OpenMP Application Program Interface

Version 3.0 May 2008

OpenMP Application Program Interface

Version 3.0 May 2008

Copyright © 1997-2008 OpenMP Architecture Review Board. Permission to copy without fee all or part of this material is granted, provided the OpenMP Architecture Review Board copyright notice and the title of this document appear. Notice is given that copying is by permission of OpenMP Architecture Review Board.

1. Introduction **1**
- 1.1 Scope 1
- 1.2 Glossary 2
  - 1.2.1 Threading Concepts 2
  - 1.2.2 OpenMP language terminology 2
  - 1.2.3 Tasking Terminology 8
  - 1.2.4 Data Terminology 9
  - 1.2.5 Implementation Terminology 10
- 1.3 Execution Model 11
- 1.4 Memory Model 13
  - 1.4.1 Structure of the OpenMP Memory Model 13
  - 1.4.2 The Flush Operation 14
  - 1.4.3 OpenMP Memory Consistency 16
- 1.5 OpenMP Compliance 16
- 1.6 Normative References 17
- 1.7 Organization of this document 18

2. Directives **21**
- 2.1 Directive Format 22
  - 2.1.1 Fixed Source Form Directives 23
  - 2.1.2 Free Source Form Directives 24
- 2.2 Conditional Compilation 26
  - 2.2.1 Fixed Source Form Conditional Compilation Sentinels 26
  - 2.2.2 Free Source Form Conditional Compilation Sentinel 27
- 2.3 Internal Control Variables 28
  - 2.3.1 ICV Descriptions 28
  - 2.3.2 Modifying and Retrieving ICV Values 29
  - 2.3.3 How the Per-task ICVs Work 30
  - 2.3.4 ICV Override Relationships 30
- 2.4 `parallel` Construct 32

i

2.4.1 Determining the Number of Threads for a **parallel** Region 35
2.5 Worksharing Constructs 37
2.5.1 Loop Construct 38
2.5.1.1 Determining the Schedule of a Worksharing Loop 45
2.5.2 **sections** Construct 47
2.5.3 **single** Construct 49
2.5.4 **workshare** Construct 51
2.6 Combined Parallel Worksharing Constructs 54
2.6.1 Parallel Loop construct 54
2.6.2 **parallel sections** Construct 56
2.6.3 **parallel workshare** Construct 58
2.7 **task** Construct 59
2.7.1 Task Scheduling 62
2.8 Master and Synchronization Constructs 63
2.8.1 **master** Construct 63
2.8.2 **critical** Construct 65
2.8.3 **barrier** Construct 66
2.8.4 **taskwait** Construct 68
2.8.5 **atomic** Construct 69
2.8.6 **flush** Construct 72
2.8.7 **ordered** Construct 75
2.9 Data Environment 77
2.9.1 Data-sharing Attribute Rules 77
2.9.1.1 Data-sharing Attribute Rules for Variables Referenced in a Construct 78
2.9.1.2 Data-sharing Attribute Rules for Variables Referenced in a Region but not in a Construct 80
2.9.2 **threadprivate** Directive 81
ii OpenMP ABI • Version 3.0 May 2008

2.9.3 Data-Sharing Attribute Clauses 85
2.9.3.1 **default** clause 86
2.9.3.2 **shared** clause 88
2.9.3.3 **private** clause 89
2.9.3.4 **firstprivate** clause 92
2.9.3.5 **lastprivate** clause 94
2.9.3.6 **reduction** clause 96
2.9.4 Data Copying Clauses 100
2.9.4.1 **copyin** clause 101
2.9.4.2 **copyprivate** clause 102
2.10 Nesting of Regions 104
3. Runtime Library Routines 107
3.1 Runtime Library Definitions 108
3.2 Execution Environment Routines 109
3.2.1 **omp_set_num_threads** 110
3.2.2 **omp_get_num_threads** 111
3.2.3 **omp_get_max_threads** 112
3.2.4 **omp_get_thread_num** 113
3.2.5 **omp_get_num_procs** 115
3.2.6 **omp_in_parallel** 116
3.2.7 **omp_set_dynamic** 117
3.2.8 **omp_get_dynamic** 118
3.2.9 **omp_set_nested** 119
3.2.10 **omp_get_nested** 120
3.2.11 **omp_set_schedule** 121
3.2.12 **omp_get_schedule** 123
3.2.13 **omp_get_thread_limit** 125
3.2.14 **omp_set_max_active_levels** 126
3.2.15 **omp_get_max_active_levels** 127
iii

3.2.16 **omp_get_level** ... 129
3.2.17 **omp_get_ancestor_thread_num** ... 130
3.2.18 **omp_get_team_size** ... 131
3.2.19 **omp_get_active_level** ... 133
3.3 Lock Routines ... 134
3.3.1 **omp_init_lock** and **omp_init_nest_lock** ... 136
3.3.2 **omp_destroy_lock** and **omp_destroy_nest_lock** ... 137
3.3.3 **omp_set_lock** and **omp_set_nest_lock** ... 138
3.3.4 **omp_unset_lock** and **omp_unset_nest_lock** ... 140
3.3.5 **omp_test_lock** and **omp_test_nest_lock** ... 141
3.4 Timing Routines ... 142
3.4.1 **omp_get_wtime** ... 142
3.4.2 **omp_get_wtick** ... 144

4. Environment Variables ... 145
4.1 **OMP_SCHEDULE** ... 146
4.2 **OMP_NUM_THREADS** ... 147
4.3 **OMP_DYNAMIC** ... 148
4.4 **OMP_NESTED** ... 148
4.5 **OMP_STACKSIZE** ... 149
4.6 **OMP_WAIT_POLICY** ... 150
4.7 **OMP_MAX_ACTIVE_LEVELS** ... 150
4.8 **OMP_THREAD_LIMIT** ... 151

A. Examples ... 153
A.1 A Simple Parallel Loop ... 153
A.2 The OpenMP Memory Model ... 154
A.3 Conditional Compilation ... 161
A.4 Internal Control Variables ... 162
A.5 The **parallel** Construct ... 164
A.6 The **num_threads** Clause ... 166

iv OpenMP ABI • Version 3.0 May 2008

A.7 Fortran Restrictions on the **do** Construct ... 167
A.8 Fortran Private Loop Iteration Variables ... 169
A.9 The **nowait** clause ... 170
A.10 The collapse clause ... 173
A.11 The **parallel sections** Construct ... 174
A.12 The **single** Construct ... 176
A.13 Tasking Constructs ... 177
A.14 The **workshare** Construct ... 191
A.15 The **master** Construct ... 195
A.16 The **critical** Construct ... 197
A.17 worksharing Constructs Inside a **critical** Construct ... 199
A.18 Binding of **barrier** Regions ... 200
A.19 The **atomic** Construct ... 202
A.20 Restrictions on the **atomic** Construct ... 205
A.21 The **flush** Construct with a List ... 208
A.22 The **flush** Construct without a List ... 211
A.23 Placement of **flush**, **barrier**, and **taskwait** Directives ... 214
A.24 The **ordered** Clause and the **ordered** Construct ... 215
A.25 The **threadprivate** Directive ... 220
A.26 Parallel Random Access Iterator Loop ... 226
A.27 Fortran Restrictions on **shared** and **private** Clauses with Common Blocks ... 227
A.28 The **default(none)** Clause ... 229
A.29 Race Conditions Caused by Implied Copies of Shared Variable in Fortran ... 231
A.30 The **private** Clause ... 232
A.31 Reprivatization ... 235
A.32 Fortran Restrictions on Storage Association with the **private** Clause ... 237
A.33 C/C++ Arrays in a **firstprivate** Clause ... 240

v

A.34 The **lastprivate** Clause 241
A.35 The **reduction** Clause 242
A.36 The **copyin** Clause 248
A.37 The **copyprivate** Clause 250
A.38 Nested Loop Constructs 255
A.39 Restrictions on Nesting of Regions 258
A.40 The **omp_set_dynamic** and **omp_set_num_threads** Routines 265
A.41 The **omp_get_num_threads** Routine 266
A.42 The **omp_init_lock** Routine 269
A.43 Ownership of Locks 270
A.44 Simple Lock Routines 271
A.45 Nestable Lock Routines 274

B. Stubs for Runtime Library Routines 277
B.1 C/C++ Stub Routines 278
B.2 Fortran Stub Routines 284

C. OpenMP C and C++ Grammar 291
C.1 Notation 291
C.2 Rules 292

D. Interface Declarations 301
D.1 Example of the **omp.h** Header File 302
D.2 Example of an Interface Declaration **include** File 304
D.3 Example of a Fortran 90 Interface Declaration **module** 306
D.4 Example of a Generic Interface for a Library Routine 310

E. Implementation Defined Behaviors in OpenMP 311

F. Changes from Version 2.5 to Version 3.0 315

vi OpenMP ABI • Version 3.0 May 2008

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, tasking constructs, worksharing constructs, and synchronization constructs, and they provide support for sharing and privatizing 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 data dependencies, data conflicts, race conditions, or deadlocks, any of which may occur in conforming programs. In addition, compliant implementations are not required to check for code sequences that cause a program to be classified as non-conforming.

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, tasking constructs, worksharing constructs, and synchronization constructs, and they provide support for sharing and privatizing 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 data dependencies, data conflicts, race conditions, or deadlocks, any of which may occur in conforming programs. In addition, compliant implementations are not required to check for code sequences that cause a program to be classified as non-conforming.

1

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 with a stack and associated static memory, called *threadprivate memory*.

OpenMP thread A *thread* that is managed by the OpenMP runtime system.

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: See Section 1.6 on page 17 for a listing of current *base languages* for OpenMP.

base 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, or an OpenMP construct.

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

COMMENTS:

For all base languages,

- Access to the *structured block* must not be the result of a branch.
- The point of exit cannot be a branch out of the *structured block*.

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 with a stack and associated static memory, called threadprivate memory.

OpenMP thread A thread that is managed by the OpenMP runtime system.

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: See Section 1.6 on page 17 for a listing of current base languages for OpenMP.

base 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, or an OpenMP construct.

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

COMMENTS:

For all base languages,

- Access to the structured block must not be the result of a branch.
- The point of exit cannot be a branch out of the structured block.

2 OpenMP API • Version 3.0 May 2008

For C/C++:

- The point of entry must not 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*.

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

COMMENT: See Section 2.1 on page 22 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 a *base program*, annotated with OpenMP *directives* and runtime library routines.

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

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*.

stand-alone directive An OpenMP *executable directive* that has no associated executable user code.

COMMENT: Only the `barrier`, `flush`, and `taskwait` *directives* are *stand-alone directives*.

For C/C++:

- The point of entry must not 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*.

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

COMMENT: See Section 2.1 on page 22 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 a *base program*, annotated with OpenMP *directives* and runtime library routines.

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

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*.

stand-alone directive An OpenMP *executable directive* that has no associated executable user code.

COMMENT: Only the `barrier`, `flush`, and `taskwait` *directives* are *stand-alone directives*.

Chapter 1 Introduction 3

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 nest 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*.

associated loop(s) The loop(s) controlled by a *loop directive*.

COMMENT: If the *loop directive* contains a **`collapse`** clause then there may be more than one *associated loop*.

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 of an OpenMP library routine. A *region* includes any code in called routines as well as any implicit code introduced by the OpenMP implementation. The generation of a *task* at the point where a **`task`** directive is encountered is a part of the *region* of the *encountering thread*, but the *explicit task region* associated with the **`task`** directive is not.

COMMENTS:

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

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

active parallel region A **`parallel`** *region* that is executed by a *team* consisting of more than one *thread.*

inactive parallel region A **`parallel`** *region* that is executed by a *team* of only one *thread*.

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 nest 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*.

associated loop(s) The loop(s) controlled by a *loop directive*.

COMMENT: If the *loop directive* contains a **`collapse`** clause then there may be more than one *associated loop*.

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 of an OpenMP library routine. A *region* includes any code in called routines as well as any implicit code introduced by the OpenMP implementation. The generation of a *task* at the point where a **`task`** directive is encountered is a part of the *region* of the *encountering thread*, but the *explicit task region* associated with the **`task`** directive is not.

COMMENTS:

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

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

active parallel region A **`parallel`** *region* that is executed by a *team* consisting of more than one *thread.*

inactive parallel region A **`parallel`** *region* that is executed by a *team* of only one *thread*.

4 OpenMP API • Version 3.0 May 2008

sequential part All code encountered during the execution of an *OpenMP program* that is not part of a **`parallel`** *region* corresponding to a **`parallel`** *construct* or a **`task`** *region* corresponding to a **`task`** *construct*.

COMMENTS:

> The *sequential part* executes as if it were enclosed by an *inactive 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.

master thread The *thread* that encounters a **`parallel`** *construct,* creates a *team*, generates a set of *tasks*, then executes one of those *tasks* as *thread* number 0.

parent thread The *thread* that encountered the **`parallel`** *construct* and generated a **`parallel`** *region* is the *parent thread* of each of the *threads* in the *team* of that **`parallel`** *region*. The *master thread* of a **`parallel`** *region* is the same *thread* as its *parent thread* with respect to any resources associated with an *OpenMP thread*.

ancestor thread For a given *thread*, its *parent thread* or one of its *parent thread's ancestor threads*.

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

COMMENTS:

> For an *active parallel region*, the *team* comprises the *master thread* and at least one additional *thread.*
>
> For an *inactive parallel region*, the *team* comprises only the *master thread*.

initial thread The *thread* that executes the *sequential part*.

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

nested construct A *construct* (lexically) enclosed by another *construct*.

nested region A *region* (dynamically) enclosed by another *region*; i.e., a *region* encountered during the execution of another *region*.

COMMENT: Some nestings are *conforming* and some are not. See Section 2.10 on page 104 for the restrictions on nesting.

sequential part All code encountered during the execution of an *OpenMP program* that is not part of a **`parallel`** *region* corresponding to a **`parallel`** *construct* or a **`task`** *region* corresponding to a **`task`** *construct*.

COMMENTS:

> The *sequential part* executes as if it were enclosed by an *inactive 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.

master thread The *thread* that encounters a **`parallel`** *construct,* creates a *team*, generates a set of *tasks*, then executes one of those *tasks* as *thread* number 0.

parent thread The *thread* that encountered the **`parallel`** *construct* and generated a **`parallel`** *region* is the *parent thread* of each of the *threads* in the *team* of that **`parallel`** *region*. The *master thread* of a **`parallel`** *region* is the same *thread* as its *parent thread* with respect to any resources associated with an *OpenMP thread*.

ancestor thread For a given *thread*, its *parent thread* or one of its *parent thread's ancestor threads*.

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

COMMENTS:

> For an *active parallel region*, the *team* comprises the *master thread* and at least one additional *thread.*
>
> For an *inactive parallel region*, the *team* comprises only the *master thread*.

initial thread The *thread* that executes the *sequential part*.

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

nested construct A *construct* (lexically) enclosed by another *construct*.

nested region A *region* (dynamically) enclosed by another *region*; i.e., a *region* encountered during the execution of another *region*.

COMMENT: Some nestings are *conforming* and some are not. See Section 2.10 on page 104 for the restrictions on nesting.

Chapter 1 Introduction 5

closely nested region A *region nested* inside another *region* with no `parallel` *region nested* between them.

all threads All OpenMP *threads* participating in the *OpenMP program*.

current team All *threads* in the *team* executing the innermost enclosing `parallel` *region*

encountering thread For a given *region*, the *thread* that encounters the corresponding *construct*.

all tasks All *tasks* participating in the *OpenMP program*.

current team tasks All *tasks* encountered during the execution of the innermost enclosing `parallel` *region* by the *threads* of the corresponding *team*. Note that the *implicit tasks* constituting the `parallel` *region* and any *descendant tasks* encountered during the execution of these *implicit tasks* are included in this *binding task set*.

generating task For a given *region* the *task* whose execution by a *thread* generated the *region*.

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 *all threads*, the *current team*, or the *encountering thread*.

COMMENT: The *binding thread set* for a particular *region* is described in its corresponding subsection of this specification.

binding task set The set of *tasks* that are affected by, or provide the context for, the execution of a *region*.

The *binding task set* for a given *region* can be *all tasks*, the *current team tasks*, or the *generating task*.

COMMENT: The *binding task set* for a particular *region* (if applicable) is described in its corresponding subsection of this specification.

closely nested region A *region nested* inside another *region* with no `parallel` *region nested* between them.

all threads All OpenMP *threads* participating in the *OpenMP program*.

current team All *threads* in the *team* executing the innermost enclosing `parallel` *region*

encountering thread For a given *region*, the *thread* that encounters the corresponding *construct*.

all tasks All *tasks* participating in the *OpenMP program*.

current team tasks All *tasks* encountered during the execution of the innermost enclosing `parallel` *region* by the *threads* of the corresponding *team*. Note that the *implicit tasks* constituting the `parallel` *region* and any *descendant tasks* encountered during the execution of these *implicit tasks* are included in this *binding task set*.

generating task For a given *region* the *task* whose execution by a *thread* generated the *region*.

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 *all threads*, the *current team*, or the *encountering thread*.

COMMENT: The *binding thread set* for a particular *region* is described in its corresponding subsection of this specification.

binding task set The set of *tasks* that are affected by, or provide the context for, the execution of a *region*.

The *binding task set* for a given *region* can be *all tasks*, the *current team tasks*, or the *generating task*.

COMMENT: The *binding task set* for a particular *region* (if applicable) is described in its corresponding subsection of this specification.

6 OpenMP API • Version 3.0 May 2008

binding region The enclosing *region* that determines the execution context and limits the scope of the effects of the bound *region* is called the *binding region*.

Binding region is not defined for *regions* whose *binding thread set* is *all threads* or the *encountering thread*, nor is it defined for *regions* whose *binding task set* is *all tasks*.

COMMENTS:

> The *binding region* for an **`ordered`** *region* is the innermost enclosing *loop region*.
>
> The *binding region* for a **`taskwait`** *region* is the innermost enclosing *task region*.
>
> For all other *regions* for which the *binding thread set* is the *current team* or the *binding task set* is the *current team tasks*, the *binding region* is the innermost enclosing **`parallel`** *region*.
>
> For *regions* for which the *binding task set* is the generating *task*, the *binding region* is the *region* of the generating *task*.
>
> A **`parallel`** *region* need not be *active* nor explicit to be a *binding region*.
>
> A *task region* need not be explicit 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 one of the *threads* 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`**.

sequential loop A loop that is not associated with any OpenMP *loop directive*.

barrier A point in the execution of a program encountered by a *team* of *threads*, beyond which no *thread* in the team may execute until all *threads* in the *team* have reached the barrier and all *explicit tasks* generated by the *team* have executed to completion.

binding region The enclosing *region* that determines the execution context and limits the scope of the effects of the bound *region* is called the *binding region*.

Binding region is not defined for *regions* whose *binding thread set* is *all threads* or the *encountering thread*, nor is it defined for *regions* whose *binding task set* is *all tasks*.

COMMENTS:

> The *binding region* for an **`ordered`** *region* is the innermost enclosing *loop region*.
>
> The *binding region* for a **`taskwait`** *region* is the innermost enclosing *task region*.
>
> For all other *regions* for which the *binding thread set* is the *current team* or the *binding task set* is the *current team tasks*, the *binding region* is the innermost enclosing **`parallel`** *region*.
>
> For *regions* for which the *binding task set* is the generating *task*, the *binding region* is the *region* of the generating *task*.
>
> A **`parallel`** *region* need not be *active* nor explicit to be a *binding region*.
>
> A *task region* need not be explicit 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 one of the *threads* 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`**.

sequential loop A loop that is not associated with any OpenMP *loop directive*.

barrier A point in the execution of a program encountered by a *team* of *threads*, beyond which no *thread* in the team may execute until all *threads* in the *team* have reached the barrier and all *explicit tasks* generated by the *team* have executed to completion.

Chapter 1 Introduction 7

1.2.3 Tasking Terminology

task A specific instance of executable code and its data environment, generated when a *thread* encounters a **`task`** *construct* or a **`parallel`** *construct*.

COMMENT: When a *thread* executes a *task*, it produces a *task region*.

task region A *region* consisting of all code encountered during the execution of a *task*.

COMMENT: A **`parallel`** *region* consists of one or more implicit *task regions*.

explicit task A *task* generated when a **`task`** *construct* is encountered during execution.

implicit task A *task* generated by the *implicit parallel region* or generated when a **`parallel`** *construct* is encountered during execution.

initial task The *implicit task* associated with the *implicit parallel region*.

current task For a given *thread*, the *task* corresponding to the *task region* in which it is executing.

child task A *task* is a *child task* of the *region* of its generating *task*. A *child task region* is not part of its generating *task region*.

descendant task A *task* that is the *child task* of a *task region* or of one of its *descendant task regions*.

task completion *Task completion* occurs when the end of the *structured block* associated with the *construct* that generated the *task* is reached.

COMMENT: Completion of the *initial task* occurs at program exit.

task scheduling point A point during the execution of the current *task region* at which it can be suspended to be resumed later; or the point of *task completion*, after which the executing thread may switch to a different *task region*.

COMMENT:

Within tied task regions, task scheduling points only appear in the following:

- encountered **`task`** *constructs*
- encountered **`taskwait`** *constructs*
- encountered **`barrier`** *directives*
- implicit **`barrier`** *regions*
- at the end of the *tied task region*

task switching The act of a *thread* switching from the execution of one *task* to another *task*.

1.2.3 Tasking Terminology

task A specific instance of executable code and its data environment, generated when a *thread* encounters a **`task`** *construct* or a **`parallel`** *construct*.

COMMENT: When a *thread* executes a *task*, it produces a *task region*.

task region A *region* consisting of all code encountered during the execution of a *task*.

COMMENT: A **`parallel`** *region* consists of one or more implicit *task regions*.

explicit task A *task* generated when a **`task`** *construct* is encountered during execution.

implicit task A *task* generated by the *implicit parallel region* or generated when a **`parallel`** *construct* is encountered during execution.

initial task The *implicit task* associated with the *implicit parallel region*.

current task For a given *thread*, the *task* corresponding to the *task region* in which it is executing.

child task A *task* is a *child task* of the *region* of its generating *task*. A *child task region* is not part of its generating *task region*.

descendant task A *task* that is the *child task* of a *task region* or of one of its *descendant task regions*.

task completion *Task completion* occurs when the end of the *structured block* associated with the *construct* that generated the *task* is reached.

COMMENT: Completion of the *initial task* occurs at program exit.

task scheduling point A point during the execution of the current *task region* at which it can be suspended to be resumed later; or the point of *task completion*, after which the executing thread may switch to a different *task region*.

COMMENT:

Within tied task regions, task scheduling points only appear in the following:

- encountered **`task`** *constructs*
- encountered **`taskwait`** *constructs*
- encountered **`barrier`** *directives*
- implicit **`barrier`** *regions*
- at the end of the *tied task region*

task switching The act of a *thread* switching from the execution of one *task* to another *task*.

8 OpenMP API • Version 3.0 May 2008

tied task A *task* that, when its *task region* is suspended, can be resumed only by the same *thread* that suspended it; that is, the *task* is tied to that *thread*.

untied task A *task* that, when its *task region* is suspended, can be resumed by any *thread* in the team; that is, the *task* is not tied to any *thread*.

task synchronization construct A **`taskwait`** or a **`barrier`** *construct*.

1.2.4 Data Terminology

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

Array sections and substrings are not considered *variables*.

private variable With respect to a given set of *task regions* that bind to the same **`parallel`** *region*, a *variable* whose name provides access to a different block of storage for each *task region*.

