C/)
    ! do work here
    !$OMP END PARALLEL

    !$OMP PARALLEL SHARED (X, Y)
    ! do work here
    !$OMP END PARALLEL

    END SUBROUTINE A23_1_GOOD
```

The following example is also conforming:
Example A.23.2f

```fortran
SUBROUTINE A23_2_GOOD()
    COMMON /C/ X, Y
    REAL X, Y

    INTEGER I

    !$OMP PARALLEL
    !$OMP DO PRIVATE(/C/)
    DO I=1,1000
        ! do work here
        ENDDO
    !$OMP END DO
    !$OMP DO PRIVATE(X)
    DO I=1,1000
        ! do work here
        ENDDO
    !$OMP END DO
    !$OMP END PARALLEL
END SUBROUTINE A23_2_GOOD
```

The following example is conforming:

Example A.23.3f

```fortran
SUBROUTINE A23_3_GOOD()
    COMMON /C/ X, Y

    !$OMP PARALLEL PRIVATE (/C/)
    ! do work here
    !$OMP END PARALLEL
    !$OMP PARALLEL SHARED (/C/)
    ! do work here
    !$OMP END PARALLEL
END SUBROUTINE A23_3_GOOD
```

The following example is non-conforming because \( x \) is a constituent element of \( c \):

Example A.23.4f

```fortran
SUBROUTINE A23_4_WRONG()
```
The following example is non-conforming because a common block may not be declared both shared and private:

**Example A.23.5f**

```fortran
SUBROUTINE A23_5_WRONG()
  COMMON /C/ X,Y
  ! Incorrect: common block C cannot be declared both
  ! shared and private
  !$OMP PARALLEL PRIVATE (/C/), SHARED(/C/)
  ! do work here
  !$OMP END PARALLEL
END SUBROUTINE A23_5_WRONG
```

---

**A.24 The default (none) Clause**

The following example distinguishes the variables that are affected by the default(none) clause from those that are not. For more information on the default clause, see Section 2.8.3.1 on page 71.

**C/C++**

```c
#include <omp.h>
int x, y, z[1000];
#pragma omp threadprivate(x)
void a24(int a) {
  const int c = 1;
  int i = 0;

  #pragma omp parallel default(none) private(a) shared(z)
  {
    int j = omp_get_num_threads();
    /* O.K. - j is declared within parallel region */
```
a = z[j]; /* O.K. - a is listed in private clause */
/*      - z is listed in shared clause */
x = c;  /* O.K. - x is threadprivate */
/*      - c has const-qualified type */
z[i] = y; /* Error - cannot reference i or y here */

#pragma omp for firstprivate(y)
for (i=0; i<10 ; i++) {
    z[i] = y; /* O.K. - i is the loop iteration variable */
    /* - y is listed in firstprivate clause */
}

z[i] = y; /* Error - cannot reference i or y here */

Example A.24.1f

SUBROUTINE A24(A)
    INCLUDE "omp_lib.h" ! or USE OMP_LIB

    INTEGER A

    INTEGER X, Y, Z(1000)
    COMMON/BLOCKX/X
    COMMON/BLOCKY/Y
    COMMON/BLOCKZ/Z

    !$OMP THREADPRIVATE(/BLOCKX/)

    INTEGER I, J
    i = 1

    !$OMP PARALLEL default(none) private(A) shared(Z) private(J)
        J = omp_get_num_threads();
        ! O.K. - J is listed in PRIVATE clause
        A = Z(J) ! O.K. - A is listed in PRIVATE clause
        ! - Z is listed in SHARED clause
        X = 1 ! O.K. - X is THREADPRIVATE
        Z(I) = Y ! Error - cannot reference I or Y here
    !$OMP

    do firstprivate(y)
        do I = 1,10
            Z(I) = Y ! O.K. - I is the loop iteration variable
            ! Y is listed in FIRSTPRIVATE clause
        end do
    end do
A.25 Race Conditions Caused by Implied Copies of Shared Variables in Fortran

The following example contains a race condition, because the shared variable, which is an array section, is passed as an actual argument to a routine that has an assumed-size array as its dummy argument (see Section 2.8.3.2 on page 72). The subroutine call passing an array section argument may cause the compiler to copy the argument into a temporary location prior to the call and copy from the temporary location into the original variable when the subroutine returns. This copying would cause races in the parallel region.

Example A.25.1f

SUBROUTINE A25

    INCLUDE "omp_lib.h" ! or USE OMP_LIB

    REAL A(20)
    INTEGER MYTHREAD

    !$OMP PARALLEL SHARED(A) PRIVATE(MYTHREAD)

    MYTHREAD = OMP_GET_THREAD_NUM()
    IF (MYTHREAD .EQ. 0) THEN
        CALL SUB(A(1:10)) ! compiler may introduce writes to A(6:10)
    ELSE
        A(6:10) = 12
    ENDIF

    !$OMP END PARALLEL

END SUBROUTINE A25

SUBROUTINE SUB(X)

    REAL X(*)
    X(1:5) = 4

END SUBROUTINE SUB
A.26 The private Clause

In the following example, the values of \( i \) and \( j \) are undefined on exit from the parallel region. For more information on the private clause, see Section 2.8.3.3 on page 73.

C/C++ Example A.26.1c

```c
#include <stdio.h>

int main()
{
    int i, j;
    i = 1;
    j = 2;
    #pragma omp parallel private(i) firstprivate(j)
    {
        i = 3;
        j = j + 2;
    }
    printf("%d %d\n", i, j); /* i and j are undefined */
    return 0;
}
```

Fortran Example A.26.1f

```fortran
PROGRAM A26
    INTEGER I, J
    I = 1
    J = 2
    !$OMP PARALLEL PRIVATE(I) FIRSTPRIVATE(J)
    I = 3
    J = J + 2
    !$OMP END PARALLEL
```

END SUBROUTINE SUB
The `private` clause of a `parallel` construct is only in effect inside the construct, and not for the rest of the region. Therefore, in the example that follows, any uses of the variable `a` within the loop in the routine `f` refers to a private copy of `a`, while a usage in routine `g` refers to the global `a`.

### C/C++ Example A.26.2c

```c
int a;

void g(int k) {
  a = k;   /* The global "a", not the private "a" in f */
}

void f(int n) {
  int a = 0;
  #pragma omp parallel for private(a)
  for (int i=1; i<n; i++) {
    a = i;
    g(a*2);   /* Private copy of "a" */
  }
}
```

### Fortran Example A.26.2f

```fortran
MODULE A26_2
REAL A

CONTAINS

  SUBROUTINE G(K)
    REAL K
    A = K  ! This is A in module A26_2, not the private A in F
  END SUBROUTINE G

  SUBROUTINE F(N)
    INTEGER N
    REAL A
```
Reprivatization

The following example demonstrates the reprivatization of variables (see Section 2.8.3.3 on page 73). Private variables can be marked `private` again in a nested construct. They do not have to be shared in the enclosing `parallel` region.

Example A.27.1c

```c
void a27()
{
    int i, a;

    #pragma omp parallel private(a)
    {
        #pragma omp parallel for private(a)
        for (i=0; i<10; i++)
        {
            /* do work here */
        }
    }
}
```

Example A.27.1f

```fortran
SUBROUTINE A27()
    INTEGER I, A

    !$OMP PARALLEL PRIVATE(A)
    !$OMP PARALLEL DO PRIVATE(A)
    DO I = 1, 10
        ! do work here
    ENDDO
END SUBROUTINE A27
```

```fortran
!$OMP PARALLEL DO PRIVATE(A)
    CALL G(A*2)
ENDDO
!$OMP END PARALLEL DO
END SUBROUTINE F
```

END MODULE A26_2
A.28 Fortran Restrictions on Storage Association with the private Clause

The following non-conforming examples illustrate the implications of the private clause rules with regard to storage association (see Section 2.8.3.3 on page 73).

Example A.28.1f

```fortran
SUBROUTINE SUB()
  CHARACTER /BLOCK/ X
  PRINT *, X  ! X is undefined
END SUBROUTINE SUB

PROGRAM A28_1
  CHARACTER /BLOCK/ X
  X = 1.0
  !$OMP PARALLEL PRIVATE (X)
  X = 2.0
  CALL SUB()
  !$OMP END PARALLEL
END PROGRAM A28_1
```

Example A.28.2f

```fortran
PROGRAM A28_2
  CHARACTER /BLOCK2/ X
  X = 1.0
  !$OMP PARALLEL PRIVATE (X)
  X = 2.0
  CALL SUB()
  !$OMP END PARALLEL
END CONTAINS

SUBROUTINE SUB()
```

END SUBROUTINE A27
Fortran (cont.)

COMMON /BLOCK2/ Y

PRINT *,X               ! X is undefined
PRINT *,Y               ! Y is undefined
END SUBROUTINE SUB

END PROGRAM A28_2

Example A.28.3f

PROGRAM A28_3
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

END PROGRAM A28_3

Example A.28.4f

PROGRAM A28_4
INTEGER I, J
INTEGER A(100), B(100)
EQUIVALENCE (A(51), B(1))

 !$OMP PARALLEL DO DEFAULT(PRIVATE) PRIVATE(I,J) LASTPRIVATE(A)
 DO I=1,100
 DO J=1,100
 B(J) = J - 1
 ENDDO
 DO J=1,100
 A(J) = J               ! B becomes undefined at this point
 ENDDO
 DO J=1,50
 B(J) = B(J) + 1        ! B is undefined
               ! A becomes undefined at this point
 ENDDO
 !$OMP END PARALLEL DO    ! The LASTPRIVATE write for A has
               ! undefined results

172 OpenMP API • Version 2.5 May 2005
Example A.28.4f

```fortran
PRINT *, B ! B is undefined since the LASTPRIVATE
! write of A was not defined
END PROGRAM A28_4
```

Example A.28.5f

```fortran
SUBROUTINE SUB1(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 SUBROUTINE SUB1

PROGRAM A28_5
  COMMON /BLOCK5/ A
  DIMENSION B(10)
  EQUIVALENCE (A,B(1))
  ! the common block has to be at least 10 words
  A = 0

  !$OMP PARALLEL PRIVATE(/BLOCK5/)

  ! 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 SUB1(A)
  !$OMP MASTER
  PRINT *, A
  !$OMP END MASTER

  !$OMP END PARALLEL
END PROGRAM A28_5
```
The following example illustrates the size and value of list items of array or pointer type in a firstprivate clause (Section 2.8.3.4 on page 75). The size of new list items is based on the type of the corresponding original list item, as determined by the base language.

In this example:

- The type of \texttt{A} is array of two arrays of two ints.
- The type of \texttt{B} is adjusted to pointer to array of \texttt{n} ints, because it is a function parameter.
- The type of \texttt{C} is adjusted to pointer to int, because it is a function parameter.
- The type of \texttt{D} is array of two arrays of two ints.
- The type of \texttt{E} is array of \texttt{n} arrays of \texttt{n} ints.

Note that \texttt{B} and \texttt{E} involve variable length array types.

The new items of array type are initialized as if each integer element of the original array is assigned to the corresponding element of the new array. Those of pointer type are initialized as if by assignment from the original item to the new item.

\textit{Example A.29.1c}

\begin{verbatim}
#include <assert.h>

int A[2][2] = {1, 2, 3, 4};

void f(int n, int B[n][n], int C[])
{
    int D[2][2] = {1, 2, 3, 4};
    int E[n][n];

    assert(n >= 2);
    E[1][1] = 4;

    #pragma omp parallel firstprivate(B, C, D, E)
    {
        assert(sizeof(B) == sizeof(int (*)[n]));
        assert(sizeof(C) == sizeof(int*));
        assert(sizeof(D) == 4 * sizeof(int));
        assert(sizeof(E) == n * n * sizeof(int));
    }
\end{verbatim}
A.30 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 (Section 2.8.3.5 on page 77) so that the values of the variables are the same as when the loop is executed sequentially.

```c
void a30 (int n, float *a, float *b)
{
    int i;

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

        a[i]=b[i];      /* i == n-1 here */
    }
}
```

```fortran
SUBROUTINE A30 (N, A, B)

    INTEGER N
    REAL A(*), B(*)

```
INTEGER I

!$OMP PARALLEL
!$OMP DO LASTPRIVATE(I)

DO I=1,N-1
   A(I) = B(I) + B(I+1)
ENDDO

!$OMP END PARALLEL

A(I) = B(I) ! I has the value of N here

END SUBROUTINE A30

---

A.31 The reduction Clause

The following example demonstrates the reduction clause (Section 2.8.3.6 on page 79):

\begin{verbatim}
void a31_1(float *x, int *y, int n)
{
   int i, b;
   float a;
   a = 0.0;
   b = 0;

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

---

176 OpenMP API • Version 2.5 May 2005
Example A.31.1f

```fortran
SUBROUTINE A31_1(A, B, X, Y, N)

    INTEGER N
    REAL X(*), Y(*), A, B

    !$OMP PARALLEL DO PRIVATE(I) SHARED(X, N) REDUCTION(+:A)
    !$OMP& REDUCTION(MIN:B)