A *variable* which is part of another variable (as an array or structure element) cannot be made private independently of other components.

shared variable With respect to a given set of *task regions* that bind to the same **`parallel`** *region*, a *variable* whose name provides access to the same block of storage for each *task region*.

A *variable* which is part of another variable (as an array or structure element) cannot be *shared* independently of the other components, except for static data members of C++ classes.

threadprivate variable A *variable* that is replicated, one instance per *thread*, by the OpenMP implementation, so that its name provides access to a different block of storage for each *thread*.

A *variable* which is part of another variable (as an array or structure element) cannot be made *threadprivate* independently of the other components, except for static data members of C++ classes.

threadprivate memory The set of *threadprivate variables* associated with each *thread*.

tied task A *task* that, when its *task region* is suspended, can be resumed only by the same *thread* that suspended it; that is, the *task* is tied to that *thread*.

untied task A *task* that, when its *task region* is suspended, can be resumed by any *thread* in the team; that is, the *task* is not tied to any *thread*.

task synchronization construct A **`taskwait`** or a **`barrier`** *construct*.

1.2.4 Data Terminology

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

Array sections and substrings are not considered *variables*.

private variable With respect to a given set of *task regions* that bind to the same **`parallel`** *region*, a *variable* whose name provides access to a different block of storage for each *task region*.

A *variable* which is part of another variable (as an array or structure element) cannot be made private independently of other components.

shared variable With respect to a given set of *task regions* that bind to the same **`parallel`** *region*, a *variable* whose name provides access to the same block of storage for each *task region*.

A *variable* which is part of another variable (as an array or structure element) cannot be *shared* independently of the other components, except for static data members of C++ classes.

threadprivate variable A *variable* that is replicated, one instance per *thread*, by the OpenMP implementation, so that its name provides access to a different block of storage for each *thread*.

A *variable* which is part of another variable (as an array or structure element) cannot be made *threadprivate* independently of the other components, except for static data members of C++ classes.

threadprivate memory The set of *threadprivate variables* associated with each *thread*.

Chapter 1 Introduction 9

data environment All the variables associated with the execution of a given *task*. The *data environment* for a given *task* is constructed from the *data environment* of the *generating task* at the time the *task* is generated.

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*.

class type For C++: Variables declared with one of the **class**, **struct**, or **union** keywords.

1.2.5 Implementation Terminology

supporting *n* levels of parallelism Implies allowing an *active parallel region* to be enclosed by `n-1` *active parallel regions*.

supporting OpenMP Supporting at least one level of parallelism.

supporting nested parallelism Supporting more than one level of parallelism.

internal control variable A conceptual variable that specifies run-time behavior of a set of *threads* or *tasks* in an *OpenMP program*.

COMMENT: The acronym ICV is used interchangeably with the term *internal control variable* in the remainder of this specification.

data environment All the variables associated with the execution of a given *task*. The *data environment* for a given *task* is constructed from the *data environment* of the *generating task* at the time the *task* is generated.

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*.

class type For C++: Variables declared with one of the **class**, **struct**, or **union** keywords.

1.2.5 Implementation Terminology

supporting *n* levels of parallelism Implies allowing an *active parallel region* to be enclosed by `n-1` *active parallel regions*.

supporting OpenMP Supporting at least one level of parallelism.

supporting nested parallelism Supporting more than one level of parallelism.

internal control variable A conceptual variable that specifies run-time behavior of a set of *threads* or *tasks* in an *OpenMP program*.

COMMENT: The acronym ICV is used interchangeably with the term *internal control variable* in the remainder of this specification.

10 OpenMP API • Version 3.0 May 2008

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 must be documented by the implementation, and which is allowed to vary among different *compliant implementations*. An implementation is allowed to define this behavior as *unspecified*.

COMMENT: All features that have *implementation defined* behavior are documented in Appendix E.

1.3 Execution Model

The OpenMP API uses the fork-join model of parallel execution. Multiple threads of execution perform tasks defined implicitly or explicitly by OpenMP directives. 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.

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 must be documented by the implementation, and which is allowed to vary among different *compliant implementations*. An implementation is allowed to define this behavior as *unspecified*.

COMMENT: All features that have *implementation defined* behavior are documented in Appendix E.

1.3 Execution Model

The OpenMP API uses the fork-join model of parallel execution. Multiple threads of execution perform tasks defined implicitly or explicitly by OpenMP directives. 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.

Chapter 1 Introduction 11

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 task region, called the initial task region, that is defined by 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. A set of implicit tasks, one per thread, is generated. The code for each task is defined by the code inside the **`parallel`** construct. Each task is assigned to a different thread in the team and becomes tied; that is, it is always executed by the thread to which it is initially assigned. The task region of the task being executed by the encountering thread is suspended, and each member of the new team executes its implicit task. There is an implicit barrier at the end of the **`parallel`** construct. Beyond the end of the **`parallel`** construct, only the master thread resumes execution, by resuming the task region that was suspended upon encountering 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 worksharing construct, the work inside the construct is divided among the members of the team, and executed cooperatively instead of being executed by every thread. There is an optional barrier at the end of each worksharing construct. Redundant execution of code by every thread in the team resumes after the end of the worksharing construct.

When any thread encounters a **`task`** construct, a new explicit task is generated. Execution of explicitly generated tasks is assigned to one of the threads in the current team, subject to the thread's availability to execute work. Thus, execution of the new task could be immediate, or deferred until later. Threads are allowed to suspend the current task region at a task scheduling point in order to execute a different task. If the suspended task region is for a tied task, the initially assigned thread later resumes execution of the suspended task region. If the suspended task region is for an untied task, then any thread may resume its execution. In untied task regions, task scheduling points may occur at implementation defined points anywhere in the region. In tied task regions, task scheduling points may occur only in **`task`**, **`taskwait`**, explicit or implicit **`barrier`** constructs, and at the completion point of the task. Completion of all explicit tasks bound to a given parallel region is guaranteed before the master thread leaves the implicit barrier at the end of the region. Completion of a subset of all explicit tasks bound to a given parallel region may be specified through the use of task synchronization constructs. Completion of all explicit tasks bound to the implicit parallel region is guaranteed by the time the program exits.

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 task region, called the initial task region, that is defined by 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. A set of implicit tasks, one per thread, is generated. The code for each task is defined by the code inside the **`parallel`** construct. Each task is assigned to a different thread in the team and becomes tied; that is, it is always executed by the thread to which it is initially assigned. The task region of the task being executed by the encountering thread is suspended, and each member of the new team executes its implicit task. There is an implicit barrier at the end of the **`parallel`** construct. Beyond the end of the **`parallel`** construct, only the master thread resumes execution, by resuming the task region that was suspended upon encountering 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 worksharing construct, the work inside the construct is divided among the members of the team, and executed cooperatively instead of being executed by every thread. There is an optional barrier at the end of each worksharing construct. Redundant execution of code by every thread in the team resumes after the end of the worksharing construct.

When any thread encounters a **`task`** construct, a new explicit task is generated. Execution of explicitly generated tasks is assigned to one of the threads in the current team, subject to the thread's availability to execute work. Thus, execution of the new task could be immediate, or deferred until later. Threads are allowed to suspend the current task region at a task scheduling point in order to execute a different task. If the suspended task region is for a tied task, the initially assigned thread later resumes execution of the suspended task region. If the suspended task region is for an untied task, then any thread may resume its execution. In untied task regions, task scheduling points may occur at implementation defined points anywhere in the region. In tied task regions, task scheduling points may occur only in **`task`**, **`taskwait`**, explicit or implicit **`barrier`** constructs, and at the completion point of the task. Completion of all explicit tasks bound to a given parallel region is guaranteed before the master thread leaves the implicit barrier at the end of the region. Completion of a subset of all explicit tasks bound to a given parallel region may be specified through the use of task synchronization constructs. Completion of all explicit tasks bound to the implicit parallel region is guaranteed by the time the program exits.

12 OpenMP API • Version 3.0 May 2008

Synchronization constructs and library routines are available in OpenMP to coordinate tasks and data access in **`parallel`** regions. In addition, library routines and environment variables are available to control or to 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 to 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 to 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 directive that accepts data-sharing attribute clauses determines two kinds of access to variables used in the directive's 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 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 task that contains code associated with the directive. Creation of the new version does not alter the value of the original variable. However, the impact of attempts to access the original variable during the region associated with the directive is unspecified; see Section 2.9.3.3 on page 89 for additional details. References to a private variable in the structured block refer to the current task's private version of the original variable. The

Synchronization constructs and library routines are available in OpenMP to coordinate tasks and data access in **`parallel`** regions. In addition, library routines and environment variables are available to control or to 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 to 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 to 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 directive that accepts data-sharing attribute clauses determines two kinds of access to variables used in the directive's 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 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 task that contains code associated with the directive. Creation of the new version does not alter the value of the original variable. However, the impact of attempts to access the original variable during the region associated with the directive is unspecified; see Section 2.9.3.3 on page 89 for additional details. References to a private variable in the structured block refer to the current task's private version of the original variable. The

Chapter 1 Introduction 13

relationship between the value of the original variable and the initial or final value of the private version depends on the exact clause that specifies it. Details of this issue, as well as other issues with privatization, are provided in Section 2.9 on page 77.

The minimum size at which memory accesses by multiple threads without synchronization, either to the same variable or to different variables that are part of the same variable (as array or structure elements), are atomic with respect to each other, is implementation defined. Any additional atomicity restrictions, such as alignment, are implementation defined.

A single access to a variable may be implemented with multiple load or store instructions, and hence is not guaranteed to be atomic with respect to other accesses to the same variable. Accesses to variables smaller than the implementation-defined minimum size or to C or C++ bit-fields may be implemented by reading, modifying, and rewriting a larger unit of memory, and may thus interfere with updates of variables or fields in the same unit of memory.

If multiple threads write without synchronization to the same memory unit, including cases due to atomicity considerations as described above, then a data race occurs. Similarly, if at least one thread reads from a memory unit and at least one thread writes without synchronization to that same memory unit, including cases due to atomicity considerations as described above, then a data race occurs. If a data race occurs then the result of the program is unspecified.

A private variable in a task region that eventually generates an inner nested **`parallel`** region is permitted to be made shared by implicit tasks in the inner **`parallel`** region. A private variable in a task region can be shared by an explicit **`task`** region generated during its execution. However, it is the programmer's responsibility to ensure through synchronization that the lifetime of the variable does not end before completion of the explicit **`task`** region sharing it. Any other access by one task to the private variables of another task 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.

relationship between the value of the original variable and the initial or final value of the private version depends on the exact clause that specifies it. Details of this issue, as well as other issues with privatization, are provided in Section 2.9 on page 77.

The minimum size at which memory accesses by multiple threads without synchronization, either to the same variable or to different variables that are part of the same variable (as array or structure elements), are atomic with respect to each other, is implementation defined. Any additional atomicity restrictions, such as alignment, are implementation defined.

A single access to a variable may be implemented with multiple load or store instructions, and hence is not guaranteed to be atomic with respect to other accesses to the same variable. Accesses to variables smaller than the implementation-defined minimum size or to C or C++ bit-fields may be implemented by reading, modifying, and rewriting a larger unit of memory, and may thus interfere with updates of variables or fields in the same unit of memory.

If multiple threads write without synchronization to the same memory unit, including cases due to atomicity considerations as described above, then a data race occurs. Similarly, if at least one thread reads from a memory unit and at least one thread writes without synchronization to that same memory unit, including cases due to atomicity considerations as described above, then a data race occurs. If a data race occurs then the result of the program is unspecified.

A private variable in a task region that eventually generates an inner nested **`parallel`** region is permitted to be made shared by implicit tasks in the inner **`parallel`** region. A private variable in a task region can be shared by an explicit **`task`** region generated during its execution. However, it is the programmer's responsibility to ensure through synchronization that the lifetime of the variable does not end before completion of the explicit **`task`** region sharing it. Any other access by one task to the private variables of another task 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.

14 OpenMP API • Version 3.0 May 2008

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 performed a write to its temporary view of a shared 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. If a thread performs multiple writes to the same variable between two flushes of that variable, the flush ensures that the value of the last write is 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 when it may again 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 performed by the thread before the flush are visible in memory to all other threads and that thread's temporary view 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.

Note – OpenMP synchronization operations, described in Section 2.8 on page 63 and in Section 3.3 on page 134, are recommended for enforcing this order. Synchronization through variables is possible; however, it is not recommended since proper timing of flushes is difficult as shown in Section A.2 on page 154.

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 performed a write to its temporary view of a shared 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. If a thread performs multiple writes to the same variable between two flushes of that variable, the flush ensures that the value of the last write is 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 when it may again 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 performed by the thread before the flush are visible in memory to all other threads and that thread's temporary view 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.

Note – OpenMP synchronization operations, described in Section 2.8 on page 63 and in Section 3.3 on page 134, are recommended for enforcing this order. Synchronization through variables is possible; however, it is not recommended since proper timing of flushes is difficult as shown in Section A.2 on page 154.

Chapter 1 Introduction 15

1.4.3 OpenMP Memory Consistency

The type of relaxed memory consistency provided by OpenMP is similar to *weak ordering* as described in S. V. Adve and K. Gharachorloo, “Shared Memory Consistency Models: A Tutorial”, *IEEE Computer, 29(12), pp.66-76, 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. OpenMP does not apply any other restriction to the reordering of memory operations executed by a single thread. The OpenMP memory model is slightly weaker than weak ordering since flushes are not ordered with respect to each other if their flush-sets have an empty intersection.

The restrictions in Section 1.4.2 on page 14 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.
- 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 2.8.6 on page 72 for details. For an example illustrating the memory model, see Section A.2 on page 154.

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 18) 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.

1.4.3 OpenMP Memory Consistency

The type of relaxed memory consistency provided by OpenMP is similar to *weak ordering* as described in S. V. Adve and K. Gharachorloo, “Shared Memory Consistency Models: A Tutorial”, *IEEE Computer, 29(12), pp.66-76, 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. OpenMP does not apply any other restriction to the reordering of memory operations executed by a single thread. The OpenMP memory model is slightly weaker than weak ordering since flushes are not ordered with respect to each other if their flush-sets have an empty intersection.

The restrictions in Section 1.4.2 on page 14 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.
- 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 2.8.6 on page 72 for details. For an example illustrating the memory model, see Section A.2 on page 154.

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 18) 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.

16 OpenMP API • Version 3.0 May 2008

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 (although 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

- ISO/IEC 9899:1990, *Information Technology - Programming Languages - C.*

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

- ISO/IEC 9899:1999, *Information Technology - Programming Languages - C.*

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

- ISO/IEC 14882:1998, *Information Technology - Programming Languages - C++.*

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

- ISO/IEC 1539:1980, *Information Technology - Programming Languages - Fortran.*

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

- ISO/IEC 1539:1991, *Information Technology - Programming Languages - Fortran.*

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

- ISO/IEC 1539-1:1997, *Information Technology - Programming Languages - Fortran.*

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 (although 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

- ISO/IEC 9899:1990, *Information Technology - Programming Languages - C.*

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

- ISO/IEC 9899:1999, *Information Technology - Programming Languages - C.*

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

- ISO/IEC 14882:1998, *Information Technology - Programming Languages - C++.*

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

- ISO/IEC 1539:1980, *Information Technology - Programming Languages - Fortran.*

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

- ISO/IEC 1539:1991, *Information Technology - Programming Languages - Fortran.*

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

- ISO/IEC 1539-1:1997, *Information Technology - Programming Languages - Fortran.*

Chapter 1 Introduction 17

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 2.5 to Version 3.0

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:

C/C++

C/C++ specific text....

C/C++

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

Fortran

Fortran specific text......

Fortran

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

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:

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 2.5 to Version 3.0

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:

C/C++

C/C++ specific text....

C/C++

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

Fortran

Fortran specific text......

Fortran

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

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:

18 OpenMP API • Version 3.0 May 2008

Note – Non-normative text....

Note – Non-normative text....

Chapter 1 Introduction **19**

20 OpenMP API • Version 3.0 May 2008

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 22)
- Mechanisms to control conditional compilation (Section 2.2 on page 26)
- Control of OpenMP API ICVs (Section 2.3 on page 28)
- Details of each OpenMP directive (Section 2.4 on page 32 to Section 2.10 on page 104)

C/C++

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

C/C++

Fortran

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.

Fortran

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.

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 22)
- Mechanisms to control conditional compilation (Section 2.2 on page 26)
- Control of OpenMP API ICVs (Section 2.3 on page 28)
- Details of each OpenMP directive (Section 2.4 on page 32 to Section 2.10 on page 104)

C/C++

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

C/C++

Fortran

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.

Fortran

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.

21

Fortran

Restrictions

The following restriction applies to all OpenMP directives:

- OpenMP directives may not appear in **PURE** or **ELEMENTAL** procedures.

Fortran

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

C/C++

Fortran

OpenMP directives for Fortran are specified as follows:

sentinel directive-name [clause[[,] clause]...]

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 23 and Section 2.1.2 on page 24.

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

Fortran

Restrictions

The following restriction applies to all OpenMP directives:

- OpenMP directives may not appear in **PURE** or **ELEMENTAL** procedures.

Fortran

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

C/C++

Fortran

OpenMP directives for Fortran are specified as follows:

sentinel directive-name [clause[[,] clause]...]

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 23 and Section 2.1.2 on page 24.

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

22 OpenMP API • Version 3.0 May 2008

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.

Fortran

Only one *directive-name* can be specified per directive (note that this includes combined directives, see Section 2.6 on page 54). 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 2.9.3 on page 85), data copying clauses (Section 2.9.4 on page 100), the **`threadprivate`** directive (Section 2.9.2 on page 81) and the **`flush`** directive (Section 2.8.6 on page 72) accept a *list*. A *list* consists of a comma-separated collection of one or more *list items*.

C/C++

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.

C/C++

Fortran

A list item is a variable name or a 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.

Fortran

Fortran

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.

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.

Fortran

Only one *directive-name* can be specified per directive (note that this includes combined directives, see Section 2.6 on page 54). 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 2.9.3 on page 85), data copying clauses (Section 2.9.4 on page 100), the **`threadprivate`** directive (Section 2.9.2 on page 81) and the **`flush`** directive (Section 2.8.6 on page 72) accept a *list*. A *list* consists of a comma-separated collection of one or more *list items*.

C/C++

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.

C/C++

Fortran

A list item is a variable name or a 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.

Fortran

Fortran

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.

Chapter 2 Directives 23

Fortran (cont.)

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):

```
c23456789
!$omp parallel do shared(a,b,c)

c$omp parallel do
c$omp+shared(a,b,c)

c$omp paralleldoshared(a,b,c)
```

2.1.2 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. Continuation 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:

Fortran (cont.)

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):

c23456789
!$omp parallel do shared(a,b,c)

c$omp parallel do
c$omp+shared(a,b,c)

c$omp paralleldoshared(a,b,c)

2.1.2 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. Continuation 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:

24 OpenMP API • Version 3.0 May 2008

```
end critical
end do
end master
end ordered
end parallel
end sections
end single
end task
end workshare
parallel do
parallel sections
parallel workshare
```

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):

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

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

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

Fortran


```
end critical
end do
end master
end ordered
end parallel
end sections
end single
end task
end workshare
parallel do
parallel sections
parallel workshare
```

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):

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

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

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

Fortran

Chapter 2 Directives 25

2.2 Conditional Compilation

In implementations that support a preprocessor, the `_OPENMP` macro name is defined to have the decimal value *yyyymm* 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 161.

Fortran

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:

```
!$ | *$ | 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.

2.2 Conditional Compilation

In implementations that support a preprocessor, the `_OPENMP` macro name is defined to have the decimal value *yyyymm* 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 161.

Fortran

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:

```
!$ | *$ | 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.

26 OpenMP API • Version 3.0 May 2008

Fortran (cont.)

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
```

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

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.

Fortran (cont.)

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
```

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

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.

Chapter 2 Directives 27

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 (ICVs) that control the behavior of an OpenMP program. These ICVs store information such as the number of threads to use for future `parallel` regions, the schedule to use for worksharing loops and whether nested parallelism is enabled or not. The ICVs are given values at various times (described below) during the execution of the program. They are initialized by the implementation itself and may be given values through OpenMP environment variables and through calls to OpenMP API routines. The program can retrieve the values of these ICVs only through OpenMP API routines.

For purposes of exposition, this document refers to the ICVs by certain names, 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 Section 2.3.2 on page 29.

2.3.1 ICV Descriptions

The following ICVs store values that affect the operation of `parallel` regions.

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 (ICVs) that control the behavior of an OpenMP program. These ICVs store information such as the number of threads to use for future `parallel` regions, the schedule to use for worksharing loops and whether nested parallelism is enabled or not. The ICVs are given values at various times (described below) during the execution of the program. They are initialized by the implementation itself and may be given values through OpenMP environment variables and through calls to OpenMP API routines. The program can retrieve the values of these ICVs only through OpenMP API routines.

For purposes of exposition, this document refers to the ICVs by certain names, 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 Section 2.3.2 on page 29.

2.3.1 ICV Descriptions

The following ICVs store values that affect the operation of `parallel` regions.

28 OpenMP API • Version 3.0 May 2008

- *dyn-var* - controls whether dynamic adjustment of the number of threads is enabled for encountered **`parallel`** regions. There is one copy of this ICV per task.
- *nest-var* - controls whether nested parallelism is enabled for encountered **`parallel`** regions. There is one copy of this ICV per task.
- *nthreads-var* - controls the number of threads requested for encountered **`parallel`** regions. There is one copy of this ICV per task.
- *thread-limit-var* - controls the maximum number of threads participating in the OpenMP program. There is one copy of this ICV for the whole program.
- *max-active-levels-var* - controls the maximum number of nested active **`parallel`** regions. There is one copy of this ICV for the whole program.

The following ICVs store values that affect the operation of loop regions.

- *run-sched-var* - controls the schedule that the **`runtime`** schedule clause uses for loop regions. There is one copy of this ICV per task.
- *def-sched-var* - controls the implementation defined default scheduling of loop regions. There is one copy of this ICV for the whole program.

The following ICVs store values that affect the program execution.

- *stacksize-var* - controls the stack size for threads that the OpenMP implementation creates. There is one copy this ICV for the whole program.
- *wait-policy-var* - controls the desired behavior of waiting threads. There is one copy of this ICV for the whole program.

2.3.2 Modifying and Retrieving ICV Values

The following table shows the methods for retrieving the values of the ICVs as well as their initial values:

ICV	Scope	Ways to modify value	Way to retrieve value	Initial value
dyn-var	task	**`OMP_DYNAMIC`** **`omp_set_dynamic()`**	**`omp_get_dynamic()`**	See comments below
nest-var	task	**`OMP_NESTED`** **`omp_set_nested()`**	**`omp_get_nested()`**	*false*
nthreads-var	task	**`OMP_NUM_THREADS`** **`omp_set_num_threads()`**	**`omp_get_max_threads()`**	Implementation defined
run-sched-var	task	**`OMP_SCHEDULE`** **`omp_set_schedule()`**	**`omp_get_schedule()`**	Implementation defined
def-sched-var	global	(none)	(none)	Implementation defined
stacksize-var	global	**`OMP_STACKSIZE`**	(none)	Implementation defined

- *dyn-var* - controls whether dynamic adjustment of the number of threads is enabled for encountered **`parallel`** regions. There is one copy of this ICV per task.
- *nest-var* - controls whether nested parallelism is enabled for encountered **`parallel`** regions. There is one copy of this ICV per task.
- *nthreads-var* - controls the number of threads requested for encountered **`parallel`** regions. There is one copy of this ICV per task.
- *thread-limit-var* - controls the maximum number of threads participating in the OpenMP program. There is one copy of this ICV for the whole program.
- *max-active-levels-var* - controls the maximum number of nested active **`parallel`** regions. There is one copy of this ICV for the whole program.