    DO I=1,N
        A = A + X(I)
        B = MIN(B, Y(I))
    END DO

END SUBROUTINE A31_1
```

A common implementation of the preceding example is to treat it as if it had been written as follows:

Example A.31.2c

```c
void a31_2(float *x, int *y, int n)
{
    int i, b, b_p;
    float a, a_p;

    a = 0.0;
    b = 0;

    #pragma omp parallel shared(a, b, x, y, n) \
       private(a_p, b_p)
    {
        a_p = 0.0;
        b_p = 0;

        #pragma omp for private(i)
        for (i=0; i<n; i++) {
```
a_p += x[i];

b_p ^= y[i];

#pragma omp critical
{
  a += a_p;
  b ^= b_p;
}

Example A.31.2

SUBROUTINE A31_2 (A, B, X, Y, N)

  INTEGER N
  REAL X(*), Y(*), A, B, A_P, B_P

  !$OMP PARALLEL SHARED(X, Y, N, A, B) PRIVATE(A_P, B_P)

  A_P = 0.0
  B_P = HUGE(B_P)

  !$OMP DO PRIVATE(I)
  DO I=1,N
    A_P = A_P + X(I)
    B_P = MIN(B_P, Y(I))
  ENDDO
  !$OMP END DO

  !$OMP CRITICAL
  A = A + A_P
  B = MIN(B, B_P)
  !$OMP END CRITICAL

  !$OMP END PARALLEL

END SUBROUTINE A31_2

The following program is non-conforming because the reduction is on the intrinsic procedure name MAX but that name has been redefined to be the variable named MAX.
Example A.31.3f

```fortran
PROGRAM A31_3_WRONG
MAX = HUGE(0)
M = 0

 !$OMP PARALLEL DO REDUCTION(MAX: M) ! MAX is no longer the
  ! intrinsic so this
  ! is non-conforming
DO I = 1, 100
   CALL SUB(M, I)
END DO
END PROGRAM A31_3_WRONG

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

The following conforming program performs the reduction using the intrinsic procedure name \texttt{MAX} even though the intrinsic \texttt{MAX} has been renamed to \texttt{REN}.

Example A.31.4f

```fortran
MODULE M
   INTRINSIC MAX
END MODULE M

PROGRAM A31_4
   USE M, REN => MAX
   N = 0
   !$OMP PARALLEL DO REDUCTION(REN: N) ! still does MAX
   DO I = 1, 100
      N = MAX(N, I)
   END DO
END PROGRAM A31_4
```

The following conforming program performs the reduction using intrinsic procedure name \texttt{MAX} even though the intrinsic \texttt{MAX} has been renamed to \texttt{MIN}.

Example A.31.5f

```fortran
MODULE MOD
```

Appendix A   Examples   179
A.32 The copyin Clause

The copyin clause (see Section 2.8.4.1 on page 84) is used to initialize threadprivate data upon entry to a parallel region. The value of the threadprivate variable in the master thread is copied to the threadprivate variable of each other team member.

```c
#include <stdlib.h>

float* work;
int size;
float tol;

#pragma omp threadprivate(work, size, tol)

void a32(float t, int n)
{
    tol = t;
    size = n;
    #pragma omp parallel copyin(tol, size)
    {
        build();
    }
}

void build()
{
    int i;
```
work = (float*)malloc( sizeof(float)*size );
for( i = 0; i < size; ++i ) work[i] = tol;

Example A.32.1f

MODULE M
  REAL, POINTER, SAVE :: WORK(:)
  INTEGER :: SIZE
  REAL :: TOL
  !$OMP THREADPRIVATE(WORK,SIZE,TOL)
END MODULE M

SUBROUTINE A32( T, N )
  USE M
  REAL :: T
  INTEGER :: N
  TOL = T
  SIZE = N
  !$OMP PARALLEL COPYIN(TOL,SIZE)
  CALL BUILD
  !$OMP END PARALLEL
END SUBROUTINE A32

SUBROUTINE BUILD
  USE M
  ALLOCATE(WORK(SIZE))
  WORK = TOL
END SUBROUTINE BUILD

A.33 The copyprivate Clause

The copyprivate clause (see Section 2.8.4.2 on page 85) can be used to broadcast values acquired by a single thread directly to all instances of the private variables in the other threads. In this example, if the routine is called from the sequential part, 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 input routine has been executed by one thread, no thread leaves the construct until the private objects designated by a, b, x, and y in all threads have become defined with the values read.
In contrast to the previous example, suppose the input 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.

```c++
#include <stdio.h>
#include <stdlib.h>

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

    #pragma omp single copyprivate(tmp)
    {
        tmp = (float *) malloc(sizeof(float));
    } /* copies the pointer only */
```
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.
Example A.33.3c

```c
#include <stdio.h>
#include <stdlib.h>
#include <omp.h>

omp_lock_t *new_lock()
{
  omp_lock_t *lock_ptr;

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

  return lock_ptr;
}
```

Example A.33.3f

```fortran
FUNCTION NEW_LOCK()
  USE OMP_LIB       ! or INCLUDE "omp_lib.h"
  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
```

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.

Example A.33.4f

```fortran
SUBROUTINE S(N)
  INTEGER N

  REAL, DIMENSION(:), ALLOCATABLE :: A
  REAL, DIMENSION(:), POINTER :: B

  ALLOCATE(A(N))
  !$OMP SINGLE
  ALLOCATE(B(N))
```
A.34 Nested Loop Constructs

The following example of loop construct nesting (see Section 2.9 on page 87) is conforming because the inner and outer loop regions bind to different parallel regions:

C/C++

```c
void good_nesting(int n) {
    int i, j;
    #pragma omp parallel default(shared)
    {
        #pragma omp for
        for (i=0; i<n; i++) {
            #pragma omp parallel shared(i, n)
            {
                #pragma omp for
                for (j=0; j < n; j++)
                    work(i, j);
            }
        }
    }
}
```

Fortran

```fortran
SUBROUTINE WORK(I, J)
```

C/C++
INTEGER I, J
END SUBROUTINE WORK

SUBROUTINE GOOD_NESTING(N)
INTEGER N

INTEGER I
!$OMP PARALLEL DEFAULT(SHARED)
!$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
END SUBROUTINE GOOD_NESTING

The following variation of the preceding example is also conforming:

Example A.34.2c

void work(int i, int j) {}

void work1(int i, int n)
{
   int j;
   #pragma omp parallel default(shared)
   {
      #pragma omp for
      for (j=0; j<n; j++)
         work(i, j);
   }
}

void good_nesting2(int n)
{
   int i;
   #pragma omp parallel default(shared)
   {
      #pragma omp for
      for (i=0; i<n; i++)
         work1(i, n);
### Appendix A Examples

#### C/C++

```c
Example A.34.2f

```SUBROUTINE WORK(I, J)
INTEGER I, J
END SUBROUTINE WORK

SUBROUTINE WORK1(I, N)
INTEGER J
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
DO J = 1, N
   CALL WORK(I, J)
END DO
!$OMP END PARALLEL
END SUBROUTINE WORK1

SUBROUTINE GOOD_NESTING2(N)
INTEGER N
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
DO I = 1, N
   CALL WORK1(I, N)
END DO
!$OMP END PARALLEL
END SUBROUTINE GOOD_NESTING2
```

#### Fortran

```fortran
Example A.34.2f

SUBROUTINE WORK(I, J)
   INTEGER I, J
END SUBROUTINE WORK

SUBROUTINE WORK1(I, N)
   INTEGER J
   !$OMP PARALLEL DEFAULT(SHARED)
   !$OMP DO
   DO J = 1, N
      CALL WORK(I, J)
   END DO
   !$OMP END PARALLEL
END SUBROUTINE WORK1

SUBROUTINE GOOD_NESTING2(N)
   INTEGER N
   !$OMP PARALLEL DEFAULT(SHARED)
   !$OMP DO
   DO I = 1, N
      CALL WORK1(I, N)
   END DO
   !$OMP END PARALLEL
END SUBROUTINE GOOD_NESTING2
```

#### A.35 Restrictions on Nesting of Regions

The examples in this section illustrate the region nesting rules. For more information on region nesting, see Section 2.9 on page 87.

The following example is non-conforming because the inner and outer loop regions are closely nested:

```c
Example A.35.1c

void work(int i, int j) {}

void wrong1(int n)
```

Appendix A  Examples  187
```c
{  
#pragma omp parallel default(shared)  
{  
    int i, j;  
#pragma omp for  
    for (i=0; i<n; i++) {  
      /* incorrect nesting of loop regions */  
#pragma omp for  
      for (j=0; j<n; j++)  
      work(i, j);  
    }  
}

C/C++

```
void wrong2(int n)
{
    #pragma omp parallel default(shared)
    {
        int i;
        #pragma omp for
        for (i=0; i<n; i++)
            work1(i, n);
    }
}

SUBROUTINE WORK1(I,N)
INTEGER I, N

INTEGER J
!$OMP DO ! incorrect nesting of loop regions
DO J = 1, N
    CALL WORK(I,J)
END DO
END SUBROUTINE WORK1

SUBROUTINE WRONG2(N)
INTEGER N

INTEGER I
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
    DO I = 1, N
        CALL WORK1(I,N)
    END DO
!$OMP END PARALLEL
END SUBROUTINE WRONG2

The following example is non-conforming because the loop and single regions are closely nested:

Example A.35.3c
void wrong3(int n)
```c
#pragma omp parallel default(shared)
{
    int i;
    #pragma omp for
    for (i=0; i<n; i++) {
        /* incorrect nesting of regions */
        #pragma omp single
        work(i, 0);
    }
}
```

**Example A.35.3f**

```fortran
SUBROUTINE WRONG3(N)
    INTEGER N
    INTEGER I
    !$OMP PARALLEL DEFAULT(SHARED)
    !$OMP DO
    DO I = 1, N
        !$OMP SINGLE ! incorrect nesting of regions
        CALL WORK(I, 1)
        !$OMP END SINGLE
    END DO
    !$OMP END PARALLEL
END SUBROUTINE WRONG3
```

The following example is non-conforming because a **barrier** region cannot be closely nested inside a loop region:

```c
void wrong4(int n)
{
    #pragma omp parallel default(shared)
    {
        int i;
        #pragma omp for
        for (i=0; i<n; i++) {
            work(i, 0);
        /* incorrect nesting of barrier region in a loop region */
            #pragma omp barrier
        }
    }
```
Example A.35.4f

SUBROUTINE WRONG4(N)
  INTEGER N

  INTEGER I
  !$OMP PARALLEL DEFAULT(SHARED)
  !$OMP DO
    DO I = 1, N
      CALL WORK(I, 1)
      !$OMP BARRIER
      CALL WORK(I, 2)
    END DO
  !$OMP END PARALLEL
END SUBROUTINE WRONG4

The following example is non-conforming because the \texttt{barrier} region cannot be closely nested inside the \texttt{critical} region. If this were permitted, it would result in deadlock due to the fact that only one thread at a time can enter the \texttt{critical} region:

Example A.35.5c

void wrong5(int n)
{
  #pragma omp parallel
  {
    #pragma omp critical
    {
      work(n, 0);
      // incorrect nesting of barrier region in a critical region */
      #pragma omp barrier
      work(n, 1);
    }
  }
}
The following example is non-conforming because the \texttt{barrier} region cannot be closely nested inside the \texttt{single} region. If this were permitted, it would result in deadlock due to the fact that only one thread executes the \texttt{single} region:

\begin{verbatim}
Example A.35.6c
void wrong6(int n)
{
    #pragma omp parallel
    {
        #pragma omp single
        {
            work(n, 0);
            /* incorrect nesting of barrier region in a single region */
            #pragma omp barrier
            work(n, 1);
        }
    }
}
\end{verbatim}

\begin{verbatim}
Example A.35.6f
SUBROUTINE WRONG6(N)
    INTEGER N

    !$OMP PARALLEL DEFAULT(SHARED)
    !$OMP SINGLE
    CALL WORK(N,1)
    ! incorrect nesting of barrier region in a single region
\end{verbatim}
A.36 The omp_set_dynamic and omp_set_num_threads Routines

Some programs rely on a fixed, prespecified number of threads to execute correctly. Because the default setting for the dynamic adjustment of the number of threads is implementation defined, such programs can choose to turn off the dynamic threads capability and set the number of threads explicitly to ensure portability. The following example shows how to do this using omp_set_dynamic (Section 3.2.7 on page 97), and omp_set_num_threads (Section 3.2.1 on page 91).

In this example, the program executes correctly only if it is executed by 16 threads. If the implementation is not capable of supporting 16 threads, the behavior of this example is implementation defined. Note that the number of threads executing a parallel region remains constant during the 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.

C/C++

```c
#include <omp.h>
#include <stdlib.h>

void do_by_16(float *x, int iam, int ipoints) {}

void a36(float *x, int npoints)
{
    int iam, ipoints;

    omp_set_dynamic(0);
    omp_set_num_threads(16);