The following ICVs store values that affect the operation of loop regions.

- *run-sched-var* - controls the schedule that the **`runtime`** schedule clause uses for loop regions. There is one copy of this ICV per task.
- *def-sched-var* - controls the implementation defined default scheduling of loop regions. There is one copy of this ICV for the whole program.

The following ICVs store values that affect the program execution.

- *stacksize-var* - controls the stack size for threads that the OpenMP implementation creates. There is one copy this ICV for the whole program.
- *wait-policy-var* - controls the desired behavior of waiting threads. There is one copy of this ICV for the whole program.

2.3.2 Modifying and Retrieving ICV Values

The following table shows the methods for retrieving the values of the ICVs as well as their initial values:

ICV	Scope	Ways to modify value	Way to retrieve value	Initial value
dyn-var	task	**`OMP_DYNAMIC`** **`omp_set_dynamic()`**	**`omp_get_dynamic()`**	See comments below
nest-var	task	**`OMP_NESTED`** **`omp_set_nested()`**	**`omp_get_nested()`**	*false*
nthreads-var	task	**`OMP_NUM_THREADS`** **`omp_set_num_threads()`**	**`omp_get_max_threads()`**	Implementation defined
run-sched-var	task	**`OMP_SCHEDULE`** **`omp_set_schedule()`**	**`omp_get_schedule()`**	Implementation defined
def-sched-var	global	(none)	(none)	Implementation defined
stacksize-var	global	**`OMP_STACKSIZE`**	(none)	Implementation defined

Chapter 2 Directives 29

ICV	Scope	Ways to modify value	Way to retrieve value	Initial value
wait-policy-var	global	`OMP_WAIT_POLICY`	(none)	Implementation defined
thread-limit-var	global	`OMP_THREAD_LIMIT`	`omp_get_thread_limit()`	Implementation defined
max-active-levels-var	global	`OMP_MAX_ACTIVE_LEVELS` `omp_set_max_active_levels()`	`omp_get_max_active_levels()`	See comments below

Comments:

- The initial value of *dyn-var* is implementation defined if the implementation supports dynamic adjustment of the number of threads; otherwise, the initial value is *false*.
- The initial value of *max-active-levels-var* is the number of levels of parallelism that the implementation supports. See the definition of *supporting n levels of parallelism* in Section 1.2.5 on page 10 for further details.

After the initial values are assigned, but 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 ICVs are modified accordingly. After this point, no changes to any OpenMP environment variables will affect the ICVs.

Clauses on OpenMP constructs do not modify the values of any of the ICVs.

2.3.3 How the Per-task ICVs Work

Each **`task`** region has its own copies of the internal variables *dyn-var*, *nest-var*, *nthreads-var*, and *run-sched-var*. When a **`task`** construct or **`parallel`** construct is encountered during the execution of a task, the child task(s) generated inherit the values of these ICVs from the generating task's ICV values.

Calls to **`omp_set_num_threads()`**, **`omp_set_dynamic()`**, **`omp_set_nested()`**, and **`omp_set_schedule()`** modify only the ICVs corresponding to their binding task region.

When encountering a loop worksharing region with **`schedule(runtime)`**, all implicit task regions that constitute the binding **`parallel`** region must have the same value for *run-sched-var*. Otherwise, the behavior is unspecified.

2.3.4 ICV Override Relationships

The override relationships among various construct clauses, OpenMP API routines, environment variables, and the initial values of ICVs are shown in the following table:

ICV	Scope	Ways to modify value	Way to retrieve value	Initial value
wait-policy-var	global	`OMP_WAIT_POLICY`	(none)	Implementation defined
thread-limit-var	global	`OMP_THREAD_LIMIT`	`omp_get_thread_limit()`	Implementation defined
max-active-levels-var	global	`OMP_MAX_ACTIVE_LEVELS` `omp_set_max_active_levels()`	`omp_get_max_active_levels()`	See comments below

Comments:

- The initial value of *dyn-var* is implementation defined if the implementation supports dynamic adjustment of the number of threads; otherwise, the initial value is *false*.
- The initial value of *max-active-levels-var* is the number of levels of parallelism that the implementation supports. See the definition of *supporting n levels of parallelism* in Section 1.2.5 on page 10 for further details.

After the initial values are assigned, but 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 ICVs are modified accordingly. After this point, no changes to any OpenMP environment variables will affect the ICVs.

Clauses on OpenMP constructs do not modify the values of any of the ICVs.

2.3.3 How the Per-task ICVs Work

Each **`task`** region has its own copies of the internal variables *dyn-var*, *nest-var*, *nthreads-var*, and *run-sched-var*. When a **`task`** construct or **`parallel`** construct is encountered during the execution of a task, the child task(s) generated inherit the values of these ICVs from the generating task's ICV values.

Calls to **`omp_set_num_threads()`**, **`omp_set_dynamic()`**, **`omp_set_nested()`**, and **`omp_set_schedule()`** modify only the ICVs corresponding to their binding task region.

When encountering a loop worksharing region with **`schedule(runtime)`**, all implicit task regions that constitute the binding **`parallel`** region must have the same value for *run-sched-var*. Otherwise, the behavior is unspecified.

2.3.4 ICV Override Relationships

The override relationships among various construct clauses, OpenMP API routines, environment variables, and the initial values of ICVs are shown in the following table:

30 OpenMP API • Version 3.0 May 2008

construct clause, if used	overrides call to API routine	overrides setting of environment variable	overrides initial value of
(none)	`omp_set_dynamic()`	`OMP_DYNAMIC`	*dyn-var*
(none)	`omp_set_nested()`	`OMP_NESTED`	*nest-var*
`num_threads`	`omp_set_num_threads()`	`OMP_NUM_THREADS`	*nthreads-var*
`schedule`	`omp_set_schedule()`	`OMP_SCHEDULE`	*run-sched-var*
`schedule`	(none)	(none)	*def-sched-var*
(none)	(none)	`OMP_STACKSIZE`	*stacksize-var*
(none)	(none)	`OMP_WAIT_POLICY`	*wait-policy-var*
(none)	(none)	`OMP_THREAD_LIMIT`	*thread-limit-var*
(none)	`omp_set_max_active_levels()`	`OMP_MAX_ACTIVE_LEVELS`	*max-active-levels-var*

Cross References:

- **`parallel`** construct, see Section 2.4 on page 32.
- **`num_threads`** clause, see Section 2.4.1 on page 35.
- **`schedule`** clause, see Section 2.5.1.1 on page 45.
- Loop construct, see Section 2.5.1 on page 38.
- **`omp_set_num_threads`** routine, see Section 3.2.1 on page 110.
- **`omp_get_max_threads`** routine, see Section 3.2.3 on page 112.
- **`omp_set_dynamic`** routine, see Section 3.2.7 on page 117.
- **`omp_get_dynamic`** routine, see Section 3.2.8 on page 118.
- **`omp_set_nested`** routine, see Section 3.2.9 on page 119.
- **`omp_get_nested`** routine, see Section 3.2.10 on page 120.
- **`omp_set_schedule`** routine, see Section 3.2.11 on page 121.
- **`omp_get_schedule`** routine, see Section 3.2.12 on page 123.
- **`omp_get_thread_limit`** routine, see Section 3.2.13 on page 125.
- **`omp_set_max_active_levels`** routine, see Section 3.2.14 on page 126.
- **`omp_get_max_active_levels`** routine, see Section 3.2.15 on page 127.
- **`OMP_SCHEDULE`** environment variable, see Section 4.1 on page 146.
- **`OMP_NUM_THREADS`** environment variable, see Section 4.2 on page 147.
- **`OMP_DYNAMIC`** environment variable, see Section 4.3 on page 148.

construct clause, if used	overrides call to API routine	overrides setting of environment variable	overrides initial value of
(none)	`omp_set_dynamic()`	`OMP_DYNAMIC`	*dyn-var*
(none)	`omp_set_nested()`	`OMP_NESTED`	*nest-var*
`num_threads`	`omp_set_num_threads()`	`OMP_NUM_THREADS`	*nthreads-var*
`schedule`	`omp_set_schedule()`	`OMP_SCHEDULE`	*run-sched-var*
`schedule`	(none)	(none)	*def-sched-var*
(none)	(none)	`OMP_STACKSIZE`	*stacksize-var*
(none)	(none)	`OMP_WAIT_POLICY`	*wait-policy-var*
(none)	(none)	`OMP_THREAD_LIMIT`	*thread-limit-var*
(none)	`omp_set_max_active_levels()`	`OMP_MAX_ACTIVE_LEVELS`	*max-active-levels-var*

Cross References:

- **`parallel`** construct, see Section 2.4 on page 32.
- **`num_threads`** clause, see Section 2.4.1 on page 35.
- **`schedule`** clause, see Section 2.5.1.1 on page 45.
- Loop construct, see Section 2.5.1 on page 38.
- **`omp_set_num_threads`** routine, see Section 3.2.1 on page 110.
- **`omp_get_max_threads`** routine, see Section 3.2.3 on page 112.
- **`omp_set_dynamic`** routine, see Section 3.2.7 on page 117.
- **`omp_get_dynamic`** routine, see Section 3.2.8 on page 118.
- **`omp_set_nested`** routine, see Section 3.2.9 on page 119.
- **`omp_get_nested`** routine, see Section 3.2.10 on page 120.
- **`omp_set_schedule`** routine, see Section 3.2.11 on page 121.
- **`omp_get_schedule`** routine, see Section 3.2.12 on page 123.
- **`omp_get_thread_limit`** routine, see Section 3.2.13 on page 125.
- **`omp_set_max_active_levels`** routine, see Section 3.2.14 on page 126.
- **`omp_get_max_active_levels`** routine, see Section 3.2.15 on page 127.
- **`OMP_SCHEDULE`** environment variable, see Section 4.1 on page 146.
- **`OMP_NUM_THREADS`** environment variable, see Section 4.2 on page 147.
- **`OMP_DYNAMIC`** environment variable, see Section 4.3 on page 148.

Chapter 2 Directives 31

- **OMP_NESTED** environment variable, see Section 4.4 on page 148.
- **OMP_STACKSIZE** environment variable, see Section 4.5 on page 149.
- **OMP_WAIT_POLICY** environment variable, see Section 4.6 on page 150.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 4.7 on page 150.
- **OMP_THREAD_LIMIT** environment variable, see Section 4.8 on page 151.

2.4 parallel Construct

Summary

This fundamental construct starts parallel execution. See Section 1.3 on page 11 for a general description of the OpenMP execution model.

Syntax

C/C++

The syntax of the **parallel** construct is as follows:

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

where *clause* is one of the following:

if(*scalar-expression***)**

num_threads(*integer-expression***)**

default(shared|none)

private(*list***)**

firstprivate(*list***)**

shared(*list***)**

copyin(*list***)**

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

C/C++

- **OMP_NESTED** environment variable, see Section 4.4 on page 148.
- **OMP_STACKSIZE** environment variable, see Section 4.5 on page 149.
- **OMP_WAIT_POLICY** environment variable, see Section 4.6 on page 150.
- **OMP_MAX_ACTIVE_LEVELS** environment variable, see Section 4.7 on page 150.
- **OMP_THREAD_LIMIT** environment variable, see Section 4.8 on page 151.

2.4 parallel Construct

Summary

This fundamental construct starts parallel execution. See Section 1.3 on page 11 for a general description of the OpenMP execution model.

Syntax

C/C++

The syntax of the **parallel** construct is as follows:

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

where *clause* is one of the following:

if(*scalar-expression***)**

num_threads(*integer-expression***)**

default(shared|none)

private(*list***)**

firstprivate(*list***)**

shared(*list***)**

copyin(*list***)**

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

C/C++

32 OpenMP API • Version 3.0 May 2008

Fortran

The syntax of the **parallel** construct is as follows:

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

where *clause* is one of the following:

if (*scalar-logical-expression*)

num_threads (*scalar-integer-expression*)

default (**private** | **firstprivate** | **shared** | **none**)

private (*list*)

firstprivate (*list*)

shared (*list*)

copyin (*list*)

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

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

Fortran

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 35 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 that encountered the **parallel** construct becomes the master thread of the new team, with a thread number of zero for the duration of the new **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.

Fortran

The syntax of the **parallel** construct is as follows:

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

where *clause* is one of the following:

if (*scalar-logical-expression*)

num_threads (*scalar-integer-expression*)

default (**private** | **firstprivate** | **shared** | **none**)

private (*list*)

firstprivate (*list*)

shared (*list*)

copyin (*list*)

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

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

Fortran

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 35 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 that encountered the **parallel** construct becomes the master thread of the new team, with a thread number of zero for the duration of the new **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.

Chapter 2 Directives 33

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

A set of implicit tasks, equal in number to the number of threads in the team, is generated by the encountering thread. The structured block of the parallel construct determines the code that will be executed in each implicit task. Each task is assigned to a different thread in the team and becomes tied. The task region of the task being executed by the encountering thread is suspended and each thread in the team executes its implicit task. Each thread can execute a path of statements that is different from that of the other threads.

The implementation may cause any thread to suspend execution of its implicit task at a task scheduling point, and switch to execute any explicit task generated by any of the threads in the team, before eventually resuming execution of the implicit task (for more details see Section 2.7 on page 59).

There is an implied barrier at the end of a **`parallel`** region. After the end of a **`parallel`** region, only the master thread of the team resumes execution of the enclosing task 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 35, 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 work done by a team prior to any barrier that 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.5 on page 164. For an example of the **`num_threads`** clause, see Section A.6 on page 166.

Restrictions

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

- A program that 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.

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

A set of implicit tasks, equal in number to the number of threads in the team, is generated by the encountering thread. The structured block of the parallel construct determines the code that will be executed in each implicit task. Each task is assigned to a different thread in the team and becomes tied. The task region of the task being executed by the encountering thread is suspended and each thread in the team executes its implicit task. Each thread can execute a path of statements that is different from that of the other threads.

The implementation may cause any thread to suspend execution of its implicit task at a task scheduling point, and switch to execute any explicit task generated by any of the threads in the team, before eventually resuming execution of the implicit task (for more details see Section 2.7 on page 59).

There is an implied barrier at the end of a **`parallel`** region. After the end of a **`parallel`** region, only the master thread of the team resumes execution of the enclosing task 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 35, 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 work done by a team prior to any barrier that 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.5 on page 164. For an example of the **`num_threads`** clause, see Section A.6 on page 166.

Restrictions

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

- A program that 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.

34 OpenMP API • Version 3.0 May 2008

C/C++

- A `throw` executed inside a `parallel` region must cause execution to resume within the same `parallel` region, and the same thread that threw the exception must catch it.

C/C++

Fortran

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

Fortran

Cross References

- `default`, `shared`, `private`, `firstprivate`, and `reduction` clauses, see Section 2.9.3 on page 85.
- `copyin` clause, see Section 2.9.4 on page 100.
- `omp_get_thread_num` routine, see Section 3.2.4 on page 113.

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, and the values of the *nthreads-var*, *dyn-var*, *thread-limit-var*, *max-active-level-var*, and *nest-var* ICVs are used to determine the number of threads to use in the region.

Note that using a variable in an `if` or `num_threads` clause expression of a `parallel` construct causes an implicit reference to the variable in all enclosing constructs. 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 encounters a `parallel` construct, the number of threads is determined according to Algorithm 2.1.

Algorithm 2.1

let *ThreadsBusy* be the number of OpenMP threads currently executing;

let *ActiveParRegions* be the number of enclosing active parallel regions;

if an `if` clause exists

C/C++

- A `throw` executed inside a `parallel` region must cause execution to resume within the same `parallel` region, and the same thread that threw the exception must catch it.

C/C++

Fortran

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

Fortran

Cross References

- `default`, `shared`, `private`, `firstprivate`, and `reduction` clauses, see Section 2.9.3 on page 85.
- `copyin` clause, see Section 2.9.4 on page 100.
- `omp_get_thread_num` routine, see Section 3.2.4 on page 113.

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, and the values of the *nthreads-var*, *dyn-var*, *thread-limit-var*, *max-active-level-var*, and *nest-var* ICVs are used to determine the number of threads to use in the region.

Note that using a variable in an `if` or `num_threads` clause expression of a `parallel` construct causes an implicit reference to the variable in all enclosing constructs. 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 encounters a `parallel` construct, the number of threads is determined according to Algorithm 2.1.

Algorithm 2.1

let *ThreadsBusy* be the number of OpenMP threads currently executing;

let *ActiveParRegions* be the number of enclosing active parallel regions;

if an `if` clause exists

Chapter 2 Directives 35

Algorithm 2.1

then let *IfClauseValue* be the value of the `if` clause expression;

else let *IfClauseValue* = *true*;

if a `num_threads` clause exists

then let *ThreadsRequested* be the value of the `num_threads` clause expression;

else let *ThreadsRequested* = *nthreads-var*;

let *ThreadsAvailable* = (*thread-limit-var* - *ThreadsBusy* + 1);

if (*IfClauseValue* = *false*)

then number of threads = 1;

else if (*ActiveParRegions* >= 1) **and** (*nest-var* = *false*)

then number of threads = 1;

else if (*ActiveParRegions* = *max-active-levels-var*)

then number of threads = 1;

else if (*dyn-var* = *true*) **and** (*ThreadsRequested* <= *ThreadsAvailable*)

then number of threads = [1 : *ThreadsRequested*];

else if (*dyn-var* = *true*) **and** (*ThreadsRequested* > *ThreadsAvailable*)

then number of threads = [1 : *ThreadsAvailable*];

else if (*dyn-var* = *false*) **and** (*ThreadsRequested* <= *ThreadsAvailable*)

then number of threads = *ThreadsRequested*;

else if (*dyn-var* = *false*) **and** (*ThreadsRequested* > *ThreadsAvailable*)

then behavior is implementation defined;

Note – Since the initial value of the *dyn-var* ICV 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.

Algorithm 2.1

then let *IfClauseValue* be the value of the `if` clause expression;

else let *IfClauseValue* = *true*;

if a `num_threads` clause exists

then let *ThreadsRequested* be the value of the `num_threads` clause expression;

else let *ThreadsRequested* = *nthreads-var*;

let *ThreadsAvailable* = (*thread-limit-var* - *ThreadsBusy* + 1);

if (*IfClauseValue* = *false*)

then number of threads = 1;

else if (*ActiveParRegions* >= 1) **and** (*nest-var* = *false*)

then number of threads = 1;

else if (*ActiveParRegions* = *max-active-levels-var*)

then number of threads = 1;

else if (*dyn-var* = *true*) **and** (*ThreadsRequested* <= *ThreadsAvailable*)

then number of threads = [1 : *ThreadsRequested*];

else if (*dyn-var* = *true*) **and** (*ThreadsRequested* > *ThreadsAvailable*)

then number of threads = [1 : *ThreadsAvailable*];

else if (*dyn-var* = *false*) **and** (*ThreadsRequested* <= *ThreadsAvailable*)

then number of threads = *ThreadsRequested*;

else if (*dyn-var* = *false*) **and** (*ThreadsRequested* > *ThreadsAvailable*)

then behavior is implementation defined;

Note – Since the initial value of the *dyn-var* ICV 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.

36 OpenMP API • Version 3.0 May 2008

Cross References

- *nthreads-var*, *dyn-var*, *thread-limit-var*, *max-active-level-var*, and *nest-var* ICVs, see Section 2.3 on page 28.

2.5 Worksharing Constructs

A worksharing construct distributes the execution of the associated region among the members of the team that encounters it. Threads execute portions of the region in the context of the implicit tasks each one is executing. If the team consists of only one thread then the worksharing region is not executed in parallel.

A worksharing region has no barrier on entry; however, an implied barrier exists at the end of the worksharing region, unless a **`nowait`** clause is specified. If a **`nowait`** clause is present, an implementation may omit the barrier at the end of the worksharing region. In this case, threads that finish early may proceed straight to the instructions following the worksharing region without waiting for the other members of the team to finish the worksharing region, and without performing a flush operation (see Section A.9 on page 170 for an example.)

OpenMP defines the following worksharing constructs, and these are described in the sections that follow:

- loop construct
- **`sections`** construct
- **`single`** construct
- **`workshare`** construct

Restrictions

The following restrictions apply to worksharing constructs:

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

Cross References

- *nthreads-var*, *dyn-var*, *thread-limit-var*, *max-active-level-var*, and *nest-var* ICVs, see Section 2.3 on page 28.

2.5 Worksharing Constructs

A worksharing construct distributes the execution of the associated region among the members of the team that encounters it. Threads execute portions of the region in the context of the implicit tasks each one is executing. If the team consists of only one thread then the worksharing region is not executed in parallel.

A worksharing region has no barrier on entry; however, an implied barrier exists at the end of the worksharing region, unless a **`nowait`** clause is specified. If a **`nowait`** clause is present, an implementation may omit the barrier at the end of the worksharing region. In this case, threads that finish early may proceed straight to the instructions following the worksharing region without waiting for the other members of the team to finish the worksharing region, and without performing a flush operation (see Section A.9 on page 170 for an example.)

OpenMP defines the following worksharing constructs, and these are described in the sections that follow:

- loop construct
- **`sections`** construct
- **`single`** construct
- **`workshare`** construct

Restrictions

The following restrictions apply to worksharing constructs:

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

Chapter 2 Directives 37

2.5.1 Loop Construct

Summary

The loop construct specifies that the iterations of one or more associated loops will be executed in parallel by threads in the team in the context of their implicit tasks. The iterations 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-loops
```

where *clause* is one of the following:

- **private**(*list*)
- **firstprivate**(*list*)
- **lastprivate**(*list*)
- **reduction**(*operator*: *list*)
- **schedule**(*kind*[, *chunk_size*])
- **collapse**(*n*)
- **ordered**
- **nowait**

2.5.1 Loop Construct

Summary

The loop construct specifies that the iterations of one or more associated loops will be executed in parallel by threads in the team in the context of their implicit tasks. The iterations 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-loops
```

where *clause* is one of the following:

- **private**(*list*)
- **firstprivate**(*list*)
- **lastprivate**(*list*)
- **reduction**(*operator*: *list*)
- **schedule**(*kind*[, *chunk_size*])
- **collapse**(*n*)
- **ordered**
- **nowait**

38 OpenMP API • Version 3.0 May 2008

C/C++ (cont.)

The **for** directive places restrictions on the structure of all associated *for-loops*. Specifically, all associated *for-loops* must have the following canonical form:

for (*init-expr*; *test-expr*; *incr-expr*) *structured-block*	
init-expr	One of the following: *var* **=** *lb* *integer-type var* **=** *lb* *random-access-iterator-type var* = *lb* *pointer-type var* = *lb*
test-expr	One of the following: *var relational-op b* *b relational-op var*
incr-expr	One of the following: **++***var* *var***++** **--***var* *var***--** *var* **+=** *incr* *var* **-=** *incr* *var* **=** *var* **+** *incr* *var* **=** *incr* **+** *var* *var* **=** *var* **-** *incr*
var	One of the following: A variable of a signed or unsigned integer type. For C++, a variable of a random access iterator type. For C, a variable of a pointer type. If this variable would otherwise be shared, it is implicitly made private in 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 unspecified.
relational-op	One of the following: **<** **<=** **>** **>=**
lb and *b*	Loop invariant expressions of a type compatible with the type of *var*.
incr	A loop invariant integer expression.