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

Example A.36.1f

SUBROUTINE DO_BY_16(X, IAM, IPOINTS)
   REAL X(*)
   INTEGER IAM, IPOINTS
END SUBROUTINE DO_BY_16

SUBROUTINE SUBA36(X, NPOINTS)
   INCLUDE "omp_lib.h"      ! or USE OMP_LIB
   INTEGER NPOINTS
   REAL X(NPOINTS)
   INTEGER IAM, IPOINTS
   CALL OMP_SET_DYNAMIC(.FALSE.)
   CALL OMP_SET_NUM_THREADS(16)
   !$OMP PARALLEL SHARED(X,NPOINTS) PRIVATE(IAM, IPOINTS)
   IF (OMP_GET_NUM_THREADS() .NE. 16) THEN
      STOP
   ENDIF
   IAM = OMP_GET_THREAD_NUM()
   IPOINTS = NPOINTS/16
   CALL DO_BY_16(X,IAM,IPOINTS)
   !$OMP END PARALLEL
END SUBROUTINE SUBA36
A.37 The \texttt{omp\_get\_num\_threads} Routine

In the following example, the \texttt{omp\_get\_num\_threads} call (see Section 3.2.2 on page 93) returns 1 in the sequential part of the code, so \texttt{np} will always be equal to 1. To determine the number of threads that will be deployed for the \texttt{parallel} region, the call should be inside the \texttt{parallel} region.

\texttt{C/C++}

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

void incorrect()
{
  int np, i;

  np = omp_get_num_threads(); /* misplaced */

  #pragma omp parallel for schedule(static)
  for (i=0; i < np; i++)
    work(i);
}
```

\texttt{Fortran}

```
SUBROUTINE WORK(I)
    INTEGER I
    I = I + 1
END SUBROUTINE WORK

SUBROUTINE INCORRECT()
    INCLUDE "omp\_lib.h" ! or USE OMP\_LIB
    INTEGER I, NP

    NP = OMP\_GET\_NUM\_THREADS() ! misplaced: will return 1

!$OMP PARALLEL DO SCHEDULE(STATIC)
   DO I = 0, NP-1
     CALL WORK(I)
   ENDDO
!$OMP END PARALLEL DO
END SUBROUTINE INCORRECT
```
The following example shows how to rewrite this program without including a query for
the number of threads:

```c++
#include <omp.h>

void work(int i);

void correct()
{
  int i;

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

```fortran
SUBROUTINE WORK(I)
  INTEGER I
  I = I + 1
END SUBROUTINE WORK

SUBROUTINE CORRECT()
  INCLUDE "omp_lib.h" ! or USE OMP_LIB
  INTEGER I
  !$OMP PARALLEL PRIVATE(I)
  I = OMP_GET_THREAD_NUM()
  CALL WORK(I)
  !$OMP END PARALLEL
END SUBROUTINE CORRECT
```
A.38 The omp_init_lock Routine

The following example demonstrates how to initialize an array of locks in a parallel region by using omp_init_lock (Section 3.3.1 on page 104).

**Example A.38.1c**

```c++
#include <omp.h>

omp_lock_t *new_locks()
{
    int i;
    omp_lock_t *lock = new omp_lock_t[1000];

    #pragma omp parallel for private(i)
    for (i=0; i<1000; i++)
    {
        omp_init_lock(&lock[i]);
    }

    return lock;
}
```

**Example A.38.1f**

```fortran
FUNCTION NEW_LOCKS()
    USE OMP_LIB        ! or INCLUDE "omp_lib.h"
    INTEGER(OMP_LOCK_KIND), DIMENSION(1000) :: NEW_LOCKS
    INTEGER I

    !$OMP PARALLEL DO PRIVATE(I)
    DO I=1,1000
        CALL OMP_INIT_LOCK(NEW_LOCKS(I))
    END DO

    !$OMP END PARALLEL DO

    END FUNCTION NEW_LOCKS
```
A.39 Simple Lock Routines

In the following example (for Section 3.3 on page 102), the lock routines cause the threads to be idle while waiting for entry to the first critical section, but to do other work while waiting for entry to the second. The `omp_set_lock` function blocks, but the `omp_test_lock` function does not, allowing the work in `skip` to be done.

```
#include <stdio.h>
#include <omp.h>

void skip(int i) {}
void work(int i) {}

int main()
{
    omp_lock_t lck;
    int id;

    omp_init_lock(&lck);

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

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

        while (!omp_test_lock(&lck)) {
            skip(id); /* we do not yet have the lock,
            so we must do something else */
        }

        work(id); /* we now have the lock
            and can do the work */

        omp_unset_lock(&lck);
    }
```

Note that the argument to the lock routines should have type `omp_lock_t`, and that there is no need to flush it.
omp_destroy_lock(&lck);

return 0;
}

---

Note that there is no need to flush the lock variable.

**Example A.39.1f**

```fortran
SUBROUTINE SKIP(ID)
END SUBROUTINE SKIP

SUBROUTINE WORK(ID)
END SUBROUTINE WORK

PROGRAM A39

INCLUDE "omp_lib.h"     ! or USE OMP_LIB

INTEGER(OMP_LOCK_KIND) LCK
INTEGER ID

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 )
```
A.40 Nestable Lock Routines

The following example (for Section 3.3 on page 102) demonstrates how a nestable lock can be used to synchronize updates both to a whole structure and to one of its members.

```c
#include <omp.h>

typedef struct {
    int a,b;
    omp_nest_lock_t lck; } pair;

int work1();
int work2();
int work3();

void incr_a(pair *p, int a)
{
    /* Called only from incr_pair, no need to lock. */
    p->a += a;
}

void incr_b(pair *p, int b)
{
    /* Called both from incr_pair and elsewhere, */
    /* so need a nestable lock. */

    omp_set_nest_lock(&p->lck);
    p->b += b;
    omp_unset_nest_lock(&p->lck);
}

void incr_pair(pair *p, int a, int b)
{
    omp_set_nest_lock(&p->lck);
    incr_a(p, a);
    incr_b(p, b);
    omp_unset_nest_lock(&p->lck);
}

void a40(pair *p)
{
}
```
#pragma omp parallel sections
{
#pragma omp section
  incr_pair(p, work1(), work2());
#pragma omp section
  incr_b(p, work3());
}

---

C/C++

Fortran

**Example A.40.1f**

```fortran
MODULE DATA
  USE OMP_LIB, ONLY: OMP_NEST_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       ! or INCLUDE "omp_lib.h"
  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       ! or INCLUDE "omp_lib.h"
  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 A40(P)
  USE OMP_LIB        ! or INCLUDE "omp_lib.h"
  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 A40
Stubs for Runtime Library Routines

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

Note that 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.

For the stub routines written in Fortran, the lock variable 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 implementations to implement locks based on the scheme used by the stub procedures.
B.1 C/C++ Stub routines

```c
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"

#ifdef __cplusplus
extern "C" {
#endif

void omp_set_num_threads(int num_threads)
{
}

int omp_get_num_threads(void)
{
    return 1;
}

int omp_get_max_threads(void)
{
    return 1;
}

int omp_get_thread_num(void)
{
    return 0;
}

int omp_get_num_procs(void)
{
    return 1;
}

void omp_set_dynamic(int dynamic_threads)
{
}

#ifdef __cplusplus
}
#endif
```
int omp_get_dynamic(void)
{
    return 0;
}

int omp_in_parallel(void)
{
    return 0;
}

void omp_set_nested(int nested)
{
}

int omp_get_nested(void)
{
    return 0;
}

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

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

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

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

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

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

enum {NOOWNER=-1, MASTER = 0};

void omp_init_nest_lock(omp_nest_lock_t *nlock)
{
    nlock->owner = NOOWNER;
    nlock->count = 0;
}

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

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

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

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

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

#ifdef __cplusplus
}
#endif
```
B.2  Fortran Stub Routines

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

INTEGER FUNCTION OMP_GET_NUM_THREADS()
   OMP_GET_NUM_THREADS = 1
   RETURN
END FUNCTION

INTEGER FUNCTION OMP_GET_MAX_THREADS()
   OMP_GET_MAX_THREADS = 1
   RETURN
END FUNCTION

INTEGER FUNCTION OMP_GET_THREAD_NUM()
   OMP_GET_THREAD_NUM = 0
   RETURN
END FUNCTION

INTEGER FUNCTION OMP_GET_NUM_PROCS()
   OMP_GET_NUM_PROCS = 1
   RETURN
END FUNCTION

SUBROUTINE OMP_SET_DYNAMIC(DYNAMIC_THREADS)
   LOGICAL DYNAMIC_THREADS
   RETURN
END SUBROUTINE
```
LOGICAL FUNCTION OMP_GET_DYNAMIC()
  OMP_GET_DYNAMIC = .FALSE.
  RETURN
END FUNCTION

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

SUBROUTINE OMP_SET_NESTED(NESTED)
  LOGICAL NESTED
  RETURN
END SUBROUTINE

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

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

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

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

    RETURN
END SUBROUTINE

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

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

    RETURN
END SUBROUTINE
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 FUNCTION

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

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

  NLOCK = 0

  RETURN
END SUBROUTINE
SUBROUTINE OMP_SET_NEST_LOCK(NLOCK)
    POINTER (NLOCK,NIL)
    INTEGER NIL
    IF (NLOCK .EQ. -1) THEN
        NLOCK = 1
    ELSEIF (NLOCK .EQ. 0) THEN
        PRINT *, 'ERROR: NESTED LOCK NOT INITIALIZED'
        STOP
    ELSE
        PRINT *, 'ERROR: DEADLOCK USING NESTED LOCK VARIABLE'
        STOP
    ENDIF
    RETURN
END SUBROUTINE

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

DOUBLE PRECISION FUNCTION OMP_GET_WTIME()
! This function does not provide a working
! wall clock timer. Replace it with a version
! customized for the target machine.

    OMP_WTIME = 0.0D0

    RETURN
END FUNCTION

DOUBLE PRECISION FUNCTION OMP_GET_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 FUNCTION
C.1 Notation

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

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

The syntactic expression $\text{term}_{\text{opt seq}}$ is equivalent to $\text{term-sequ}_{\text{opt}}$ with the following additional rules:

\[
\text{term-sequ} : \\
\quad \text{term} \\
\quad \text{term-sequ term} \\
\quad \text{term-sequ , term}
\]
C.2 Rules

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

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

statement-seq:
  statement
  openmp-directive
  statement-seq statement
  statement-seq openmp-directive

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

statement-list:
  statement
  openmp-directive
  statement-list statement
  statement-list openmp-directive

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

block-item:
  declaration
  statement
  openmp-directive
statement:
  /* standard statements */
  openmp-construct

openmp-construct:
  parallel-construct
  for-construct
  sections-construct
  single-construct
  parallel-for-construct
  parallel-sections-construct
  master-construct
  critical-construct
  atomic-construct
  ordered-construct

openmp-directive:
  barrier-directive
  flush-directive

structured-block:
  statement
  parallel-construct:
    parallel-directive structured-block
  parallel-directive:
    # pragma omp parallel parallel-clause optseq new-line
  parallel-clause:
    unique-parallel-clause
    data-clause
unique-parallel-clause:

    if ( expression )

num_threads ( expression )

for-construct:

    for-directive iteration-statement

for-directive:

    # pragma omp for for-clauseoptseq new-line

for-clause:

    unique-for-clause

data-clause

nowait

unique-for-clause:

    ordered

schedule ( schedule-kind )

schedule ( schedule-kind , expression )

schedule-kind:

    static

dynamic

guided

runtime

sections-construct:

    sections-directive section-scope

sections-directive:

    # pragma omp sections sections-clauseoptseq new-line

sections-clause:

    data-clause

nowait

section-scope:
section-sequence:
  section-directive opt structured-block
  section-sequence section-directive structured-block

section-directive:
  # pragma omp section new-line

single-construct:
  single-directive structured-block

single-directive:
  # pragma omp single single-clause optseq new-line

single-clause:
  data-clause
  nowait

parallel-for-construct:
  parallel-for-directive iteration-statement

parallel-for-directive:
  # pragma omp parallel for parallel-for-clause optseq new-line

parallel-for-clause:
  unique-parallel-clause
  unique-for-clause
  data-clause

parallel-sections-construct:
  parallel-sections-directive section-scope

parallel-sections-directive:
  # pragma omp parallel sections parallel-sections-clause optseq new-line

parallel-sections-clause:
  unique-parallel-clause
  data-clause
master-construct:
  master-directive structured-block
master-directive:
  # pragma omp master new-line
critical-construct:
critical-directive structured-block
critical-directive:
  # pragma omp critical region-phrase new-line
region-phrase:
  ( identifier )
barrier-directive:
  # pragma omp barrier new-line
atomic-construct:
  atomic-directive expression-statement
atomic-directive:
  # pragma omp atomic new-line
flush-directive:
  # pragma omp flush flush-vars new-line
flush-vars:
  ( variable-list )
ordered-construct structured-block
ordered-directive:
  # pragma omp ordered new-line
declaration:
  /* standard declarations */
threadprivate-directive
threadprivate-directive:
#pragma omp threadprivate ( variable-list ) new-line

data-clause:
  private ( variable-list )
  copyprivate ( variable-list )
  firstprivate ( variable-list )
  lastprivate ( variable-list )
  shared ( variable-list )
  default ( shared )
  default ( none )
  reduction ( reduction-operator : variable-list )
  copyin ( variable-list )

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

/* in C */

variable-list:
  identifier
  variable-list , identifier

/* in C++ */

variable-list:
  id-expression
  variable-list , id-expression
Interface Declarations

This appendix gives examples of the C/C++ header file, the Fortran include file and Fortran 90 module that shall be provided by implementations as specified in Chapter 3. It also includes an example of a Fortran 90 generic interface for a library routine.

D.1 Example of the omp.h Header File