C/C++ (cont.)

The **for** directive places restrictions on the structure of all associated *for-loops*. Specifically, all associated *for-loops* must have the following canonical form:

for (*init-expr*; *test-expr*; *incr-expr*) *structured-block*	
init-expr	One of the following: *var* **=** *lb* *integer-type var* **=** *lb* *random-access-iterator-type var* = *lb* *pointer-type var* = *lb*
test-expr	One of the following: *var relational-op b* *b relational-op var*
incr-expr	One of the following: **++***var* *var***++** **--***var* *var***--** *var* **+=** *incr* *var* **-=** *incr* *var* **=** *var* **+** *incr* *var* **=** *incr* **+** *var* *var* **=** *var* **-** *incr*
var	One of the following: A variable of a signed or unsigned integer type. For C++, a variable of a random access iterator type. For C, a variable of a pointer type. If this variable would otherwise be shared, it is implicitly made private in 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 unspecified.
relational-op	One of the following: **<** **<=** **>** **>=**
lb and *b*	Loop invariant expressions of a type compatible with the type of *var*.
incr	A loop invariant integer expression.

Chapter 2 Directives 39

The canonical form allows the iteration count of all associated loops to be computed before executing the outermost loop. The computation is performed for each loop in an integer type. This type is derived from the type of *var* as follows:

- If *var* is of an integer type, then the type is the type of *var*.
- For C++, if *var* is of a random access iterator type, then the type is the type that would be used by *std::distance* applied to variables of the type of *var*.
- For C, if *var* is of a pointer type, then the type is *ptrdiff_t*.

The behavior is unspecified if any intermediate result required to compute the iteration count cannot be represented in the type determined above.

There is no implied synchronization during the evaluation of the *lb, b*, or *incr* expressions. It is unspecified whether, in what order, or how many times any side effects within the *lb, b*, or *incr* expressions occur.

Note – Random access iterators are required to support random access to elements in constant time. Other iterators are precluded by the restrictions since they can take linear time or offer limited functionality. It is therefore advisable to use tasks to parallelize those cases.

C/C++

Fortran

The syntax of the loop construct is as follows:

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

where *clause* is one of the following:

private(*list*)

firstprivate(*list*)

lastprivate(*list*)

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

The canonical form allows the iteration count of all associated loops to be computed before executing the outermost loop. The computation is performed for each loop in an integer type. This type is derived from the type of *var* as follows:

- If *var* is of an integer type, then the type is the type of *var*.
- For C++, if *var* is of a random access iterator type, then the type is the type that would be used by *std::distance* applied to variables of the type of *var*.
- For C, if *var* is of a pointer type, then the type is *ptrdiff_t*.

The behavior is unspecified if any intermediate result required to compute the iteration count cannot be represented in the type determined above.

There is no implied synchronization during the evaluation of the *lb, b*, or *incr* expressions. It is unspecified whether, in what order, or how many times any side effects within the *lb, b*, or *incr* expressions occur.

Note – Random access iterators are required to support random access to elements in constant time. Other iterators are precluded by the restrictions since they can take linear time or offer limited functionality. It is therefore advisable to use tasks to parallelize those cases.

C/C++

Fortran

The syntax of the loop construct is as follows:

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

where *clause* is one of the following:

private(*list*)

firstprivate(*list*)

lastprivate(*list*)

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

31 40 OpenMP API • Version 3.0 May 2008

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

collapse(*n***)**

ordered

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

All associated *do-loops* must be *do-constructs* as defined by the Fortran standard. If an **end do** directive follows a *do-construct* in which several loop statements share a **DO** termination statement, then the directive can only be specified for the outermost of these **DO** statements. See Section A.7 on page 167 for examples.

If any of the loop iteration variables would otherwise be shared, they are implicitly made private on the loop construct. See Section A.8 on page 169 for examples. Unless the loop iteration variables are specified **lastprivate** on the loop construct, their values after the loop are unspecified.

Fortran

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

The loop construct is associated with a loop nest consisting of one or more loops that follow the directive.

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

The **collapse** clause may be used to specify how many loops are associated with the loop construct. The parameter of the **collapse** clause must be a constant positive integer expression. If no **collapse** clause is present, the only loop that is associated with the loop construct is the one that immediately follows the construct.

If more than one loop is associated with the loop construct, then the iterations of all associated loops are collapsed into one larger iteration space which is then divided according to the **schedule** clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

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

collapse(*n***)**

ordered

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

All associated *do-loops* must be *do-constructs* as defined by the Fortran standard. If an **end do** directive follows a *do-construct* in which several loop statements share a **DO** termination statement, then the directive can only be specified for the outermost of these **DO** statements. See Section A.7 on page 167 for examples.

If any of the loop iteration variables would otherwise be shared, they are implicitly made private on the loop construct. See Section A.8 on page 169 for examples. Unless the loop iteration variables are specified **lastprivate** on the loop construct, their values after the loop are unspecified.

Fortran

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

The loop construct is associated with a loop nest consisting of one or more loops that follow the directive.

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

The **collapse** clause may be used to specify how many loops are associated with the loop construct. The parameter of the **collapse** clause must be a constant positive integer expression. If no **collapse** clause is present, the only loop that is associated with the loop construct is the one that immediately follows the construct.

If more than one loop is associated with the loop construct, then the iterations of all associated loops are collapsed into one larger iteration space which is then divided according to the **schedule** clause. The sequential execution of the iterations in all associated loops determines the order of the iterations in the collapsed iteration space.

Chapter 2 Directives 41

The iteration count for each associated loop is computed before entry to the outermost loop. If execution of any associated loop changes any of the values used to compute any of the iteration counts then the behavior is unspecified.

The integer type (or kind, for Fortran) used to compute the iteration count for the collapsed loop is implementation defined.

A worksharing loop has logical iterations numbered 0,1,...,N-1 where N is the number of loop iterations, and the logical numbering denotes the sequence in which the iterations would be executed if the associated loop(s) were executed by a single thread. The **`schedule`** clause specifies how iterations of the associated loops are divided into contiguous non-empty subsets, called chunks, and how these chunks are distributed among threads of the team. Each thread executes its assigned chunk(s) in the context of its implicit task. The *chunk_size* expression is evaluated using the original list items of any variables that are made private in the loop construct. It is unspecified whether, in what order, or how many times, any side-effects of the evaluation of this expression occur. The use of a variable in a **`schedule`** clause expression of a loop construct causes an implicit reference to the variable in all enclosing constructs.

Different loop regions with the same schedule and iteration count, even if they occur in the same parallel region, can distribute iterations among threads differently. The only exception is for the **`static`** schedule as specified in Table 2-1. Programs that depend on which thread executes a particular iteration under any other circumstances are non-conforming.

See Section 2.5.1.1 on page 45 for details of how the schedule for a worksharing loop is determined.

The schedule *kind* can be one of those specified in Table 2-1.

The iteration count for each associated loop is computed before entry to the outermost loop. If execution of any associated loop changes any of the values used to compute any of the iteration counts then the behavior is unspecified.

The integer type (or kind, for Fortran) used to compute the iteration count for the collapsed loop is implementation defined.

A worksharing loop has logical iterations numbered 0,1,...,N-1 where N is the number of loop iterations, and the logical numbering denotes the sequence in which the iterations would be executed if the associated loop(s) were executed by a single thread. The **`schedule`** clause specifies how iterations of the associated loops are divided into contiguous non-empty subsets, called chunks, and how these chunks are distributed among threads of the team. Each thread executes its assigned chunk(s) in the context of its implicit task. The *chunk_size* expression is evaluated using the original list items of any variables that are made private in the loop construct. It is unspecified whether, in what order, or how many times, any side-effects of the evaluation of this expression occur. The use of a variable in a **`schedule`** clause expression of a loop construct causes an implicit reference to the variable in all enclosing constructs.

Different loop regions with the same schedule and iteration count, even if they occur in the same parallel region, can distribute iterations among threads differently. The only exception is for the **`static`** schedule as specified in Table 2-1. Programs that depend on which thread executes a particular iteration under any other circumstances are non-conforming.

See Section 2.5.1.1 on page 45 for details of how the schedule for a worksharing loop is determined.

The schedule *kind* can be one of those specified in Table 2-1.

25 42 OpenMP API • Version 3.0 May 2008

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

`static`	When **`schedule(static,`** *chunk_size*) is specified, iterations are divided into chunks of size *chunk_size*, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number. When no *chunk_size* is specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread. Note that the size of the chunks is unspecified in this case. A compliant implementation of **`static`** schedule must ensure that the same assignment of logical iteration numbers to threads will be used in two loop regions if the following conditions are satisfied: 1) both loop regions have the same number of loop iterations, 2) both loop regions have the same value of *chunk_size* specified, or both loop regions have no *chunk_size* specified, and 3) both loop regions bind to the same parallel region. A data dependence between the same logical iterations in two such loops is guaranteed to be satisfied allowing safe use of the **`nowait`** clause (see Section A.9 on page 170 for examples).
`dynamic`	When **`schedule(dynamic,`** *chunk_size*) is specified, the iterations are distributed to threads in the team in chunks as the threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed. Each chunk contains *chunk_size* iterations, except for the last chunk to be distributed, which may have fewer iterations. When no *chunk_size* is specified, it defaults to 1.
`guided`	When **`schedule(guided,`** *chunk_size*) is specified, the iterations are assigned to threads in the team in chunks as the executing threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned. For a *chunk_size* of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to 1. For a *chunk_size* with value *k* (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 *k* iterations (except for the last chunk to be assigned, which may have fewer than *k* iterations). When no *chunk_size* is specified, it defaults to 1.
`auto`	When **`schedule(auto)`** is specified, the decision regarding scheduling is delegated to the compiler and/or runtime system. The programmer gives the implementation the freedom to choose any possible mapping of iterations to threads in the team.

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

`static`	When **`schedule(static,`** *chunk_size*) is specified, iterations are divided into chunks of size *chunk_size*, and the chunks are assigned to the threads in the team in a round-robin fashion in the order of the thread number. When no *chunk_size* is specified, the iteration space is divided into chunks that are approximately equal in size, and at most one chunk is distributed to each thread. Note that the size of the chunks is unspecified in this case. A compliant implementation of **`static`** schedule must ensure that the same assignment of logical iteration numbers to threads will be used in two loop regions if the following conditions are satisfied: 1) both loop regions have the same number of loop iterations, 2) both loop regions have the same value of *chunk_size* specified, or both loop regions have no *chunk_size* specified, and 3) both loop regions bind to the same parallel region. A data dependence between the same logical iterations in two such loops is guaranteed to be satisfied allowing safe use of the **`nowait`** clause (see Section A.9 on page 170 for examples).
`dynamic`	When **`schedule(dynamic,`** *chunk_size*) is specified, the iterations are distributed to threads in the team in chunks as the threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be distributed. Each chunk contains *chunk_size* iterations, except for the last chunk to be distributed, which may have fewer iterations. When no *chunk_size* is specified, it defaults to 1.
`guided`	When **`schedule(guided,`** *chunk_size*) is specified, the iterations are assigned to threads in the team in chunks as the executing threads request them. Each thread executes a chunk of iterations, then requests another chunk, until no chunks remain to be assigned. For a *chunk_size* of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads in the team, decreasing to 1. For a *chunk_size* with value *k* (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 *k* iterations (except for the last chunk to be assigned, which may have fewer than *k* iterations). When no *chunk_size* is specified, it defaults to 1.
`auto`	When **`schedule(auto)`** is specified, the decision regarding scheduling is delegated to the compiler and/or runtime system. The programmer gives the implementation the freedom to choose any possible mapping of iterations to threads in the team.

Chapter 2 Directives **43**

`runtime`	When **`schedule(runtime)`** is specified, the decision regarding scheduling is deferred until run time, and the schedule and chunk size are taken from the *run-sched-var* ICV. If the ICV is set to **`auto`**, the schedule is implementation defined.

Note – For a team of p threads and a loop of n iterations, let $\lceil n/p \rceil$ be the integer q that satisfies $n = p*q - r$, with $0 <= r < p$. 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:

- All loops associated with the loop construct must be perfectly nested; that is, there must be no intervening code nor any OpenMP directive between any two loops.
- The values of the loop control expressions of the loops associated with the loop directive must be the same for all the threads in the team.
- Only one **`schedule`** clause can appear on a loop directive.
- Only one **`collapse`** 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.
- The value of the *run-sched-var* ICV must be the same for all threads in the team.
- When **`schedule(runtime)`** or **`schedule(auto)`** 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.

`runtime`	When **`schedule(runtime)`** is specified, the decision regarding scheduling is deferred until run time, and the schedule and chunk size are taken from the *run-sched-var* ICV. If the ICV is set to **`auto`**, the schedule is implementation defined.

Note – For a team of p threads and a loop of n iterations, let $\lceil n/p \rceil$ be the integer q that satisfies $n = p*q - r$, with $0 <= r < p$. 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:

- All loops associated with the loop construct must be perfectly nested; that is, there must be no intervening code nor any OpenMP directive between any two loops.
- The values of the loop control expressions of the loops associated with the loop directive must be the same for all the threads in the team.
- Only one **`schedule`** clause can appear on a loop directive.
- Only one **`collapse`** 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.
- The value of the *run-sched-var* ICV must be the same for all threads in the team.
- When **`schedule(runtime)`** or **`schedule(auto)`** 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.

44 OpenMP API • Version 3.0 May 2008

- The loop iteration variable may not appear in a **`threadprivate`** directive.

C/C++

- The associated *for-loops* must be structured blocks.
- Only an iteration of the innermost associated loop may be curtailed by a **`continue`** statement.
- No statement can branch to any associated **`for`** statement.
- Only one **`nowait`** clause can appear on a **`for`** directive.
- If *relational-op* is **`<`** or **`<=`** then *incr-expr* must cause *var* to increase on each iteration of the loop. Conversely, if *relational-op* is **`>`** or **`>=`** then *incr-expr* must cause *var* to decrease on each iteration of the loop.
- A throw executed inside a loop region must cause execution to resume within the same iteration of the loop region, and the same thread that threw the exception must catch it.

C/C++

Fortran

- The associated *do-loops* must be structured blocks.
- Only an iteration of the innermost associated loop may be curtailed by a **`CYCLE`** statement.
- No statement in the associated loops other than the **`DO`** statements can cause a branch out of the loops.
- 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.

Fortran

Cross References

- **`private`**, **`firstprivate`**, **`lastprivate`**, and **`reduction`** clauses, see Section 2.9.3 on page 85.
- **`OMP_SCHEDULE`** environment variable, see Section 4.1 on page 146.
- **`ordered`** construct, see Section 2.8.7 on page 75.

2.5.1.1 Determining the Schedule of a Worksharing Loop

When execution encounters a loop directive, the **`schedule`** clause (if any) on the directive, and the *run-sched-var* and *def-sched-var* ICVs are used to determine how loop iterations are assigned to threads. See Section 2.3 on page 28 for details of how the values of the ICVs are determined. If the loop directive does not have a **`schedule`**

- The loop iteration variable may not appear in a **`threadprivate`** directive.

C/C++

- The associated *for-loops* must be structured blocks.
- Only an iteration of the innermost associated loop may be curtailed by a **`continue`** statement.
- No statement can branch to any associated **`for`** statement.
- Only one **`nowait`** clause can appear on a **`for`** directive.
- If *relational-op* is **`<`** or **`<=`** then *incr-expr* must cause *var* to increase on each iteration of the loop. Conversely, if *relational-op* is **`>`** or **`>=`** then *incr-expr* must cause *var* to decrease on each iteration of the loop.
- A throw executed inside a loop region must cause execution to resume within the same iteration of the loop region, and the same thread that threw the exception must catch it.

C/C++

Fortran

- The associated *do-loops* must be structured blocks.
- Only an iteration of the innermost associated loop may be curtailed by a **`CYCLE`** statement.
- No statement in the associated loops other than the **`DO`** statements can cause a branch out of the loops.
- 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.

Fortran

Cross References

- **`private`**, **`firstprivate`**, **`lastprivate`**, and **`reduction`** clauses, see Section 2.9.3 on page 85.
- **`OMP_SCHEDULE`** environment variable, see Section 4.1 on page 146.
- **`ordered`** construct, see Section 2.8.7 on page 75.

2.5.1.1 Determining the Schedule of a Worksharing Loop

When execution encounters a loop directive, the **`schedule`** clause (if any) on the directive, and the *run-sched-var* and *def-sched-var* ICVs are used to determine how loop iterations are assigned to threads. See Section 2.3 on page 28 for details of how the values of the ICVs are determined. If the loop directive does not have a **`schedule`**

Chapter 2 Directives 45

clause then the current value of the *def-sched-var* ICV determines the schedule. If the loop directive has a **`schedule`** clause that specifies the **`runtime`** schedule kind then the current value of the *run-sched-var* ICV determines the schedule. Otherwise, the value of the **`schedule`** clause determines the schedule. Figure 2-1 describes how the schedule for a worksharing loop is determined.

Cross References

- ICVs, see Section 2.3 on page 28.

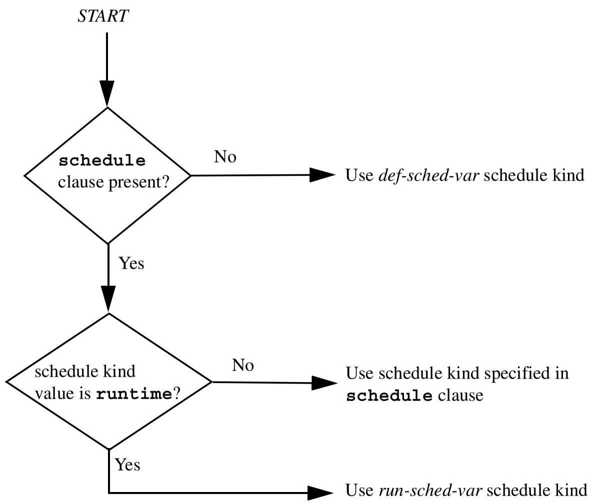

FIGURE 2-1 Determining the schedule for a worksharing loop.

clause then the current value of the *def-sched-var* ICV determines the schedule. If the loop directive has a **`schedule`** clause that specifies the **`runtime`** schedule kind then the current value of the *run-sched-var* ICV determines the schedule. Otherwise, the value of the **`schedule`** clause determines the schedule. Figure 2-1 describes how the schedule for a worksharing loop is determined.

Cross References

- ICVs, see Section 2.3 on page 28.

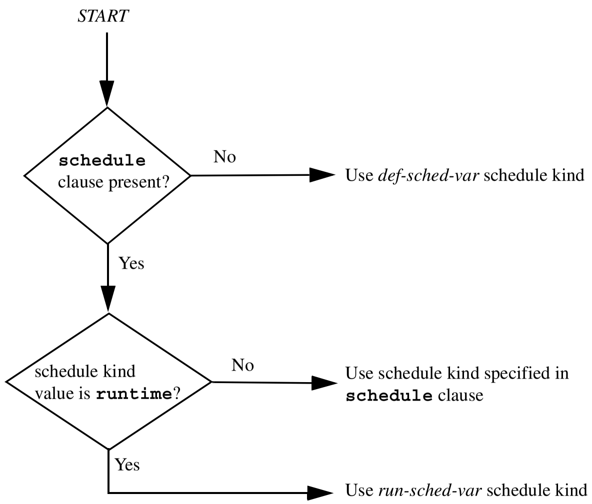

FIGURE 2-1 Determining the schedule for a worksharing loop.

46 OpenMP API • Version 3.0 May 2008

2.5.2 `sections` Construct

Summary

The **`sections`** construct is a noniterative worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team. Each structured block is executed once by one of the threads in the team in the context of its implicit task.

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 ]
   ...
   }
```

where *clause* is one of the following:

`private`(*list*)

`firstprivate`(*list*)

`lastprivate`(*list*)

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

`nowait`

C/C++

2.5.2 sections Construct

Summary

The sections construct is a noniterative worksharing construct that contains a set of structured blocks that are to be distributed among and executed by the threads in a team. Each structured block is executed once by one of the threads in the team in the context of its implicit task.

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]
...
}

where clause is one of the following:

private(list)

firstprivate(list)

lastprivate(list)

reduction(operator: list)

nowait

C/C++

Chapter 2 Directives 47

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]
```

where *clause* is one of the following:

private(*list*)

firstprivate(*list*)

lastprivate(*list*)

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

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 the 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.

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]
```

where *clause* is one of the following:

private(*list*)

firstprivate(*list*)

lastprivate(*list*)

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

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 the 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.

48 OpenMP API • Version 3.0 May 2008

Restrictions

Restrictions to the **`sections`** construct are as follows:

- Orphaned **`section`** directives are prohibited; i.e., 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++

- A throw executed inside a **`sections`** region must cause execution to resume within the same section of the **`sections`** region, and the same thread that threw the exception must catch it.

C/C++

Cross References

- **`private`**, **`firstprivate`**, **`lastprivate`**, and **`reduction`** clauses, see Section 2.9.3 on page 85.

2.5.3 `single` Construct

Summary

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

Syntax

C/C++

The syntax of the **`single`** construct is as follows:

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


Restrictions

Restrictions to the **`sections`** construct are as follows:

- Orphaned **`section`** directives are prohibited; i.e., 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++

- A throw executed inside a **`sections`** region must cause execution to resume within the same section of the **`sections`** region, and the same thread that threw the exception must catch it.

C/C++

Cross References

- **`private`**, **`firstprivate`**, **`lastprivate`**, and **`reduction`** clauses, see Section 2.9.3 on page 85.

2.5.3 `single` Construct

Summary

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

Syntax

C/C++

The syntax of the **`single`** construct is as follows:

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

Chapter 2 Directives 49

where *clause* is one of the following:

private(*list*)
firstprivate(*list*)
copyprivate(*list*)
nowait

C/C++

Fortran

The syntax of the **single** construct is as follows:

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

where *clause* is one of the following:

private(*list*)
firstprivate(*list*)

and *end_clause* is one of the following:

copyprivate(*list*)
nowait

Fortran

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 the (optional) implicit barrier of the **single** region.

where *clause* is one of the following:

private(*list*)
firstprivate(*list*)
copyprivate(*list*)
nowait

C/C++

Fortran

The syntax of the **single** construct is as follows:

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

where *clause* is one of the following:

private(*list*)
firstprivate(*list*)

and *end_clause* is one of the following:

copyprivate(*list*)
nowait

Fortran

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 the (optional) implicit barrier of the **single** region.

50 OpenMP API • Version 3.0 May 2008

Description

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

For an example of the **`single`** construct, see Section A.12 on page 176.

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.

C/C++

- A throw executed inside a **`single`** region must cause execution to resume within the same **`single`** region, and the same thread that threw the exception must catch it.

C/C++

Cross References

- **`private`** and **`firstprivate`** clauses, see Section 2.9.3 on page 85.
- **`copyprivate`** clause, see Section 2.9.4.2 on page 102.

Fortran

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 by one thread, in the context of its implicit task.

Description

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

For an example of the **`single`** construct, see Section A.12 on page 176.

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.

C/C++

- A throw executed inside a **`single`** region must cause execution to resume within the same **`single`** region, and the same thread that threw the exception must catch it.

C/C++

Cross References

- **`private`** and **`firstprivate`** clauses, see Section 2.9.3 on page 85.
- **`copyprivate`** clause, see Section 2.9.4.2 on page 102.

Fortran

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 by one thread, in the context of its implicit task.

Chapter 2 Directives 51

Fortran (cont.)

Syntax

The syntax of the **workshare** construct is as follows:

```
!$omp workshare
    structured-block
!$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 the (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 complete prior to the effects of assigning to the left hand side.