```c
#ifndef _OMP_H_DEF
#define _OMP_H_DEF

/*
 * define the lock data types
 */
#endif

ifndef __OMP_LOCK_T_DEF
# define __OMP_LOCK_T_DEF
  typedef struct __omp_lock *omp_lock_t;
#endif

ifndef __OMP_NEST_LOCK_T_DEF
# define __OMP_NEST_LOCK_T_DEF
  typedef struct __omp_nest_lock *omp_nest_lock_t;
#endif
```
/*
 * exported OpenMP functions
 */

#ifdef __cplusplus
extern "C" {
#endif

#if defined(__stdc__) || defined(__STDC__) ||
defined(__cplusplus)
extern void omp_set_num_threads(int num_threads);
extern int omp_get_num_threads(void);
extern int omp_get_max_threads(void);
extern int omp_get_thread_num(void);
extern int omp_get_num_procs(void);
extern int omp_in_parallel(void);
extern void omp_set_dynamic(int dynamic_threads);
extern int omp_get_dynamic(void);
extern void omp_set_nested(int nested);
extern int omp_get_nested(void);
extern void omp_init_lock(omp_lock_t *lock);
extern void omp_destroy_lock(omp_lock_t *lock);
extern void omp_set_nest_lock(omp_nest_lock_t *lock);
extern void omp_unset_nest_lock(omp_nest_lock_t *lock);
extern int omp_test_nest_lock(omp_nest_lock_t *lock);
extern double omp_get_wtime(void);
extern double omp_get_wtick(void);
#else
extern void omp_set_num_threads();
extern int omp_get_num_threads();
extern int omp_get_max_threads();
extern int omp_get_thread_num();
extern int omp_get_num_procs();
extern int omp_in_parallel();
extern void omp_set_dynamic();
#endif

#endif
D.2 Example of an Interface Declaration

```c
#include File

extern int omp_get_dynamic();
extern void omp_set_nested();
extern int omp_get_nested();
extern void omp_init_lock();
extern void omp_destroy_lock();
extern void omp_set_lock();
extern void omp_unset_lock();
extern int omp_test_lock();

extern void omp_init_nest_lock();
extern void omp_destroy_nest_lock();
extern void omp_set_nest_lock();
extern void omp_unset_nest_lock();
extern int omp_test_nest_lock();

extern double omp_get_wtime();
extern double omp_get_wtick();
#endif
#ifdef __cplusplus
}
#endif
#endif

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 v2.5

integer     openmp_version
parameter ( openmp_version = 200505 )

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
D.3 Example of a Fortran 90 Interface Declaration

module

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

! 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 v2.5
    integer, parameter :: openmp_version = 200505

    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.4 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 accommodated.

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:

```fortran
! 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 Defined Behaviors in OpenMP

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

- **Memory model**: it is implementation defined as to whether, and in what sizes, memory accesses by multiple threads to the same variable without synchronization are atomic with respect to each other (see Section 1.4.1 on page 10).

- **Internal control variables**: the number of copies of the internal control variables, and their effects, during the execution of any explicit parallel region are implementation defined. The initial values of \texttt{nthreads-var}, \texttt{dyn-var}, \texttt{run-sched-var}, and \texttt{def-sched-var} are implementation defined (see Section 2.3 on page 24).

- **Nested parallelism**: the number of levels of parallelism supported is implementation defined (see Section 1.2.4 on page 8 and Section 2.4.1 on page 29).

- **Dynamic adjustment of threads**: it is implementation defined whether the ability to dynamically adjust the number of threads is provided. Implementations are allowed to deliver fewer threads (but at least one) than indicated in Figure 2-1 in exceptional situations, such as when there is a lack of resources, even if dynamic adjustment is disabled. In these situations, the behavior of the program is implementation defined (see Section 2.4.1 on page 29).

- **sections construct**: the method of scheduling the structured blocks among threads in the team is implementation defined (see Section 2.5.2 on page 39).

- **single construct**: the method of choosing a thread to execute the structured block is implementation defined (see Section 2.5.3 on page 42).

- **atomic construct**: a compliant implementation may enforce exclusive access between \texttt{atomic} regions which update different storage locations. The circumstances under which this occurs are implementation defined (see Section 2.7.4 on page 55).
- **omp_set_num_threads** routine: when called from within any explicit `parallel` region, the binding thread set (and binding region, if required) for the `omp_set_num_threads` region is implementation defined. If the number of threads requested exceeds the number the implementation can support, or is not a positive integer, the behavior of this routine is implementation defined. If this routine is called from within any explicit `parallel` region, the behavior of this routine is implementation defined (see Section 3.2.1 on page 91).

- **omp_get_max_threads** routine: when called from within any explicit `parallel` region, the binding thread set (and binding region, if required) for the `omp_get_max_threads` region is implementation defined (see Section 3.2.3 on page 94).

- **omp_set_dynamic** routine: when called from within any explicit `parallel` region, the binding thread set (and binding region, if required) for the `omp_set_dynamic` region is implementation defined. If called from within any explicit `parallel` region, the behavior of this routine is implementation defined (see Section 3.2.7 on page 97).

- **omp_get_dynamic** routine: when called from within any explicit `parallel` region, the binding thread set (and binding region, if required) for the `omp_get_dynamic` region is implementation defined (see Section 3.2.8 on page 99).

- **omp_set_nested** routine: when called from within any explicit `parallel` region, the binding thread set (and binding region, if required) for the `omp_set_nested` region is implementation defined. If called from within any explicit `parallel` region, the behavior of this routine is implementation defined (see Section 3.2.9 on page 100).

- **omp_get_nested** routine: when called from within any explicit `parallel` region, the binding thread set (and binding region, if required) for the `omp_get_nested` region is implementation defined (see Section 3.2.10 on page 101).

- **OMP_NUM_THREADS** environment variable: if the requested value of **OMP_NUM_THREADS** is greater than the number of threads an implementation can support, or if the value is not a positive integer, the behavior of the program is implementation defined (see Section 4.2 on page 115).

---

**Fortran**

- **threadprivate directive**: if the conditions for values of data in the threadprivate objects of threads (other than the initial thread) to persist between two consecutive active `parallel` regions do not all hold, the allocation status of an allocatable array in the second region is implementation defined (see Section 2.8.2 on page 66).
• **shared clause**: passing a shared variable to a non-intrinsic procedure 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. Situations where this occurs other than those specified are implementation defined (see Section 2.8.3.2 on page 72).

• **Runtime library definitions**: it is implementation defined whether the `include` file `omp_lib.h` or the `module` file `omp_lib` (or both) is provided. It is implementation defined whether any of the OpenMP runtime library routines that take an argument are extended with a generic interface so arguments of different `KIND` type can be accommodated (see Section 3.1 on page 90).
Changes from Version 2.0 to Version 2.5

This appendix summarizes the major changes between the OpenMP API Version 2.0 specifications and the OpenMP API Version 2.5 specification. There are no additional language features in Version 2.5. However, large parts of the text have been rewritten and restructured in order to accommodate all the base languages in a single document, and a number of inconsistencies have been resolved. Only the major changes are listed here.

Terminology

Many terms have been redefined, and the Glossary (Section 1.2 on page 2) has been significantly extended. In particular, readers should note the following changes:

- The Glossary contains new definitions of construct and region. The terms lexical extent and dynamic extent are no longer used.
- The term parallel region is no longer used. Instead, the terms parallel construct and parallel region are used, as appropriate.
- The term serial region is no longer used: this has been replaced with sequential part.
- The Glossary defines binding in terms of binding thread set and binding region.
- The term serialize is no longer used. The terms active and inactive parallel region are used instead.
- The definition of variable differs from the previous definitions.
- The Glossary defines what is meant by supporting nested parallelism.

Memory model

Version 2.5 contains a description of the OpenMP memory model (see Section 1.4 on page 10). This describes the underlying consistency model assumed by OpenMP, and defines the flush operation. It also describes the circumstances under which it is
permitted for one thread to access private variables belonging to another thread. The
memory model states that a race condition on a shared variable results in unspecified
behavior. The relationship between the flush operation and the volatile keyword in
the C and C++ languages is explained.

---

**Fortran**

**PURE and ELEMENTAL Procedures**

OpenMP directives and runtime library routine calls may not appear in PURE or
ELEMENTAL procedures.

---

**Internal Control Variables**

Version 2.5 introduces the notion of internal control variables (see Section 2.3 on page
24), that store the information for determining the number of threads to use for a
parallel region and how to schedule a work-sharing loop. The behavior of certain
execution environment routines (see Section 3.2 on page 91) and environment variables
(see Chapter 4) is described in terms of the internal control variables.

**Determining the Number of Threads for a parallel Region**

The rules which determine the number of threads to use for a parallel region have
been clarified. See Section 2.4.1 on page 29.

**Loop Construct**

The definition of the guided schedule kind has been relaxed: the size of each chunk is
proportional to the number of unassigned iterations divided by the number of threads.
See Section 2.5.1 on page 33.

**sections Construct**

The method of scheduling the structured blocks among threads in the team is
implementation defined in Version 2.5. See Section 2.5.2 on page 39.

**single Construct**

The method of choosing a thread to execute the structured block is implementation
defined in Version 2.5. See Section 2.5.3 on page 42.
Chapter F  Changes from Version 2.0 to Version 2.5

critical Construct

The term critical section is no longer used. Instead, the terms critical construct and critical region are used, as appropriate.

flush Construct

In Version 2.5 it is stated that the flush operation does not imply any ordering between itself and operations on variables not in the flush-set, nor does it imply any ordering between two or more flush constructs if the intersection of their flush-sets is empty (see Section 2.7.5 on page 58). Such implied orderings were assumed in Version 2.0, and as a result, the examples in Section A.18 on page 147 in Version 2.5 differ from the equivalent examples in Version 2.0.

In Version 2.0 no flush operation was implied by calls to OpenMP lock routines. In Version 2.5 a flush without a list is implied whenever a lock is set or unset. See Section 2.7.5 on page 58 and Section 3.3 on page 102.

ordered Construct

The description of the ordered construct has been modified to account for the case where not every iteration of the loop encounters an ordered region.

Sharing Attribute Rules

Version 2.5 clarifies the rules which determine the sharing attributes of variable. See Section 2.8.1 on page 63.

threadprivate Directive

Version 2.5 clarifies the circumstances under which the values of data in the threadprivate objects of threads other than the initial thread are guaranteed to persist between two consecutive active parallel regions. See Section 2.8.2 on page 66.

private Clause

Version 2.5 confirms that variables that appear in expressions for statement function definitions may not appear in a private clause. Section 2.8.3.3 on page 73.
Private Arrays

The behavior of arrays which appear in \texttt{private}, \texttt{firstprivate} and \texttt{lastprivate} clauses has been clarified. See Section 2.8.3.3 on page 73, Section 2.8.3.4 on page 75 and Section 2.8.3.5 on page 77.

\begin{verbatim}

\end{verbatim}

reduction Clause

Fortran pointers and Cray pointers are not permitted in a \texttt{reduction} clause. This restriction was omitted in Version 2.0.

\begin{verbatim}

\end{verbatim}

Data Copying Clauses

In Version 2.5, the \texttt{copyin} and \texttt{copyprivate} clauses are no longer considered data-sharing attribute clauses, but are described as data copying clauses.

\begin{verbatim}

\end{verbatim}

Nesting of Regions

The rules governing the nesting of regions are described using the concept of \texttt{closely nested} regions. See Section 2.9 on page 87.

\begin{verbatim}

\end{verbatim}

Execution Environment Routines

In Version 2.0, the behavior of the \texttt{omp_set_num_threads}, \texttt{omp_set_dynamic} and \texttt{omp_set_nested} routines was undefined if called from an explicit \texttt{parallel} region. In Version 2.5, their behavior in this case is implementation defined. See Section 3.2.1 on page 91, Section 3.2.7 on page 97 and Section 3.2.9 on page 100.

\begin{verbatim}

\end{verbatim}

Examples

The examples in Appendix A have been extended, corrected and reordered in Version 2.5. Where appropriate, equivalent examples have been provided for C/C++ and Fortran.

Except for examples illustrating non-conforming programs, all the examples consist of compilable program units.
Interface Declarations

An example of the `omp.h` header file has been included in Version 2.5.

Using the schedule Clause

This material, which appeared in Appendix D in OpenMP C/C++ Version 2.0 and in Appendix C in OpenMP Fortran Version 2.0, has been removed.