Fortran (cont.)

Syntax

The syntax of the **workshare** construct is as follows:

```
!$omp workshare
    structured-block
!$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 the (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 complete prior to the effects of assigning to the left hand side.

52 OpenMP API • Version 3.0 May 2008

Fortran (cont.)

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.
- For an array assignment statement, the assignment of each element is a unit of work.
- For a scalar assignment statement, the assignment operation is a unit of work.
- For a **WHERE** statement or construct, the evaluation of the mask expression and the masked assignments are each a unit of work.
- For 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 each a unit of work.
- For an **atomic** construct, the update of the scalar variable is a unit of work.
- For a **critical** construct, the construct is a single unit of work.
- For a **parallel** construct, the construct is a unit of work with respect to the **workshare** construct. The statements contained in the **parallel** construct are executed by a new thread team.
- If none of the rules above apply to a portion of a statement in the structured block, then that portion is a 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.14 on page 191.

Fortran (cont.)

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.
- For an array assignment statement, the assignment of each element is a unit of work.
- For a scalar assignment statement, the assignment operation is a unit of work.
- For a **WHERE** statement or construct, the evaluation of the mask expression and the masked assignments are each a unit of work.
- For 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 each a unit of work.
- For an **atomic** construct, the update of the scalar variable is a unit of work.
- For a **critical** construct, the construct is a single unit of work.
- For a **parallel** construct, the construct is a unit of work with respect to the **workshare** construct. The statements contained in the **parallel** construct are executed by a new thread team.
- If none of the rules above apply to a portion of a statement in the structured block, then that portion is a 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.14 on page 191.

Chapter 2 Directives 53

Restrictions

The following restrictions apply to the `workshare` directive:

- The construct must not contain any user defined function calls unless the function is **`ELEMENTAL`**.

Fortran

2.6 Combined Parallel Worksharing Constructs

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

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

The following sections describe the combined parallel worksharing 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.

Restrictions

The following restrictions apply to the `workshare` directive:

- The construct must not contain any user defined function calls unless the function is **`ELEMENTAL`**.

Fortran

2.6 Combined Parallel Worksharing Constructs

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

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

The following sections describe the combined parallel worksharing 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.

54 OpenMP API • Version 3.0 May 2008

Syntax

C/C++

The syntax of the parallel loop construct is as follows:

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

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

C/C++

Fortran

The syntax of the parallel loop construct is as follows:

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

where *clause* can be any of the clauses accepted by the **parallel** or **do** directives, with identical meanings and restrictions.

If an **end parallel do** directive is not specified, an **end parallel do** directive is assumed at the end of the *do-loop*. **nowait** may not be specified on an **end parallel do** directive.

Fortran

Description

C/C++

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

C/C++

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.

Fortran

Syntax

C/C++

The syntax of the parallel loop construct is as follows:

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

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

C/C++

Fortran

The syntax of the parallel loop construct is as follows:

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

where *clause* can be any of the clauses accepted by the **parallel** or **do** directives, with identical meanings and restrictions.

If an **end parallel do** directive is not specified, an **end parallel do** directive is assumed at the end of the *do-loop*. **nowait** may not be specified on an **end parallel do** directive.

Fortran

Description

C/C++

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

C/C++

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.

Fortran

Chapter 2 Directives 55

Restrictions

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

Cross References

- **`parallel`** construct, see Section 2.4 on page 32.
- loop construct, see Section 2.5.1 on page 38.
- Data attribute clauses, see Section 2.9.3 on page 85.

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

C/C++

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

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

where *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++

Restrictions

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

Cross References

- **`parallel`** construct, see Section 2.4 on page 32.
- loop construct, see Section 2.5.1 on page 38.
- Data attribute clauses, see Section 2.9.3 on page 85.

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

C/C++

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

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

where *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++

25 **56** OpenMP API • Version 3.0 May 2008

Fortran

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

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

where *clause* can be any of the clauses accepted by the **`parallel`** or **`sections`** directives, with identical meanings and restrictions.

The last section ends at the **`end parallel sections`** directive. **`nowait`** cannot be specified on an **`end parallel sections`** directive.

Fortran

Description

C/C++

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.11 on page 174.

Restrictions

The restrictions for the **`parallel`** construct and the **`sections`** construct apply.

Cross References:

- **`parallel`** construct, see Section 2.4 on page 32.

Fortran

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

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

where *clause* can be any of the clauses accepted by the **`parallel`** or **`sections`** directives, with identical meanings and restrictions.

The last section ends at the **`end parallel sections`** directive. **`nowait`** cannot be specified on an **`end parallel sections`** directive.

Fortran

Description

C/C++

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.11 on page 174.

Restrictions

The restrictions for the **`parallel`** construct and the **`sections`** construct apply.

Cross References:

- **`parallel`** construct, see Section 2.4 on page 32.

Chapter 2 Directives 57

- **sections** construct, see Section 2.5.2 on page 47.
- Data attribute clauses, see Section 2.9.3 on page 85.

Fortran

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:

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

where *clause* can be any of the clauses accepted by the **parallel** directive, with identical meanings and restrictions. **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 32.
- **workshare** construct, see Section 2.5.4 on page 51.

- **sections** construct, see Section 2.5.2 on page 47.
- Data attribute clauses, see Section 2.9.3 on page 85.

Fortran

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:

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

where *clause* can be any of the clauses accepted by the **parallel** directive, with identical meanings and restrictions. **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 32.
- **workshare** construct, see Section 2.5.4 on page 51.

58 OpenMP API • Version 3.0 May 2008

- Data attribute clauses, see Section 2.9.3 on page 85.

Fortran

2.7 `task` Construct

Summary

The **`task`** construct defines an explicit task.

Syntax

C/C++

The syntax of the **`task`** construct is as follows:

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

where *clause* is one of the following:

```
if(scalar-expression)
untied
default(shared|none)
private(list)
firstprivate(list)
shared(list)
```

C/C++

- Data attribute clauses, see Section 2.9.3 on page 85.

Fortran

2.7 task Construct

Summary

The task construct defines an explicit task.

Syntax

C/C++

The syntax of the task construct is as follows:

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

where clause is one of the following:

if(scalar-expression)
untied
default(shared|none)
private(list)
firstprivate(list)
shared(list)

C/C++

Chapter 2 Directives 59

Fortran

The syntax of the **task** construct is as follows:

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

where *clause* is one of the following:

if(*scalar-logical-expression***)**

untied

default(private | firstprivate | shared | none)

private(*list***)**

firstprivate(*list***)**

shared(*list***)**

Fortran

Binding

The binding thread set of the **task** region is the current parallel team. A **task** region binds to the innermost enclosing **parallel** region.

Description

When a thread encounters a **task** construct, a task is generated from the code for the associated structured block. The data environment of the task is created according to the data-sharing attribute clauses on the **task** construct and any defaults that apply.

The encountering thread may immediately execute the task, or defer its execution. In the latter case, any thread in the team may be assigned the task. Completion of the task can be guaranteed using task synchronization constructs. A **task** construct may be nested inside an outer task, but the **task** region of the inner task is not a part of the **task** region of the outer task.

When an **if** clause is present on a **task** construct and the **if** clause expression evaluates to *false*, the encountering thread must suspend the current task region and begin execution of the generated task immediately, and the suspended task region may

Fortran

The syntax of the **task** construct is as follows:

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

where *clause* is one of the following:

if(*scalar-logical-expression***)**

untied

default(private | firstprivate | shared | none)

private(*list***)**

firstprivate(*list***)**

shared(*list***)**

Fortran

Binding

The binding thread set of the **task** region is the current parallel team. A **task** region binds to the innermost enclosing **parallel** region.

Description

When a thread encounters a **task** construct, a task is generated from the code for the associated structured block. The data environment of the task is created according to the data-sharing attribute clauses on the **task** construct and any defaults that apply.

The encountering thread may immediately execute the task, or defer its execution. In the latter case, any thread in the team may be assigned the task. Completion of the task can be guaranteed using task synchronization constructs. A **task** construct may be nested inside an outer task, but the **task** region of the inner task is not a part of the **task** region of the outer task.

When an **if** clause is present on a **task** construct and the **if** clause expression evaluates to *false*, the encountering thread must suspend the current task region and begin execution of the generated task immediately, and the suspended task region may

60 OpenMP API • Version 3.0 May 2008

not be resumed until the generated task is completed. The task still behaves as a distinct task region with respect to data environment, lock ownership, and synchronization constructs. Note that the use of a variable in an **`if`** clause expression of a **`task`** construct causes an implicit reference to the variable in all enclosing constructs.

A thread that encounters a task scheduling point within the **`task`** region may temporarily suspend the **`task`** region. By default, a task is tied and its suspended task region can only be resumed by the thread that started its execution. If the **`untied`** clause is present on a **`task`** construct, any thread in the team can resume the **`task`** region after a suspension.

The **`task`** construct includes a task scheduling point in the task region of its generating task, immediately following the generation of the explicit task. Each explicit **`task`** region includes a task scheduling point at its point of completion. An implementation may add task scheduling points anywhere in untied **`task`** regions.

Note – When storage is shared by an explicit **`task`** region, it is the programmer's responsibility to ensure, by adding proper synchronization, that the storage does not reach the end of its lifetime before the explicit **`task`** region completes its execution.

Restrictions

Restrictions to the **`task`** construct are as follows:

- A program that branches into or out of a **`task`** region is non-conforming.
- A program must not depend on any ordering of the evaluations of the clauses of the **`task`** directive, or on any side effects of the evaluations of the clauses.
- At most one **`if`** clause can appear on the directive.

C/C++

- A throw executed inside a **`task`** region must cause execution to resume within the same **`task`** region, and the same thread that threw the exception must catch it.

C/C++

Fortran

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

Fortran

not be resumed until the generated task is completed. The task still behaves as a distinct task region with respect to data environment, lock ownership, and synchronization constructs. Note that the use of a variable in an **`if`** clause expression of a **`task`** construct causes an implicit reference to the variable in all enclosing constructs.

A thread that encounters a task scheduling point within the **`task`** region may temporarily suspend the **`task`** region. By default, a task is tied and its suspended task region can only be resumed by the thread that started its execution. If the **`untied`** clause is present on a **`task`** construct, any thread in the team can resume the **`task`** region after a suspension.

The **`task`** construct includes a task scheduling point in the task region of its generating task, immediately following the generation of the explicit task. Each explicit **`task`** region includes a task scheduling point at its point of completion. An implementation may add task scheduling points anywhere in untied **`task`** regions.

Note – When storage is shared by an explicit **`task`** region, it is the programmer's responsibility to ensure, by adding proper synchronization, that the storage does not reach the end of its lifetime before the explicit **`task`** region completes its execution.

Restrictions

Restrictions to the **`task`** construct are as follows:

- A program that branches into or out of a **`task`** region is non-conforming.
- A program must not depend on any ordering of the evaluations of the clauses of the **`task`** directive, or on any side effects of the evaluations of the clauses.
- At most one **`if`** clause can appear on the directive.

C/C++

- A throw executed inside a **`task`** region must cause execution to resume within the same **`task`** region, and the same thread that threw the exception must catch it.

C/C++

Fortran

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

Fortran

Chapter 2 Directives 61

2.7.1 Task Scheduling

Whenever a thread reaches a task scheduling point, the implementation may cause it to perform a task switch, beginning or resuming execution of a different task bound to the current team. Task scheduling points are implied at the following locations:

- the point immediately following the generation of an explicit task
- after the last instruction of a **`task`** region
- in **`taskwait`** regions
- in implicit and explicit *barrier* regions.

In addition, implementations may insert task scheduling points in untied tasks anywhere that they are not specifically prohibited in this specification.

When a thread encounters a task scheduling point it may do one of the following, subject to the *Task Scheduling Constraints* (below):

- begin execution of a tied task bound to the current team.
- resume any suspended task region, bound to the current team, to which it is tied.
- begin execution of an untied task bound to the current team.
- resume any suspended untied task region bound to the current team.

If more than one of the above choices is available, it is unspecified as to which will be chosen.

Task Scheduling Constraints

1. An explicit task whose construct contained an **`if`** clause whose **`if`** clause expression evaluated to *false* is executed immediately after generation of the task.
2. Other scheduling of new tied tasks is constrained by the set of task regions that are currently tied to the thread, and that are not suspended in a **`barrier`** region. If this set is empty, any new tied task may be scheduled. Otherwise, a new tied task may be scheduled only if it is a descendant of every task in the set.

A program relying on any other assumption about task scheduling is non-conforming.

Note – Task scheduling points dynamically divide task regions into parts. Each part is executed uninterruptedly from start to end. Different parts of the same task region are executed in the order in which they are encountered. In the absence of task synchronization constructs, the order in which a thread executes parts of different schedulable tasks is unspecified.

A correct program must behave correctly and consistently with all conceivable scheduling sequences that are compatible with the rules above.

2.7.1 Task Scheduling

Whenever a thread reaches a task scheduling point, the implementation may cause it to perform a task switch, beginning or resuming execution of a different task bound to the current team. Task scheduling points are implied at the following locations:

- the point immediately following the generation of an explicit task
- after the last instruction of a **`task`** region
- in **`taskwait`** regions
- in implicit and explicit *barrier* regions.

In addition, implementations may insert task scheduling points in untied tasks anywhere that they are not specifically prohibited in this specification.

When a thread encounters a task scheduling point it may do one of the following, subject to the *Task Scheduling Constraints* (below):

- begin execution of a tied task bound to the current team.
- resume any suspended task region, bound to the current team, to which it is tied.
- begin execution of an untied task bound to the current team.
- resume any suspended untied task region bound to the current team.

If more than one of the above choices is available, it is unspecified as to which will be chosen.

Task Scheduling Constraints

1. An explicit task whose construct contained an **`if`** clause whose **`if`** clause expression evaluated to *false* is executed immediately after generation of the task.
2. Other scheduling of new tied tasks is constrained by the set of task regions that are currently tied to the thread, and that are not suspended in a **`barrier`** region. If this set is empty, any new tied task may be scheduled. Otherwise, a new tied task may be scheduled only if it is a descendant of every task in the set.

A program relying on any other assumption about task scheduling is non-conforming.

Note – Task scheduling points dynamically divide task regions into parts. Each part is executed uninterruptedly from start to end. Different parts of the same task region are executed in the order in which they are encountered. In the absence of task synchronization constructs, the order in which a thread executes parts of different schedulable tasks is unspecified.

A correct program must behave correctly and consistently with all conceivable scheduling sequences that are compatible with the rules above.

62 OpenMP API • Version 3.0 May 2008

For example, if threadprivate storage is accessed (explicitly in the source code or implicitly in calls to library routines) in one part of a task region, its value cannot be assumed to be preserved into the next part of the same task region if another schedulable task exists that modifies it (see Example A.13.7c on page 186, Example A.13.7f on page 186, Example A.13.8c on page 187 and Example A.13.8f on page 187).

As another example, if a lock acquire and release happen in different parts of a task region, no attempt should be made to acquire the same lock in any part of another task that the executing thread may schedule; otherwise, a deadlock is possible. A similar situation can occur when a critical region spans multiple parts of a task and another schedulable task contains a critical region with the same name (see Example A.13.9c on page 188, Example A.13.9f on page 189, Example A.13.10c on page 190 and Example A.13.10f on page 191).

The use of threadprivate variables and the use of locks or critical sections in an explicit task with an **`if`** clause must take into account that when the **`if`** clause evaluates to false, the task is executed immediately, without regard to *Task Scheduling Constraint* 2.

2.8 Master and Synchronization Constructs

The following sections describe :

- the **`master`** construct.
- the **`critical`** construct.
- the **`barrier`** construct.
- the **`taskwait`** construct.
- the **`atomic`** construct.
- the **`flush`** construct.
- the **`ordered`** construct.

2.8.1 `master` Construct

Summary

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

For example, if threadprivate storage is accessed (explicitly in the source code or implicitly in calls to library routines) in one part of a task region, its value cannot be assumed to be preserved into the next part of the same task region if another schedulable task exists that modifies it (see Example A.13.7c on page 186, Example A.13.7f on page 186, Example A.13.8c on page 187 and Example A.13.8f on page 187).

As another example, if a lock acquire and release happen in different parts of a task region, no attempt should be made to acquire the same lock in any part of another task that the executing thread may schedule; otherwise, a deadlock is possible. A similar situation can occur when a critical region spans multiple parts of a task and another schedulable task contains a critical region with the same name (see Example A.13.9c on page 188, Example A.13.9f on page 189, Example A.13.10c on page 190 and Example A.13.10f on page 191).

The use of threadprivate variables and the use of locks or critical sections in an explicit task with an **`if`** clause must take into account that when the **`if`** clause evaluates to false, the task is executed immediately, without regard to *Task Scheduling Constraint* 2.

2.8 Master and Synchronization Constructs

The following sections describe :

- the **`master`** construct.
- the **`critical`** construct.
- the **`barrier`** construct.
- the **`taskwait`** construct.
- the **`atomic`** construct.
- the **`flush`** construct.
- the **`ordered`** construct.

2.8.1 `master` Construct

Summary

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

Chapter 2 Directives 63

Syntax

C/C++

The syntax of the **master** construct is as follows:

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

C/C++

Fortran

The syntax of the **master** construct is as follows:

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

Fortran

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.15 on page 195.

Restrictions

C/C++

- A throw executed inside a **master** region must cause execution to resume within the same **master** region, and the same thread that threw the exception must catch it.

C/C++

Syntax

C/C++

The syntax of the **master** construct is as follows:

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

C/C++

Fortran

The syntax of the **master** construct is as follows:

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

Fortran

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.15 on page 195.

Restrictions

C/C++

- A throw executed inside a **master** region must cause execution to resume within the same **master** region, and the same thread that threw the exception must catch it.

C/C++

64 OpenMP API • Version 3.0 May 2008

2.8.2 `critical` Construct

Summary

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

Syntax

C/C++

The syntax of the **`critical`** construct is as follows:

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

C/C++

Fortran

The syntax of the **`critical`** construct is as follows:

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

Fortran

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 thread is executing a **`critical`**

2.8.2 critical Construct

Summary

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

Syntax

C/C++

The syntax of the critical construct is as follows:

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

C/C++

Fortran

The syntax of the critical construct is as follows:

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

Fortran

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 thread is executing a critical

Chapter 2 Directives 65

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 those threads in the current team.

C/C++

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

C/C++

Fortran

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.

Fortran

For an example of the **`critical`** construct, see Section A.16 on page 197.

Restrictions

C/C++

- A throw executed inside a **`critical`** region must cause execution to resume within the same **`critical`** region, and the same thread that threw the exception must catch it.

C/C++

Fortran

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.

Fortran

2.8.3 `barrier` Construct

Summary

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

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 those threads in the current team.

C/C++

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

C/C++

Fortran

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.

Fortran

For an example of the **`critical`** construct, see Section A.16 on page 197.

Restrictions

C/C++

- A throw executed inside a **`critical`** region must cause execution to resume within the same **`critical`** region, and the same thread that threw the exception must catch it.

C/C++

Fortran

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.

Fortran

2.8.3 `barrier` Construct

Summary

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

66 OpenMP API • Version 3.0 May 2008

Syntax

C/C++

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

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

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 be placed only at a point where a base language statement is allowed. The **barrier** directive may not be used in place of the statement following an *if*, *while*, *do*, *switch*, or *label*. See Appendix C for the formal grammar. The examples in Section A.23 on page 214 illustrate these restrictions.

C/C++

Fortran

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

```
!$omp barrier
```

Fortran

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.18 on page 200 for examples.

Description

All threads of the team executing the binding **parallel** region must execute the **barrier** region and complete execution of all explicit tasks generated in the binding **parallel** region up to this point before any are allowed to continue execution beyond the barrier.

The **barrier** region includes an implicit task scheduling point in the current task region.

Syntax

C/C++

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

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

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 be placed only at a point where a base language statement is allowed. The **barrier** directive may not be used in place of the statement following an *if*, *while*, *do*, *switch*, or *label*. See Appendix C for the formal grammar. The examples in Section A.23 on page 214 illustrate these restrictions.

C/C++

Fortran

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

```
!$omp barrier
```

Fortran

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.18 on page 200 for examples.

Description

All threads of the team executing the binding **parallel** region must execute the **barrier** region and complete execution of all explicit tasks generated in the binding **parallel** region up to this point before any are allowed to continue execution beyond the barrier.

The **barrier** region includes an implicit task scheduling point in the current task region.

Chapter 2 Directives 67

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 worksharing regions and **barrier** regions encountered must be the same for every thread in a team.

2.8.4 taskwait Construct

Summary

The **taskwait** construct specifies a wait on the completion of child tasks generated since the beginning of the current task.

Syntax

C/C++

The syntax of the **taskwait** construct is as follows:

```
#pragma omp taskwait newline
```

Because the **taskwait** construct does not have a C language statement as part of its syntax, there are some restrictions on its placement within a program. The **taskwait** directive may be placed only at a point where a base language statement is allowed. The **taskwait** directive may not be used in place of the statement following an *if*, *while*, *do*, *switch*, or *label*. See Appendix C for the formal grammar. The examples in Section A.23 on page 214 illustrate these restrictions.

C/C++

Fortran

The syntax of the *taskwait* construct is as follows:

```
!$omp taskwait
```

Fortran

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 worksharing regions and **barrier** regions encountered must be the same for every thread in a team.

2.8.4 taskwait Construct

Summary

The **taskwait** construct specifies a wait on the completion of child tasks generated since the beginning of the current task.

Syntax

C/C++

The syntax of the **taskwait** construct is as follows:

```
#pragma omp taskwait newline
```

Because the **taskwait** construct does not have a C language statement as part of its syntax, there are some restrictions on its placement within a program. The **taskwait** directive may be placed only at a point where a base language statement is allowed. The **taskwait** directive may not be used in place of the statement following an *if*, *while*, *do*, *switch*, or *label*. See Appendix C for the formal grammar. The examples in Section A.23 on page 214 illustrate these restrictions.

C/C++

Fortran

The syntax of the *taskwait* construct is as follows:

```
!$omp taskwait
```

Fortran

68 OpenMP API • Version 3.0 May 2008

Binding

A **`taskwait`** region binds to the current task region. The binding thread set of the **`taskwait`** region is the current team.

Description

The **`taskwait`** region includes an implicit task scheduling point in the current task region. The current task region is suspended at the task scheduling point until execution of all its child tasks generated before the **`taskwait`** region are completed.

2.8.5 `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++

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

```
#pragma omp atomic new-line
    expression-stmt
```

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

x binop`=` *expr*

x`++`

`++`*x*

x`--`

`--`*x*

In the preceding expressions:

- *x* is an lvalue expression with scalar type.

Binding

A **`taskwait`** region binds to the current task region. The binding thread set of the **`taskwait`** region is the current team.

Description

The **`taskwait`** region includes an implicit task scheduling point in the current task region. The current task region is suspended at the task scheduling point until execution of all its child tasks generated before the **`taskwait`** region are completed.

2.8.5 `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++

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

```
#pragma omp atomic new-line
    expression-stmt
```

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

x binop`=` *expr*

x`++`

`++`*x*

x`--`

`--`*x*

In the preceding expressions:

- *x* is an lvalue expression with scalar type.

Chapter 2 Directives 69

- *expr* is an expression with scalar type, and it does not reference the variable designated by *x*.
- *binop* is one of `+`, `*`, `-`, `/`, `&`, `^`, `|`, `<<`, or `>>`.
- *binop*, *binop*`=`, `++`, and `--` are not overloaded operators.

C/C++

Fortran

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

```
!$omp atomic
    statement
```

where *statement* has one of the following forms:

x = x operator expr

x = expr operator x

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

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

In the preceding statements:

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

- *expr* is an expression with scalar type, and it does not reference the variable designated by *x*.
- *binop* is one of `+`, `*`, `-`, `/`, `&`, `^`, `|`, `<<`, or `>>`.
- *binop*, *binop*`=`, `++`, and `--` are not overloaded operators.

C/C++

Fortran

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

```
!$omp atomic
    statement
```

where *statement* has one of the following forms:

x = x operator expr

x = expr operator x

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

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

In the preceding statements:

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

70 OpenMP API • Version 3.0 May 2008

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 teams to which the threads belong.

Description

Only the load and store of the variable designated by *x* are atomic; the evaluation of *expr* is not atomic. No task scheduling points are allowed between the load and the store of the variable designated by *x*. To avoid race conditions, all updates of the location that 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 that access the same storage location *x*. However, other OpenMP synchronization can ensure the desired exclusive access. For example, a barrier following a series of atomic updates to x guarantees that subsequent accesses do not form a race with the atomic accesses.

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

For an example of the **`atomic`** construct, see Section A.19 on page 202.

Restrictions

C/C++

The following restriction applies to the **`atomic`** construct:

- All atomic references to the storage location *x* throughout the program are required to have a compatible type. See Section A.20 on page 205 for examples.

C/C++

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.20 on page 205 for examples.

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 teams to which the threads belong.

Description

Only the load and store of the variable designated by *x* are atomic; the evaluation of *expr* is not atomic. No task scheduling points are allowed between the load and the store of the variable designated by *x*. To avoid race conditions, all updates of the location that 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 that access the same storage location *x*. However, other OpenMP synchronization can ensure the desired exclusive access. For example, a barrier following a series of atomic updates to x guarantees that subsequent accesses do not form a race with the atomic accesses.

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

For an example of the **`atomic`** construct, see Section A.19 on page 202.

Restrictions

C/C++

The following restriction applies to the **`atomic`** construct:

- All atomic references to the storage location *x* throughout the program are required to have a compatible type. See Section A.20 on page 205 for examples.

C/C++

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.20 on page 205 for examples.

Fortran

Chapter 2 Directives 71

Cross References

- **`critical`** construct, see Section 2.8.2 on page 65.

2.8.6 `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 13 for more details.

Syntax

C/C++

The syntax of the **`flush`** construct is as follows:

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

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 **`taskwait`** directive may be placed only at a point where a base language statement is allowed. The **`taskwait`** directive may not be used in place of the statement following an *if*, *while*, *do*, *switch*, or *label*. See Appendix C for the formal grammar. See Section A.23 on page 214 for an example that illustrates these placement restrictions.

C/C++

Fortran

The syntax of the **`flush`** construct is as follows:

```
!$omp flush [(list)]
```

Fortran

Cross References

- **`critical`** construct, see Section 2.8.2 on page 65.

2.8.6 `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 13 for more details.

Syntax

C/C++

The syntax of the **`flush`** construct is as follows:

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

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 **`taskwait`** directive may be placed only at a point where a base language statement is allowed. The **`taskwait`** directive may not be used in place of the statement following an *if*, *while*, *do*, *switch*, or *label*. See Appendix C for the formal grammar. See Section A.23 on page 214 for an example that illustrates these placement restrictions.

C/C++

Fortran

The syntax of the **`flush`** construct is as follows:

```
!$omp flush [(list)]
```

Fortran

72 OpenMP API • Version 3.0 May 2008

Binding

The binding thread set for a **`flush`** region is the encountering thread. Execution of a **`flush`** region affects the memory and the temporary view of memory of only the thread that executes the region. It does not affect the temporary view of other threads. Other threads must themselves execute a flush operation in order to be guaranteed 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.

C/C++

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

C/C++

Fortran

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.

Fortran

For examples of the **`flush`** construct, see Section A.21 on page 208 and Section A.22 on page 211.

Binding

The binding thread set for a **`flush`** region is the encountering thread. Execution of a **`flush`** region affects the memory and the temporary view of memory of only the thread that executes the region. It does not affect the temporary view of other threads. Other threads must themselves execute a flush operation in order to be guaranteed 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.

C/C++

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

C/C++

Fortran

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.

Fortran

For examples of the **`flush`** construct, see Section A.21 on page 208 and Section A.22 on page 211.

Chapter 2 Directives **73**

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:

```
                    a = b = 0

thread 1                       thread 2

b = 1                          a = 1
flush(b)                       flush(a)
flush(a)                       flush(b)
if (a == 0) then               if (b == 0) then
    critical section               critical section
end if                         end if
```

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, both threads can simultaneously executed the critical section.

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. This occurs if both flushes complete prior to either thread executing its **if** statement.

Correct example:

```
                    a = b = 0

thread 1                       thread 2

b = 1                          a = 1
flush(a,b)                     flush(a,b)
if (a == 0) then               if (b == 0) then
    critical section               critical section
end if                         end if
```


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:

```
                    a = b = 0

thread 1                       thread 2

b = 1                          a = 1
flush(b)                       flush(a)
flush(a)                       flush(b)
if (a == 0) then               if (b == 0) then
    critical section               critical section
end if                         end if
```

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, both threads can simultaneously executed the critical section.

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. This occurs if both flushes complete prior to either thread executing its **if** statement.

Correct example:

```
                    a = b = 0

thread 1                       thread 2

b = 1                          a = 1
flush(a,b)                     flush(a,b)
if (a == 0) then               if (b == 0) then
    critical section               critical section
end if                         end if
```

74 OpenMP API • Version 3.0 May 2008

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.

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 worksharing regions unless a **`nowait`** is present.
- At entry to and exit from combined parallel worksharing 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.
- Immediately before and immediately after every task scheduling point.

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 variable updated in the **`atomic`** construct.

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

- At entry to worksharing regions.
- At entry to or exit from a **`master`** region.

2.8.7 `ordered` Construct

Summary

The **`ordered`** construct specifies a structured block in a loop region that 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.

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.

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 worksharing regions unless a **`nowait`** is present.
- At entry to and exit from combined parallel worksharing 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.
- Immediately before and immediately after every task scheduling point.

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 variable updated in the **`atomic`** construct.

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

- At entry to worksharing regions.
- At entry to or exit from a **`master`** region.

2.8.7 `ordered` Construct

Summary

The **`ordered`** construct specifies a structured block in a loop region that 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.

Chapter 2 Directives 75

Syntax

C/C++

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

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

C/C++

Fortran

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

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

Fortran

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 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 execution of all the **ordered** regions belonging to all previous iterations have completed.

For examples of the **ordered** construct, see Section A.24 on page 215.

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.

Syntax

C/C++

The syntax of the ordered construct is as follows:

#pragma omp ordered new-line
    structured-block

C/C++

Fortran

The syntax of the ordered construct is as follows:

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

Fortran

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 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 execution of all the ordered regions belonging to all previous iterations have completed.

For examples of the ordered construct, see Section A.24 on page 215.

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.

76 OpenMP API • Version 3.0 May 2008

- During execution of an iteration of a loop or a loop nest within a loop region, a thread must not execute more than one **`ordered`** region that binds to the same loop region.

C/C++

- A throw executed inside a **`ordered`** region must cause execution to resume within the same **`ordered`** region, and the same thread that threw the exception must catch it.

C/C++

Cross References

- loop construct, see Section 2.5.1 on page 38.
- parallel loop construct, see Section 2.6.1 on page 54.

2.9 Data Environment

This section presents a directive and several clauses for controlling the data environment during the execution of **`parallel`**, **`task`**, and worksharing regions.

- Section 2.9.1 on page 77 describes how the data-sharing attributes of variables referenced in **`parallel`**, **`task`**, and worksharing regions are determined.
- The **`threadprivate`** directive, which is provided to create threadprivate memory, is described in Section 2.9.2 on page 81.
- Clauses that may be specified on directives to control the data-sharing attributes of variables referenced in **`parallel`**, **`task`**, or worksharing constructs are described in Section 2.9.3 on page 85.
- Clauses that may be specified on directives to copy data values from private or threadprivate variables on one thread to the corresponding variables on other threads in the team are described in Section 2.9.4 on page 100.

2.9.1 Data-sharing Attribute Rules

This section describes how the data-sharing attributes of variables referenced in **`parallel`**, **`task`**, and worksharing regions are determined. The following two cases are described separately:

- Section 2.9.1.1 on page 78 describes the data-sharing attribute rules for variables referenced in a construct.

- During execution of an iteration of a loop or a loop nest within a loop region, a thread must not execute more than one **`ordered`** region that binds to the same loop region.

C/C++

- A throw executed inside a **`ordered`** region must cause execution to resume within the same **`ordered`** region, and the same thread that threw the exception must catch it.

C/C++

Cross References

- loop construct, see Section 2.5.1 on page 38.
- parallel loop construct, see Section 2.6.1 on page 54.

2.9 Data Environment

This section presents a directive and several clauses for controlling the data environment during the execution of **`parallel`**, **`task`**, and worksharing regions.

- Section 2.9.1 on page 77 describes how the data-sharing attributes of variables referenced in **`parallel`**, **`task`**, and worksharing regions are determined.
- The **`threadprivate`** directive, which is provided to create threadprivate memory, is described in Section 2.9.2 on page 81.
- Clauses that may be specified on directives to control the data-sharing attributes of variables referenced in **`parallel`**, **`task`**, or worksharing constructs are described in Section 2.9.3 on page 85.
- Clauses that may be specified on directives to copy data values from private or threadprivate variables on one thread to the corresponding variables on other threads in the team are described in Section 2.9.4 on page 100.

2.9.1 Data-sharing Attribute Rules

This section describes how the data-sharing attributes of variables referenced in **`parallel`**, **`task`**, and worksharing regions are determined. The following two cases are described separately:

- Section 2.9.1.1 on page 78 describes the data-sharing attribute rules for variables referenced in a construct.

Chapter 2 Directives 77

- Section 2.9.1.2 on page 80 describes the data-sharing attribute rules for variables referenced in a region, but outside any construct.

2.9.1.1 Data-sharing Attribute Rules for Variables Referenced in a Construct

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

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 data-sharing attributes:

C/C++

- Variables appearing in **`threadprivate`** directives are threadprivate.
- Variables with automatic storage duration that 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(s) in the associated *for-loop(s)* of a **`for`** or **`parallel for`** construct is(are) private.
- Variables with const-qualified type having no mutable member are shared.
- Static variables which are declared in a scope inside the construct are shared.

C/C++

Fortran

- Variables and common blocks appearing in **`threadprivate`** directives are threadprivate.
- The loop iteration variable(s) in the associated *do-loop(s)* of a **`do`** or **`parallel do`** construct is(are) private.
- A loop iteration variable for a sequential loop in a **`parallel`** or **`task`** construct is private in the innermost such construct that encloses the loop.
- **`implied-do`** and **`forall`** indices are private.
- Cray pointees inherit the data-sharing attribute of the storage with which their Cray pointers are associated.

Fortran

- Section 2.9.1.2 on page 80 describes the data-sharing attribute rules for variables referenced in a region, but outside any construct.

2.9.1.1 Data-sharing Attribute Rules for Variables Referenced in a Construct

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

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 data-sharing attributes:

C/C++

- Variables appearing in **`threadprivate`** directives are threadprivate.
- Variables with automatic storage duration that 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(s) in the associated *for-loop(s)* of a **`for`** or **`parallel for`** construct is(are) private.
- Variables with const-qualified type having no mutable member are shared.
- Static variables which are declared in a scope inside the construct are shared.

C/C++

Fortran

- Variables and common blocks appearing in **`threadprivate`** directives are threadprivate.
- The loop iteration variable(s) in the associated *do-loop(s)* of a **`do`** or **`parallel do`** construct is(are) private.
- A loop iteration variable for a sequential loop in a **`parallel`** or **`task`** construct is private in the innermost such construct that encloses the loop.
- **`implied-do`** and **`forall`** indices are private.
- Cray pointees inherit the data-sharing attribute of the storage with which their Cray pointers are associated.

Fortran

78 OpenMP API • Version 3.0 May 2008

Variables with predetermined data-sharing attributes may not be listed in data-sharing attribute clauses, except for the cases listed below. For these exceptions only, listing a predetermined variable in a data-sharing attribute clause is allowed and overrides the variable's predetermined data-sharing attributes.

C/C++

- The loop iteration variable(s) in the associated *for-loop(s)* of a **`for`** or **`parallel for`** construct may be listed in a **`private`** or **`lastprivate`** clause.

C/C++

Fortran

- The loop iteration variable(s) in the associated *do-loop(s)* 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 or a **`task`** construct may be listed in **`private`**, **`firstprivate`**, **`lastprivate`**, **`shared`**, or **`reduction`** clauses on the **`parallel`** or **`task`** construct, and on enclosed constructs, subject to other restrictions.
- Assumed-size arrays may be listed in a **`shared`** clause.

Fortran

Additional restrictions on the variables which may appear in individual clauses are described with each clause in Section 2.9.3 on page 85.

Variables with explicitly determined data-sharing attributes are those that are referenced in a given construct and are listed in a data-sharing attribute clause on the construct.

Variables with implicitly determined data-sharing attributes are those that are referenced in a given construct, do not have predetermined data-sharing attributes, and are not listed in a data-sharing attribute clause on the construct.

Rules for variables with implicitly determined data-sharing attributes are as follows:

- In a **`parallel`** or **`task`** construct, the data-sharing attributes of these variables are determined by the **`default`** clause, if present (see Section 2.9.3.1 on page 86).
- In a **`parallel`** construct, if no **`default`** clause is present, these variables are shared.
- For constructs other than **`task`**, if no **`default`** clause is present, these variables inherit their data-sharing attributes from the enclosing context.
- In a **`task`** construct, if no **`default`** clause is present, a variable that is determined to be shared in all enclosing constructs, up to and including the innermost enclosing parallel construct, is shared.
- In a **`task`** construct, if no **`default`** clause is present, a variable whose data-sharing attribute is not determined by the rule above is firstprivate.

Variables with predetermined data-sharing attributes may not be listed in data-sharing attribute clauses, except for the cases listed below. For these exceptions only, listing a predetermined variable in a data-sharing attribute clause is allowed and overrides the variable's predetermined data-sharing attributes.

C/C++

- The loop iteration variable(s) in the associated *for-loop(s)* of a **`for`** or **`parallel for`** construct may be listed in a **`private`** or **`lastprivate`** clause.

C/C++

Fortran

- The loop iteration variable(s) in the associated *do-loop(s)* 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 or a **`task`** construct may be listed in **`private`**, **`firstprivate`**, **`lastprivate`**, **`shared`**, or **`reduction`** clauses on the **`parallel`** or **`task`** construct, and on enclosed constructs, subject to other restrictions.
- Assumed-size arrays may be listed in a **`shared`** clause.

Fortran

Additional restrictions on the variables which may appear in individual clauses are described with each clause in Section 2.9.3 on page 85.

Variables with explicitly determined data-sharing attributes are those that are referenced in a given construct and are listed in a data-sharing attribute clause on the construct.

Variables with implicitly determined data-sharing attributes are those that are referenced in a given construct, do not have predetermined data-sharing attributes, and are not listed in a data-sharing attribute clause on the construct.

Rules for variables with implicitly determined data-sharing attributes are as follows:

- In a **`parallel`** or **`task`** construct, the data-sharing attributes of these variables are determined by the **`default`** clause, if present (see Section 2.9.3.1 on page 86).
- In a **`parallel`** construct, if no **`default`** clause is present, these variables are shared.
- For constructs other than **`task`**, if no **`default`** clause is present, these variables inherit their data-sharing attributes from the enclosing context.
- In a **`task`** construct, if no **`default`** clause is present, a variable that is determined to be shared in all enclosing constructs, up to and including the innermost enclosing parallel construct, is shared.
- In a **`task`** construct, if no **`default`** clause is present, a variable whose data-sharing attribute is not determined by the rule above is firstprivate.

Chapter 2 Directives 79

Additional restrictions on the variables whose data-sharing attributes cannot be implicitly determined in a **`task`** construct are described in the *Restrictions* section of the **`firstprivate`** clause (Section 2.9.3.4 on page 92).

2.9.1.2 Data-sharing Attribute Rules for Variables Referenced in a Region but not in a Construct

The data-sharing attributes of variables that 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 unless they appear in a **`threadprivate`** directive.
- 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.

C/C++

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.
- Cray pointees inherit the data-sharing attribute of the storage with which their Cray pointers are associated.
- **`implied-do`** indices, **`forall`** indices, and other local variables declared in called routines in the region are private.

Fortran

Additional restrictions on the variables whose data-sharing attributes cannot be implicitly determined in a **`task`** construct are described in the *Restrictions* section of the **`firstprivate`** clause (Section 2.9.3.4 on page 92).

2.9.1.2 Data-sharing Attribute Rules for Variables Referenced in a Region but not in a Construct

The data-sharing attributes of variables that 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 unless they appear in a **`threadprivate`** directive.
- 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.

C/C++

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.
- Cray pointees inherit the data-sharing attribute of the storage with which their Cray pointers are associated.
- **`implied-do`** indices, **`forall`** indices, and other local variables declared in called routines in the region are private.

Fortran

80 OpenMP API • Version 3.0 May 2008

2.9.2 `threadprivate` Directive

Summary

The **`threadprivate`** directive specifies that variables are replicated, with each thread having its own copy.

Syntax

C/C++

The syntax of the **`threadprivate`** directive is as follows:

```
#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.

C/C++

Fortran

The syntax of the **`threadprivate`** directive is as follows:

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

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

Fortran

Description

Each copy of a threadprivate variable 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. The storage of all copies of a threadprivate variable is freed according to how static variables are handled in the base language, but at an unspecified point in the program.

A program in which a thread references another thread's copy of a threadprivate variable is non-conforming.

2.9.2 `threadprivate` Directive

Summary

The **`threadprivate`** directive specifies that variables are replicated, with each thread having its own copy.

Syntax

C/C++

The syntax of the **`threadprivate`** directive is as follows:

```
#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.

C/C++

Fortran

The syntax of the **`threadprivate`** directive is as follows:

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

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

Fortran

Description

Each copy of a threadprivate variable 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. The storage of all copies of a threadprivate variable is freed according to how static variables are handled in the base language, but at an unspecified point in the program.

A program in which a thread references another thread's copy of a threadprivate variable is non-conforming.

Chapter 2 Directives 81

The content of a threadprivate variable can change across a task scheduling point if the executing thread switches to another schedulable task that modifies the variable. For more details on task scheduling, see Section 1.3 on page 11 and Section 2.7 on page 59.

In **`parallel`** regions, references by the master thread will be to the copy of the variable in the thread which encountered the **`parallel`** region.

During the sequential part references will be to the initial thread's copy of the variable. The values of data in the initial thread's copy of a threadprivate variable are guaranteed to persist between any two consecutive references to the variable in the program.

The values of data in the threadprivate variables of non-initial threads are guaranteed to persist between two consecutive active **`parallel`** regions only if all the following conditions hold:

- Neither **`parallel`** region is nested inside another explicit **`parallel`** region.
- The number of threads used to execute both **`parallel`** regions is the same.
- The value of the *dyn-var* internal control variable in the enclosing task region is false at entry to both **`parallel`** regions.

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

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 unspecified.

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

Note – The order in which any constructors for different threadprivate variables of class type are called is unspecified. The order in which any destructors for different threadprivate variables of class type are called is unspecified.

C/C++

Fortran

A variable is 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.

The content of a threadprivate variable can change across a task scheduling point if the executing thread switches to another schedulable task that modifies the variable. For more details on task scheduling, see Section 1.3 on page 11 and Section 2.7 on page 59.

In **`parallel`** regions, references by the master thread will be to the copy of the variable in the thread which encountered the **`parallel`** region.

During the sequential part references will be to the initial thread's copy of the variable. The values of data in the initial thread's copy of a threadprivate variable are guaranteed to persist between any two consecutive references to the variable in the program.

The values of data in the threadprivate variables of non-initial threads are guaranteed to persist between two consecutive active **`parallel`** regions only if all the following conditions hold:

- Neither **`parallel`** region is nested inside another explicit **`parallel`** region.
- The number of threads used to execute both **`parallel`** regions is the same.
- The value of the *dyn-var* internal control variable in the enclosing task region is false at entry to both **`parallel`** regions.

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

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 unspecified.

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

Note – The order in which any constructors for different threadprivate variables of class type are called is unspecified. The order in which any destructors for different threadprivate variables of class type are called is unspecified.

C/C++

Fortran

A variable is 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.

82 OpenMP API • Version 3.0 May 2008

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.

Fortran

For examples of the **threadprivate** directive, see Section A.25 on page 220.

Restrictions

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

- A threadprivate variable must not appear in any clause except the **copyin**, **copyprivate**, **schedule**, **num_threads**, and **if** clauses.
- A program in which an untied task accesses threadprivate storage is non-conforming.

C/C++

- A variable which is part of another variable (as an array or structure element) cannot appear in a **threadprivate** clause unless it is a static data member of a C++ class.

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.

Fortran

For examples of the **threadprivate** directive, see Section A.25 on page 220.

Restrictions

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

- A threadprivate variable must not appear in any clause except the **copyin**, **copyprivate**, **schedule**, **num_threads**, and **if** clauses.
- A program in which an untied task accesses threadprivate storage is non-conforming.

C/C++

- A variable which is part of another variable (as an array or structure element) cannot appear in a **threadprivate** clause unless it is a static data member of a C++ class.

Chapter 2 Directives 83

- 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 static class member variables must appear in the class definition, in the same scope in which the member variables are declared, 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, namespace, or class scope must refer to a variable declaration at file, namespace, or class 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 class type must have:
  - an accessible, unambiguous default constructor in case of default initialization without a given initializer;
  - an accessible, unambiguous constructor accepting the given argument in case of direct initialization;
  - an accessible, unambiguous copy constructor in case of copy initialization with an explicit initializer.

C/C++

Fortran

- A variable which is part of another variable (as an array or structure element) cannot appear in a **threadprivate** clause.
- 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

- 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 static class member variables must appear in the class definition, in the same scope in which the member variables are declared, 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, namespace, or class scope must refer to a variable declaration at file, namespace, or class 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 class type must have:
  - an accessible, unambiguous default constructor in case of default initialization without a given initializer;
  - an accessible, unambiguous constructor accepting the given argument in case of direct initialization;
  - an accessible, unambiguous copy constructor in case of copy initialization with an explicit initializer.

C/C++

Fortran

- A variable which is part of another variable (as an array or structure element) cannot appear in a **threadprivate** clause.
- 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

84 OpenMP API • Version 3.0 May 2008

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 appear 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.

Fortran

Cross References:

- *dyn-var* ICV, see Section 2.3 on page 28.
- number of threads used to execute a **parallel** region, see Section 2.4.1 on page 35.
- **copyin** clause, see Section 2.9.4.1 on page 101.

2.9.3 Data-Sharing Attribute Clauses

Several constructs accept clauses that allow a user to control the data-sharing attributes of variables referenced in the construct. Data-sharing attribute clauses apply only to variables whose names are visible in the construct on which the clause appears.

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 22). 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.

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 appear 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.

Fortran

Cross References:

- *dyn-var* ICV, see Section 2.3 on page 28.
- number of threads used to execute a **parallel** region, see Section 2.4.1 on page 35.
- **copyin** clause, see Section 2.9.4.1 on page 101.

2.9.3 Data-Sharing Attribute Clauses

Several constructs accept clauses that allow a user to control the data-sharing attributes of variables referenced in the construct. Data-sharing attribute clauses apply only to variables whose names are visible in the construct on which the clause appears.

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 22). 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.

Chapter 2 Directives 85

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 unspecified.

C/C++

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.27 on page 227 for examples). When individual members of a common block appear in a **`private`**, **`firstprivate`**, **`lastprivate`**, or **`reduction`** clause of a directive, the storage of the specified variables is no longer associated with the storage of the common block itself (see Section A.32 on page 237 for examples).

Fortran

2.9.3.1 `default` clause

Summary

The default clause allows the user to control the data-sharing attributes of variables that are referenced in a **`parallel`** or **`task`** construct, and whose data-sharing attributes are implicitly determined (see Section 2.9.1.1 on page 78).

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 unspecified.

C/C++

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.27 on page 227 for examples). When individual members of a common block appear in a **`private`**, **`firstprivate`**, **`lastprivate`**, or **`reduction`** clause of a directive, the storage of the specified variables is no longer associated with the storage of the common block itself (see Section A.32 on page 237 for examples).

Fortran

2.9.3.1 `default` clause

Summary

The default clause allows the user to control the data-sharing attributes of variables that are referenced in a **`parallel`** or **`task`** construct, and whose data-sharing attributes are implicitly determined (see Section 2.9.1.1 on page 78).

86 OpenMP API • Version 3.0 May 2008

Syntax

C/C++

The syntax of the **`default`** clause is as follows:

```
default(shared|none)
```

C/C++

Fortran

The syntax of the **`default`** clause is as follows:

```
default(private|firstprivate|shared|none)
```

Fortran

Description

The **`default(shared)`** clause causes all variables referenced in the construct that have implicitly determined data-sharing attributes to be shared.

Fortran

The **`default(firstprivate)`** clause causes all variables in the construct that have implicitly determined data-sharing attributes to be firstprivate.

The **`default(private)`** clause causes all variables referenced in the construct that have implicitly determined data-sharing attributes to be private.

Fortran

The **`default(none)`** clause requires that each variable that is referenced in the construct, and that does not have a predetermined data-sharing attribute, must have its data-sharing attribute explicitly determined by being listed in a data-sharing attribute clause. See Section A.28 on page 229 for examples.

Restrictions

The restrictions to the **`default`** clause are as follows:

- Only a single default clause may be specified on a **`parallel`** or **`task`** directive.

Syntax

C/C++

The syntax of the **`default`** clause is as follows:

```
default(shared|none)
```

C/C++

Fortran

The syntax of the **`default`** clause is as follows:

```
default(private|firstprivate|shared|none)
```

Fortran

Description

The **`default(shared)`** clause causes all variables referenced in the construct that have implicitly determined data-sharing attributes to be shared.

Fortran

The **`default(firstprivate)`** clause causes all variables in the construct that have implicitly determined data-sharing attributes to be firstprivate.

The **`default(private)`** clause causes all variables referenced in the construct that have implicitly determined data-sharing attributes to be private.

Fortran

The **`default(none)`** clause requires that each variable that is referenced in the construct, and that does not have a predetermined data-sharing attribute, must have its data-sharing attribute explicitly determined by being listed in a data-sharing attribute clause. See Section A.28 on page 229 for examples.

Restrictions

The restrictions to the **`default`** clause are as follows:

- Only a single default clause may be specified on a **`parallel`** or **`task`** directive.

Chapter 2 Directives 87

2.9.3.2 `shared` clause

Summary

The **`shared`** clause declares one or more list items to be shared by tasks generated by a **`parallel`** or **`task`** construct.

Syntax

The syntax of the **`shared`** clause is as follows:

```
shared (list)
```

Description

All references to a list item within a task refer to the storage area of the original variable at the point the directive was encountered.

It is the programmer's responsibility to ensure, by adding proper synchronization, that storage shared by an explicit **`task`** region does not reach the end of its lifetime before the explicit **`task`** region completes its execution.

Fortran

The association status of a shared pointer becomes undefined upon entry to and on exit from the **`parallel`** or **`task`** 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 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. It is implementation defined when this situation occurs. See Section A.29 on page 231 for an example of this behavior.

Note – This situation may 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.

2.9.3.2 `shared` clause

Summary

The **`shared`** clause declares one or more list items to be shared by tasks generated by a **`parallel`** or **`task`** construct.

Syntax

The syntax of the **`shared`** clause is as follows:

```
shared (list)
```

Description

All references to a list item within a task refer to the storage area of the original variable at the point the directive was encountered.

It is the programmer's responsibility to ensure, by adding proper synchronization, that storage shared by an explicit **`task`** region does not reach the end of its lifetime before the explicit **`task`** region completes its execution.

Fortran

The association status of a shared pointer becomes undefined upon entry to and on exit from the **`parallel`** or **`task`** 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 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. It is implementation defined when this situation occurs. See Section A.29 on page 231 for an example of this behavior.

Note – This situation may 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.

88 OpenMP API • Version 3.0 May 2008

- 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.
- 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 temporary storage during the procedure reference. Any references to (or definitions of) the shared storage that is associated with the dummy argument by any other task must be synchronized with the procedure reference to avoid possible race conditions.

Fortran

2.9.3.3 `private` clause

Summary

The **`private`** clause declares one or more list items to be private to a task.

Syntax

The syntax of the **`private`** clause is as follows:

```
private(list)
```

Description

Each task 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. In the rest of the region, it is unspecified whether references are to the new list item or the original list item. Therefore, if an

- 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.
- 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 temporary storage during the procedure reference. Any references to (or definitions of) the shared storage that is associated with the dummy argument by any other task must be synchronized with the procedure reference to avoid possible race conditions.

Fortran

2.9.3.3 private clause

Summary

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

Syntax

The syntax of the private clause is as follows:

private(list)

Description

Each task 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. In the rest of the region, it is unspecified whether references are to the new list item or the original list item. Therefore, if an

Chapter 2 Directives 89

attempt is made to reference the original item, its value after the region is also unspecified. If a task does not reference a list item that appears in a **`private`** clause, it is unspecified whether that task receives a new list item.

The value and/or allocation status of the original list item will change only:

- if accessed and modified via pointer,
- if (possibly) accessed in the region but outside of the construct, or
- as a side effect of directives or clauses.

List items that appear in a **`private`**, **`firstprivate`**, or **`reduction`** clause in a **`parallel`** construct may also appear in a **`private`** clause in an enclosed **`parallel`**, **`task`**, or workshare construct. List items that appear in a **`private`** or **`firstprivate`** clause in a **`task`** construct may also appear in a **`private`** clause in an enclosed **`parallel`** or **`task`** construct. List items that appear in a **`private`**, **`firstprivate`**, **`lastprivate`**, or **`reduction`** clause in a workshare construct may also appear in a **`private`** clause in an enclosed **`parallel`** or **`task`** construct. See Section A.31 on page 235 for an example.

C/C++

A new list item of the same type, with automatic storage duration, is allocated for the construct. The storage and thus lifetime of these list items lasts until the block in which they are created exits. The size and alignment of the new list item are determined by the type of the variable. This allocation occurs once for each task generated by the construct, if the task references the list item in any statement.

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 variables of class type are called is unspecified. The order in which any destructors for different private variables of class type are called is unspecified.

C/C++

Fortran

A new list item of the same type is allocated once for each implicit task in the **`parallel`** region, or for each task generated by a **`task`** construct, if the construct references the list item in any statement. The initial value of the new list item is undefined. Within a **`parallel`**, **`worksharing`**, or **`task`** region, the initial status of a **`private`** pointer is undefined.

For a list item with the **`ALLOCATABLE`** attribute:

- if the list item is "not currently allocated", the new list item will have an initial state of "not currently allocated";
- if the list item is allocated, the new list item will have an initial state of allocated with the same array bounds.

attempt is made to reference the original item, its value after the region is also unspecified. If a task does not reference a list item that appears in a **`private`** clause, it is unspecified whether that task receives a new list item.

The value and/or allocation status of the original list item will change only:

- if accessed and modified via pointer,
- if (possibly) accessed in the region but outside of the construct, or
- as a side effect of directives or clauses.

List items that appear in a **`private`**, **`firstprivate`**, or **`reduction`** clause in a **`parallel`** construct may also appear in a **`private`** clause in an enclosed **`parallel`**, **`task`**, or workshare construct. List items that appear in a **`private`** or **`firstprivate`** clause in a **`task`** construct may also appear in a **`private`** clause in an enclosed **`parallel`** or **`task`** construct. List items that appear in a **`private`**, **`firstprivate`**, **`lastprivate`**, or **`reduction`** clause in a workshare construct may also appear in a **`private`** clause in an enclosed **`parallel`** or **`task`** construct. See Section A.31 on page 235 for an example.

C/C++

A new list item of the same type, with automatic storage duration, is allocated for the construct. The storage and thus lifetime of these list items lasts until the block in which they are created exits. The size and alignment of the new list item are determined by the type of the variable. This allocation occurs once for each task generated by the construct, if the task references the list item in any statement.

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 variables of class type are called is unspecified. The order in which any destructors for different private variables of class type are called is unspecified.

C/C++

Fortran

A new list item of the same type is allocated once for each implicit task in the **`parallel`** region, or for each task generated by a **`task`** construct, if the construct references the list item in any statement. The initial value of the new list item is undefined. Within a **`parallel`**, **`worksharing`**, or **`task`** region, the initial status of a **`private`** pointer is undefined.

For a list item with the **`ALLOCATABLE`** attribute:

- if the list item is "not currently allocated", the new list item will have an initial state of "not currently allocated";
- if the list item is allocated, the new list item will have an initial state of allocated with the same array bounds.

90 OpenMP API • Version 3.0 May 2008

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** or **COMMON**. If *A* is a variable appearing in a **private** clause and *B* is a variable that is storage-associated with *A*, then:

- The contents, allocation, and association status of *B* are undefined on entry to the **parallel** or **task** 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.32 on page 237.

Fortran

For examples of the **private** clause, see Section A.30 on page 232.

Restrictions

The restrictions to the **private** clause are as follows:

- A variable which is part of another variable (as an array or structure element) cannot appear in a **private** clause.

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.
- 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

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** or **COMMON**. If *A* is a variable appearing in a **private** clause and *B* is a variable that is storage-associated with *A*, then:

- The contents, allocation, and association status of *B* are undefined on entry to the **parallel** or **task** 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.32 on page 237.

Fortran

For examples of the **private** clause, see Section A.30 on page 232.

Restrictions

The restrictions to the **private** clause are as follows:

- A variable which is part of another variable (as an array or structure element) cannot appear in a **private** clause.

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.
- 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

Chapter 2 Directives 91

2.9.3.4 `firstprivate` clause

Summary

The **`firstprivate`** clause declares one or more list items to be private to a task, 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 2.9.3.3 on page 89. 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 task that references the list item in any statement in the construct. The initialization is done prior to the execution of the construct.

For a **`firstprivate`** clause on a **`parallel`** or **`task`** construct, the initial value of the new list item is the value of the original list item that exists immediately prior to the construct in the task region where the construct is encountered. For a **`firstprivate`** clause on a worksharing construct, the initial value of the new list item for each implicit task of the threads that execute the worksharing construct is the value of the original list item that exists in the implicit task immediately prior to the point in time that the worksharing construct is encountered.

To avoid race conditions, concurrent updates of the original list item must be synchronized with the read of the original list item that occurs as a result of the **`firstprivate`** clause.

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

2.9.3.4 `firstprivate` clause

Summary

The **`firstprivate`** clause declares one or more list items to be private to a task, 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 2.9.3.3 on page 89. 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 task that references the list item in any statement in the construct. The initialization is done prior to the execution of the construct.

For a **`firstprivate`** clause on a **`parallel`** or **`task`** construct, the initial value of the new list item is the value of the original list item that exists immediately prior to the construct in the task region where the construct is encountered. For a **`firstprivate`** clause on a worksharing construct, the initial value of the new list item for each implicit task of the threads that execute the worksharing construct is the value of the original list item that exists in the implicit task immediately prior to the point in time that the worksharing construct is encountered.

To avoid race conditions, concurrent updates of the original list item must be synchronized with the read of the original list item that occurs as a result of the **`firstprivate`** clause.

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

92 OpenMP API • Version 3.0 May 2008

C/C++

For variables of non-array type, the initialization occurs by copy assignment. For a (possibly multi-dimensional) array of elements 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 variables of class type, a copy constructor is invoked to perform the initialization. The order in which copy constructors for different variables of class type are called is unspecified.

C/C++

Fortran

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

Fortran

Restrictions

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

- A variable which is part of another variable (as an array or structure element) cannot appear in a **firstprivate** clause.
- A list item that is private within a **parallel** region must not appear in a **firstprivate** clause on a worksharing construct if any of the worksharing regions arising from the worksharing construct ever bind to any of the **parallel** regions arising from the **parallel** construct.
- A list item that appears in a **reduction** clause of a **parallel** construct must not appear in a **firstprivate** clause on a worksharing or **task** construct if any of the worksharing or **task** regions arising from the worksharing or **task** construct ever bind to any of the **parallel** regions arising from the **parallel** construct.
- A list item that appears in a **reduction** clause in a worksharing construct must not appear in a **firstprivate** clause in a task construct encountered during execution of any of the worksharing regions arising from the worksharing construct.

C/C++

- A variable of class type (or array thereof) that appears in a **firstprivate** clause requires an accessible, unambiguous copy constructor for the class type.
- A variable that appears in a **firstprivate** clause must not have a **const**-qualified type unless it is of class type with a **mutable** member.
- A variable that appears in a **firstprivate** clause must not have an incomplete type or a reference type.

C/C++

Fortran

- A variable that appears in a **firstprivate** clause must be definable.

C/C++

For variables of non-array type, the initialization occurs by copy assignment. For a (possibly multi-dimensional) array of elements 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 variables of class type, a copy constructor is invoked to perform the initialization. The order in which copy constructors for different variables of class type are called is unspecified.

C/C++

Fortran

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

Fortran

Restrictions

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

- A variable which is part of another variable (as an array or structure element) cannot appear in a **firstprivate** clause.
- A list item that is private within a **parallel** region must not appear in a **firstprivate** clause on a worksharing construct if any of the worksharing regions arising from the worksharing construct ever bind to any of the **parallel** regions arising from the **parallel** construct.
- A list item that appears in a **reduction** clause of a **parallel** construct must not appear in a **firstprivate** clause on a worksharing or **task** construct if any of the worksharing or **task** regions arising from the worksharing or **task** construct ever bind to any of the **parallel** regions arising from the **parallel** construct.
- A list item that appears in a **reduction** clause in a worksharing construct must not appear in a **firstprivate** clause in a task construct encountered during execution of any of the worksharing regions arising from the worksharing construct.

C/C++

- A variable of class type (or array thereof) that appears in a **firstprivate** clause requires an accessible, unambiguous copy constructor for the class type.
- A variable that appears in a **firstprivate** clause must not have a **const**-qualified type unless it is of class type with a **mutable** member.
- A variable that appears in a **firstprivate** clause must not have an incomplete type or a reference type.

C/C++

Fortran

- A variable that appears in a **firstprivate** clause must be definable.

Chapter 2 Directives 93

- Fortran pointers, Cray pointers, and assumed-size arrays may not appear in a **`firstprivate`** clause.
- Variables that appear in namelist statements, in variable format expressions, and in expressions for statement function definitions, may not appear in a **`firstprivate`** clause.

Fortran

2.9.3.5 `lastprivate` clause

Summary

The **`lastprivate`** clause declares one or more list items to be private to an implicit task, 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 2.9.3.3 on page 89. In addition, when a **`lastprivate`** clause appears on the directive that identifies a worksharing construct, the value of each new list item from the sequentially last iteration of the associated loops, or the lexically last **`section`** construct, is assigned to the original list item.

C/C++

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

C/C++

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

- Fortran pointers, Cray pointers, and assumed-size arrays may not appear in a **`firstprivate`** clause.
- Variables that appear in namelist statements, in variable format expressions, and in expressions for statement function definitions, may not appear in a **`firstprivate`** clause.

Fortran

2.9.3.5 `lastprivate` clause

Summary

The **`lastprivate`** clause declares one or more list items to be private to an implicit task, 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 2.9.3.3 on page 89. In addition, when a **`lastprivate`** clause appears on the directive that identifies a worksharing construct, the value of each new list item from the sequentially last iteration of the associated loops, or the lexically last **`section`** construct, is assigned to the original list item.

C/C++

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

C/C++

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

94 OpenMP API • Version 3.0 May 2008

The original list item becomes defined at the end of the construct if there is an implicit barrier at that point. To avoid race conditions, concurrent reads or updates of the original list item must be synchronized with the update of the original list item that occurs as a result of the `lastprivate` clause.

If the `lastprivate` clause is used on a construct to which `nowait` is applied, accesses to the original list item may create a data race. To avoid this, synchronization must be inserted to ensure that the sequentially last iteration or lexically last section construct has stored and flushed 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.34 on page 241.

Restrictions

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

- A variable which is part of another variable (as an array or structure element) cannot appear in a `lastprivate` clause.
- 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 worksharing construct if any of the corresponding worksharing regions ever binds to any of the corresponding `parallel` regions.

C/C++

- 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 variables of class type 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.

C/C++

Fortran

- A variable that appears in a `lastprivate` clause must be definable.

The original list item becomes defined at the end of the construct if there is an implicit barrier at that point. To avoid race conditions, concurrent reads or updates of the original list item must be synchronized with the update of the original list item that occurs as a result of the `lastprivate` clause.

If the `lastprivate` clause is used on a construct to which `nowait` is applied, accesses to the original list item may create a data race. To avoid this, synchronization must be inserted to ensure that the sequentially last iteration or lexically last section construct has stored and flushed 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.34 on page 241.

Restrictions

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

- A variable which is part of another variable (as an array or structure element) cannot appear in a `lastprivate` clause.
- 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 worksharing construct if any of the corresponding worksharing regions ever binds to any of the corresponding `parallel` regions.

C/C++

- 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 variables of class type 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.

C/C++

Fortran

- A variable that appears in a `lastprivate` clause must be definable.

Chapter 2 Directives 95

- An original list item with the **`ALLOCATABLE`** attribute must be in the allocated state at entry to the construct containing the lastprivate clause. The list item in the sequentially last iteration or lexically last section must be in the allocated state upon exit from that iteration or section with the same bounds as the corresponding original list item.
- Fortran pointers, Cray pointers, and assumed-size arrays may not appear in a **`lastprivate`** clause.
- Variables that appear in namelist statements, in variable format expressions, and in expressions for statement function definitions, may not appear in a **`lastprivate`** clause.

Fortran

2.9.3.6 `reduction` clause

Summary

The **`reduction`** clause specifies an operator and one or more list items. For each list item, a private copy is created in each implicit task, 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

C/C++

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

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

The following table lists the *operator*s that are valid and their initialization values. The actual initialization value depends on the data type of the reduction list item.

Operator	Initialization value
+	0
*	1
-	0

- An original list item with the **`ALLOCATABLE`** attribute must be in the allocated state at entry to the construct containing the lastprivate clause. The list item in the sequentially last iteration or lexically last section must be in the allocated state upon exit from that iteration or section with the same bounds as the corresponding original list item.
- Fortran pointers, Cray pointers, and assumed-size arrays may not appear in a **`lastprivate`** clause.
- Variables that appear in namelist statements, in variable format expressions, and in expressions for statement function definitions, may not appear in a **`lastprivate`** clause.

Fortran

2.9.3.6 `reduction` clause

Summary

The **`reduction`** clause specifies an operator and one or more list items. For each list item, a private copy is created in each implicit task, 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

C/C++

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

reduction(operator:list)

The following table lists the *operator*s that are valid and their initialization values. The actual initialization value depends on the data type of the reduction list item.

Operator Initialization value
+ 0
* 1
- 0

96 OpenMP API • Version 3.0 May 2008

&	`~0`
|	`0`
^	`0`
&&	**1**
||	`0`

C/C++

Fortran

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 list item.

Operator/ Intrinsic	Initialization value
+	0
*****	**1**
-	0
.and.	**.true.**
.or.	**.false.**
.eqv.	**.true.**
.neqv.	**.false.**
max	Most negative representable number in the reduction list item type
min	Largest representable number in the reduction list item type
iand	All bits on
ior	0
ieor	0

Fortran

&	`~0`
|	`0`
^	`0`
&&	**1**
||	`0`

C/C++

Fortran

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 list item.

Operator/ Intrinsic	Initialization value
+	0
*****	**1**
-	0
.and.	**.true.**
.or.	**.false.**
.eqv.	**.true.**
.neqv.	**.false.**
max	Most negative representable number in the reduction list item type
min	Largest representable number in the reduction list item type
iand	All bits on
ior	0
ieor	0

Fortran

Chapter 2 Directives 97

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 implicit task, 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.)

If **`nowait`** is not used, the reduction 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, accesses to the original list item will create a race and, thus, have unspecified effect unless synchronization ensures that they occur after all threads have executed all of their iterations or **`section`** constructs, and the reduction computation has completed and stored the computed value of that list item. This can most simply be ensured through a barrier synchronization.

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.

To avoid race conditions, concurrent reads or updates of the original list item must be synchronized with the update of the original list item that occurs as a result of the **`reduction`** computation.

Note – List items specified in a **`reduction`** clause are typically used in the enclosed region in certain forms.

C/C++

A reduction is typically specified for statements of the form:

```
x = x op expr
x binop= expr
x = expr op x        (except for subtraction)
x++
++x
x--
--x
```


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 implicit task, 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.)

If **`nowait`** is not used, the reduction 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, accesses to the original list item will create a race and, thus, have unspecified effect unless synchronization ensures that they occur after all threads have executed all of their iterations or **`section`** constructs, and the reduction computation has completed and stored the computed value of that list item. This can most simply be ensured through a barrier synchronization.

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.

To avoid race conditions, concurrent reads or updates of the original list item must be synchronized with the update of the original list item that occurs as a result of the **`reduction`** computation.

Note – List items specified in a **`reduction`** clause are typically used in the enclosed region in certain forms.

C/C++

A reduction is typically specified for statements of the form:

```
x = x op expr
x binop= expr
x = expr op x        (except for subtraction)
x++
++x
x--
--x
```

98 OpenMP API • Version 3.0 May 2008

where *expr* has scalar type and does not reference *x*, *op* is not an overloaded operator, but one of **+**, *****, **-**, **&**, **^**, **|**, **&&**, or **||**, and *binop* is not an overloaded operator, but one of **+**, *****, **-**, **&**, **^**, or **|**.

C/C++

Fortran

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

```
x = x op expr
x = expr op x    (except for subtraction)
```

where *op* is **+**, *****, **-**, **.and.**, **.or.**, **.eqv.**, or **.neqv.**, the expression does not involve *x*, and the reduction *op* is the last operation performed on the right hand side.

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

```
x = intr(x,expr_list)
x = intr(expr_list, x)
```

where *intr* is **max**, **min**, **iand**, **ior**, or **ieor** and *expr_list* is a comma separated list of expressions not involving *x*.

Fortran

For examples, see Section A.35 on page 242.

Restrictions

The restrictions to the **reduction** clause are as follows:

- A list item that appears in a **reduction** clause of a worksharing construct must be shared in the **parallel** regions to which any of the worksharing regions arising from the worksharing construct bind.
- A list item that appears in a **reduction** clause of the innermost enclosing worksharing or **parallel** construct may not be accessed in an explicit task.
- 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.

where *expr* has scalar type and does not reference *x*, *op* is not an overloaded operator, but one of **+**, *****, **-**, **&**, **^**, **|**, **&&**, or **||**, and *binop* is not an overloaded operator, but one of **+**, *****, **-**, **&**, **^**, or **|**.

C/C++

Fortran

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

x = x op expr
x = expr op x   (except for subtraction)

where *op* is **+**, *****, **-**, **.and.**, **.or.**, **.eqv.**, or **.neqv.**, the expression does not involve *x*, and the reduction *op* is the last operation performed on the right hand side.

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

x = intr(x,expr_list)
x = intr(expr_list, x)

where *intr* is **max**, **min**, **iand**, **ior**, or **ieor** and *expr_list* is a comma separated list of expressions not involving *x*.

Fortran

For examples, see Section A.35 on page 242.

Restrictions

The restrictions to the **reduction** clause are as follows:

- A list item that appears in a **reduction** clause of a worksharing construct must be shared in the **parallel** regions to which any of the worksharing regions arising from the worksharing construct bind.
- A list item that appears in a **reduction** clause of the innermost enclosing worksharing or **parallel** construct may not be accessed in an explicit task.
- 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.

Chapter 2 Directives 99

- A list item 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 list items that appear in that clause.

C/C++

Fortran

- The type of a list item that appears in a **`reduction`** clause must be valid for the reduction operator or intrinsic.
- A list item 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.
- An original list item with the **`ALLOCATABLE`** attribute must be in the allocated state at entry to the construct containing the reduction clause. Additionally, the list item must not be deallocated and/or allocated within the region.
- Fortran pointers, Cray pointers and assumed-size 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 list items must be via intrinsic assignment. See Section A.35 on page 242 for examples.

Fortran

2.9.4 Data Copying Clauses

This section describes the **`copyin`** clause (valid on the **`parallel`** directive and combined parallel worksharing directives) and the **`copyprivate`** clause (valid on the **`single`** directive).

These clauses support the copying of data values from private or threadprivate variables on one implicit task or thread to the corresponding variables on other implicit tasks or threads in the team.

The clauses accept a comma-separated list of list items (see Section 2.1 on page 22). 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.

- A list item 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 list items that appear in that clause.

C/C++

Fortran

- The type of a list item that appears in a **`reduction`** clause must be valid for the reduction operator or intrinsic.
- A list item 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.
- An original list item with the **`ALLOCATABLE`** attribute must be in the allocated state at entry to the construct containing the reduction clause. Additionally, the list item must not be deallocated and/or allocated within the region.
- Fortran pointers, Cray pointers and assumed-size 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 list items must be via intrinsic assignment. See Section A.35 on page 242 for examples.

Fortran

2.9.4 Data Copying Clauses

This section describes the **`copyin`** clause (valid on the **`parallel`** directive and combined parallel worksharing directives) and the **`copyprivate`** clause (valid on the **`single`** directive).

These clauses support the copying of data values from private or threadprivate variables on one implicit task or thread to the corresponding variables on other implicit tasks or threads in the team.

The clauses accept a comma-separated list of list items (see Section 2.1 on page 22). 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.

100 OpenMP API • Version 3.0 May 2008

2.9.4.1 `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 associated structured block. For variables of non-array type, the copy occurs by copy assignment. For a (possibly multi-dimensional) array of elements 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 variables of class type are called is unspecified.

C/C++

Fortran

The copy is done, as if by assignment, after the team is formed and prior to the start of execution of the associated structured block.

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.

2.9.4.1 `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 associated structured block. For variables of non-array type, the copy occurs by copy assignment. For a (possibly multi-dimensional) array of elements 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 variables of class type are called is unspecified.

C/C++

Fortran

The copy is done, as if by assignment, after the team is formed and prior to the start of execution of the associated structured block.

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.

Chapter 2 Directives 101

- If it does not have the **`POINTER`** attribute, each copy becomes defined with the value of the master thread's copy as if by intrinsic assignment.

Fortran

For an example of the **`copyin`** clause, see Section A.36 on page 248.

Restrictions

The restrictions to the **`copyin`** clause are as follows:

C/C++

- A list item that appears in a **`copyin`** clause must be threadprivate.
- A variable of class type (or array thereof) that appears in a **`copyin`** clause requires an accessible, unambiguous copy assignment operator for the class type.

C/C++

Fortran

- A list item that appears in a **`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.
- A common block name that appears in a **`copyin`** clause must be declared to be a common block in the same scoping unit in which the **`copyin`** clause appears.
- An array with the **`ALLOCATABLE`** attribute must be in the allocated state. Each thread's copy of that array must be allocated with the same bounds.

Fortran

2.9.4.2 `copyprivate` clause

Summary

The **`copyprivate`** clause provides a mechanism to use a private variable to broadcast a value from the data environment of one implicit task to the data environments of the other implicit tasks belonging to the **`parallel`** region.

To avoid race conditions, concurrent reads or updates of the list item must be synchronized with the update of the list item that occurs as a result of the **`copyprivate`** clause.

- If it does not have the **`POINTER`** attribute, each copy becomes defined with the value of the master thread's copy as if by intrinsic assignment.

Fortran

For an example of the **`copyin`** clause, see Section A.36 on page 248.

Restrictions

The restrictions to the **`copyin`** clause are as follows:

C/C++

- A list item that appears in a **`copyin`** clause must be threadprivate.
- A variable of class type (or array thereof) that appears in a **`copyin`** clause requires an accessible, unambiguous copy assignment operator for the class type.

C/C++

Fortran

- A list item that appears in a **`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.
- A common block name that appears in a **`copyin`** clause must be declared to be a common block in the same scoping unit in which the **`copyin`** clause appears.
- An array with the **`ALLOCATABLE`** attribute must be in the allocated state. Each thread's copy of that array must be allocated with the same bounds.

Fortran

2.9.4.2 `copyprivate` clause

Summary

The **`copyprivate`** clause provides a mechanism to use a private variable to broadcast a value from the data environment of one implicit task to the data environments of the other implicit tasks belonging to the **`parallel`** region.

To avoid race conditions, concurrent reads or updates of the list item must be synchronized with the update of the list item that occurs as a result of the **`copyprivate`** clause.

102 OpenMP API • Version 3.0 May 2008

Syntax

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

```
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 49), and before any of the threads in the team have left the barrier at the end of the construct.

C/C++

In all other implicit tasks belonging to the **`parallel`** region, each specified list item becomes defined with the value of the corresponding list item in the implicit task whose thread executed the structured block. For variables of non-array type, the definition occurs by copy assignment. For a (possibly multi-dimensional) array of elements of non-array type, each element is copied by copy assignment from an element of the array in the data environment of the implicit task whose thread executed the structured block to the corresponding element of the array in the data environment of the other implicit tasks. For class types, a copy assignment operator is invoked. The order in which copy assignment operators for different variables of class type are called is unspecified.

C/C++

Fortran

If a list item is not a pointer, then in all other implicit tasks belonging to the **`parallel`** region, the list item becomes defined (as if by assignment) with the value of the corresponding list item in the implicit task whose thread executed the structured block. If the list item is a pointer, then in all other implicit tasks belonging to the **`parallel`** region, the list item becomes pointer associated (as if by pointer assignment) with the corresponding list item in the implicit task whose thread executed the structured block.

Fortran

For examples of the **`copyprivate`** clause, see Section A.37 on page 250.

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).

Syntax

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

```
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 49), and before any of the threads in the team have left the barrier at the end of the construct.

C/C++

In all other implicit tasks belonging to the **`parallel`** region, each specified list item becomes defined with the value of the corresponding list item in the implicit task whose thread executed the structured block. For variables of non-array type, the definition occurs by copy assignment. For a (possibly multi-dimensional) array of elements of non-array type, each element is copied by copy assignment from an element of the array in the data environment of the implicit task whose thread executed the structured block to the corresponding element of the array in the data environment of the other implicit tasks. For class types, a copy assignment operator is invoked. The order in which copy assignment operators for different variables of class type are called is unspecified.

C/C++

Fortran

If a list item is not a pointer, then in all other implicit tasks belonging to the **`parallel`** region, the list item becomes defined (as if by assignment) with the value of the corresponding list item in the implicit task whose thread executed the structured block. If the list item is a pointer, then in all other implicit tasks belonging to the **`parallel`** region, the list item becomes pointer associated (as if by pointer assignment) with the corresponding list item in the implicit task whose thread executed the structured block.

Fortran

For examples of the **`copyprivate`** clause, see Section A.37 on page 250.

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).

Chapter 2 Directives **103**

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.

C/C++

Fortran

- A common block that appears in a **`copyprivate`** clause must be threadprivate.
- Assumed-size arrays may not appear in a **`copyprivate`** clause.
- An array with the **`ALLOCATABLE`** attribute must be in the allocated state with the same bounds in all threads affected by the **`copyprivate`** clause.

Fortran

2.10 Nesting of Regions

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

- A worksharing region may not be closely nested inside a worksharing, explicit **`task`**, **`critical`**, **`ordered`**, or **`master`** region.
- A **`barrier`** region may not be closely nested inside a worksharing, explicit **`task`**, **`critical`**, **`ordered`**, or **`master`** region.
- A **`master`** region may not be closely nested inside a worksharing or explicit **`task`** region.
- An **`ordered`** region may not be closely nested inside a **`critical`** or explicit **`task`** 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.

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.

C/C++

Fortran

- A common block that appears in a **`copyprivate`** clause must be threadprivate.
- Assumed-size arrays may not appear in a **`copyprivate`** clause.
- An array with the **`ALLOCATABLE`** attribute must be in the allocated state with the same bounds in all threads affected by the **`copyprivate`** clause.

Fortran

2.10 Nesting of Regions

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

- A worksharing region may not be closely nested inside a worksharing, explicit **`task`**, **`critical`**, **`ordered`**, or **`master`** region.
- A **`barrier`** region may not be closely nested inside a worksharing, explicit **`task`**, **`critical`**, **`ordered`**, or **`master`** region.
- A **`master`** region may not be closely nested inside a worksharing or explicit **`task`** region.
- An **`ordered`** region may not be closely nested inside a **`critical`** or explicit **`task`** 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.

104 OpenMP API • Version 3.0 May 2008

For examples illustrating these rules, see Section A.17 on page 199, Section A.38 on page 255, Section A.39 on page 258, and Section A.13 on page 177.

For examples illustrating these rules, see Section A.17 on page 199, Section A.38 on page 255, Section A.39 on page 258, and Section A.13 on page 177.

Chapter 2 Directives 105

106 OpenMP API • Version 3.0 May 2008

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 108).
- Execution environment routines that can be used to control and query the parallel execution environment (Section 3.2 on page 109).
- Lock routines that can be used to synchronize access to data (Section 3.3 on page 134).
- Portable timer routines (Section 3.4 on page 142).

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.

C/C++

Fortran

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

Fortran

Fortran

Restrictions

The following restriction applies to all OpenMP runtime library routines:

- OpenMP runtime library routines may not be called from **PURE** or **ELEMENTAL** procedures.

Fortran

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 108).
- Execution environment routines that can be used to control and query the parallel execution environment (Section 3.2 on page 109).
- Lock routines that can be used to synchronize access to data (Section 3.3 on page 134).
- Portable timer routines (Section 3.4 on page 142).

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.

C/C++

Fortran

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

Fortran

Fortran

Restrictions

The following restriction applies to all OpenMP runtime library routines:

- OpenMP runtime library routines may not be called from **PURE** or **ELEMENTAL** procedures.

Fortran

107

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*, *nestable lock* and *schedule* 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`**.
- The type **`omp_sched_t.`**

See Section D.1 on page 302 for an example of this file.

C/C++

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 omp_sched_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 26).

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*, *nestable lock* and *schedule* 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`**.
- The type **`omp_sched_t.`**

See Section D.1 on page 302 for an example of this file.

C/C++

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 omp_sched_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 26).

108 OpenMP API • Version 3.0 May 2008

See Section D.2 on page 304 and Section D.3 on page 306 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.

Fortran

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.
- the **`omp_set_schedule`** routine.
- the **`omp_get_schedule`** routine.
- the **`omp_get_thread_limit`** routine.
- the **`omp_set_max_active_levels`** routine.
- the **`omp_get_max_active_levels`** routine.
- the **`omp_get_level`** routine.
- the **`omp_get_ancestor_thread_num`** routine.
- the **`omp_get_team_size`** routine.
- the **`omp_get_active_level`** routine.

See Section D.2 on page 304 and Section D.3 on page 306 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.

Fortran

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.
- the **`omp_set_schedule`** routine.
- the **`omp_get_schedule`** routine.
- the **`omp_get_thread_limit`** routine.
- the **`omp_set_max_active_levels`** routine.
- the **`omp_get_max_active_levels`** routine.
- the **`omp_get_level`** routine.
- the **`omp_get_ancestor_thread_num`** routine.
- the **`omp_get_team_size`** routine.
- the **`omp_get_active_level`** routine.

Chapter 3 Runtime Library Routines 109

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* ICV.

Format

C/C++

```
void omp_set_num_threads(int num_threads);
```

C/C++

Fortran

```
subroutine omp_set_num_threads(num_threads)
integer num_threads
```

Fortran

Constraints on Arguments

The value of the argument passed to this routine must evaluate to a positive integer, or else the behavior of this routine is implementation defined.

Binding

The binding task set for an **omp_set_num_threads** region is the generating task.

Effect

The effect of this routine is to set the value of the *nthreads-var* ICV to the value specified in the argument.

See Section 2.4.1 on page 35 for the rules governing the number of threads used to execute a **parallel** region.

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* ICV.

Format

C/C++

```
void omp_set_num_threads(int num_threads);
```

C/C++

Fortran

```
subroutine omp_set_num_threads(num_threads)
integer num_threads
```

Fortran

Constraints on Arguments

The value of the argument passed to this routine must evaluate to a positive integer, or else the behavior of this routine is implementation defined.

Binding

The binding task set for an **omp_set_num_threads** region is the generating task.

Effect

The effect of this routine is to set the value of the *nthreads-var* ICV to the value specified in the argument.

See Section 2.4.1 on page 35 for the rules governing the number of threads used to execute a **parallel** region.

110 OpenMP API • Version 3.0 May 2008

For an example of the **omp_set_num_threads** routine, see Section A.40 on page 265.

Cross References

- *nthreads-var* ICV, see Section 2.3 on page 28.
- **OMP_NUM_THREADS** environment variable, see Section 4.2 on page 147.
- **omp_get_max_threads** routine, see Section 3.2.3 on page 112.
- **parallel** construct, see Section 2.4 on page 32.
- **num_threads** clause, see Section 2.4 on page 32.

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/C++

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

C/C++

Fortran

```
integer function omp_get_num_threads()
```

Fortran

Binding

The binding region for an **omp_get_num_threads** region is the innermost enclosing **parallel** region.

For an example of the **omp_set_num_threads** routine, see Section A.40 on page 265.

Cross References

- *nthreads-var* ICV, see Section 2.3 on page 28.
- **OMP_NUM_THREADS** environment variable, see Section 4.2 on page 147.
- **omp_get_max_threads** routine, see Section 3.2.3 on page 112.
- **parallel** construct, see Section 2.4 on page 32.
- **num_threads** clause, see Section 2.4 on page 32.

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/C++

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

C/C++

Fortran

```
integer function omp_get_num_threads()
```

Fortran

Binding

The binding region for an **omp_get_num_threads** region is the innermost enclosing **parallel** region.

Chapter 3 Runtime Library Routines 111

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.41 on page 266.

See Section 2.4.1 on page 35 for the rules governing the number of threads used to execute a **parallel** region.

Cross References

- **parallel** construct, see Section 2.4 on page 32.
- **omp_set_num_threads** routine, see Section 3.2.1 on page 110.
- **OMP_NUM_THREADS** environment variable, see Section 4.2 on page 147.

3.2.3 omp_get_max_threads

Summary

The **omp_get_max_threads** routine returns an upper bound on the number of threads that could be used to form a new team if a **parallel** region without a **num_threads** clause were encountered after execution returns from this routine.

Format

C/C++

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

C/C++

Fortran

```
integer function omp_get_max_threads()
```

Fortran

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.41 on page 266.

See Section 2.4.1 on page 35 for the rules governing the number of threads used to execute a **parallel** region.

Cross References

- **parallel** construct, see Section 2.4 on page 32.
- **omp_set_num_threads** routine, see Section 3.2.1 on page 110.
- **OMP_NUM_THREADS** environment variable, see Section 4.2 on page 147.

3.2.3 omp_get_max_threads

Summary

The **omp_get_max_threads** routine returns an upper bound on the number of threads that could be used to form a new team if a **parallel** region without a **num_threads** clause were encountered after execution returns from this routine.

Format

C/C++

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

C/C++

Fortran

```
integer function omp_get_max_threads()
```

Fortran

112 OpenMP API • Version 3.0 May 2008

Binding

The binding task set for an **`omp_get_max_threads`** region is the generating task.

Effect

The value returned by **`omp_get_max_threads`** is the value of the *nthreads-var* ICV. This value is also an upper bound on the number of threads that could be used to form a new team if a parallel region without a **`num_threads`** clause were encountered after execution returns from this routine.

See Section 2.4.1 on page 35 for the rules governing the number of threads used to execute a **`parallel`** region.

Note – The return value of the **`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

- *nthreads-var* ICV, see Section 2.3 on page 28.
- **`parallel`** construct, see Section 2.4 on page 32.
- **`num_threads`** clause, see Section 2.4 on page 32.
- **`omp_set_num_threads`** routine, see Section 3.2.1 on page 110.
- **`OMP_NUM_THREADS`** environment variable, see Section 4.2 on page 147.

3.2.4 `omp_get_thread_num`

Summary

The **`omp_get_thread_num`** routine returns the thread number, within the current team, of the thread executing the implicit or explicit **`task`** region from which **`omp_get_thread_num`** is called.

Binding

The binding task set for an **`omp_get_max_threads`** region is the generating task.

Effect

The value returned by **`omp_get_max_threads`** is the value of the *nthreads-var* ICV. This value is also an upper bound on the number of threads that could be used to form a new team if a parallel region without a **`num_threads`** clause were encountered after execution returns from this routine.

See Section 2.4.1 on page 35 for the rules governing the number of threads used to execute a **`parallel`** region.

Note – The return value of the **`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

- *nthreads-var* ICV, see Section 2.3 on page 28.
- **`parallel`** construct, see Section 2.4 on page 32.
- **`num_threads`** clause, see Section 2.4 on page 32.
- **`omp_set_num_threads`** routine, see Section 3.2.1 on page 110.
- **`OMP_NUM_THREADS`** environment variable, see Section 4.2 on page 147.

3.2.4 `omp_get_thread_num`

Summary

The **`omp_get_thread_num`** routine returns the thread number, within the current team, of the thread executing the implicit or explicit **`task`** region from which **`omp_get_thread_num`** is called.

Chapter 3 Runtime Library Routines 113

Format

C/C++

```
int omp_get_thread_num(void);
```

C/C++

Fortran

```
integer function omp_get_thread_num()
```

Fortran

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.

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.

Note – The thread number may change at any time during the execution of an untied task. The value returned by **omp_get_thread_num** is not generally useful during the execution of such a task region.

Cross References

- **omp_get_num_threads** routine, see Section 3.2.2 on page 111.

Format

C/C++

```
int omp_get_thread_num(void);
```

C/C++

Fortran

```
integer function omp_get_thread_num()
```

Fortran

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.

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.

Note – The thread number may change at any time during the execution of an untied task. The value returned by **omp_get_thread_num** is not generally useful during the execution of such a task region.

Cross References

- **omp_get_num_threads** routine, see Section 3.2.2 on page 111.

114 OpenMP API • Version 3.0 May 2008

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/C++

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

C/C++

Fortran

```
integer function omp_get_num_procs()
```

Fortran

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. Note that this value may change between the time that it is determined by the **`omp_get_num_procs`** routine and the time that it is read in the calling context due to system actions outside the control of the OpenMP implementation.

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/C++

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

C/C++

Fortran

```
integer function omp_get_num_procs()
```

Fortran

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. Note that this value may change between the time that it is determined by the **`omp_get_num_procs`** routine and the time that it is read in the calling context due to system actions outside the control of the OpenMP implementation.

Chapter 3 Runtime Library Routines 115

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*.

Format

C/C++

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

C/C++

Fortran

```
logical function omp_in_parallel()
```

Fortran

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 *true* if any enclosing **parallel** region is active. If the routine call is enclosed by only inactive **parallel** regions (including the implicit parallel region), then it returns *false*.

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*.

Format

C/C++

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

C/C++

Fortran

```
logical function omp_in_parallel()
```

Fortran

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 *true* if any enclosing **parallel** region is active. If the routine call is enclosed by only inactive **parallel** regions (including the implicit parallel region), then it returns *false*.

116 OpenMP API • Version 3.0 May 2008

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 subsequent **parallel** regions by setting the value of the *dyn-var* ICV.

Format

C/C++

```
void omp_set_dynamic(int dynamic_threads);
```

C/C++

Fortran

```
subroutine omp_set_dynamic (dynamic_threads)
logical dynamic_threads
```

Fortran

Binding

The binding task set for an **omp_set_dynamic** region is the generating task.

Effect

For implementations that support dynamic adjustment of the number of threads, if the argument to **omp_set_dynamic** evaluates to *true*, dynamic adjustment is enabled; otherwise, dynamic adjustment is disabled. For implementations that do not support dynamic adjustment of 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.40 on page 265.

See Section 2.4.1 on page 35 for the rules governing the number of threads used to execute a **parallel** region.

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 subsequent **parallel** regions by setting the value of the *dyn-var* ICV.

Format

C/C++

```
void omp_set_dynamic(int dynamic_threads);
```

C/C++

Fortran

```
subroutine omp_set_dynamic (dynamic_threads)
logical dynamic_threads
```

Fortran

Binding

The binding task set for an **omp_set_dynamic** region is the generating task.

Effect

For implementations that support dynamic adjustment of the number of threads, if the argument to **omp_set_dynamic** evaluates to *true*, dynamic adjustment is enabled; otherwise, dynamic adjustment is disabled. For implementations that do not support dynamic adjustment of 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.40 on page 265.

See Section 2.4.1 on page 35 for the rules governing the number of threads used to execute a **parallel** region.

Chapter 3 Runtime Library Routines 117

Cross References:

- *dyn-var* ICV, see Section 2.3 on page 28.
- **`omp_get_num_threads`** routine, see Section 3.2.2 on page 111.
- **`omp_get_dynamic`** routine, see Section 3.2.8 on page 118.
- **`OMP_DYNAMIC`** environment variable, see Section 4.3 on page 148.

3.2.8 `omp_get_dynamic`

Summary

The **`omp_get_dynamic`** routine returns the value of the *dyn-var* ICV, which determines whether dynamic adjustment of the number of threads is enabled or disabled.

Format

C/C++

```
int omp_get_dynamic(void);
```

C/C++

Fortran

```
logical function omp_get_dynamic()
```

Fortran

Binding

The binding task set for an **`omp_get_dynamic`** region is the generating task.

Effect

This routine returns *true* if dynamic adjustment of the number of threads is enabled; it returns *false*, otherwise. If an implementation does not support dynamic adjustment of the number of threads, then this routine always returns *false*.

Cross References:

- *dyn-var* ICV, see Section 2.3 on page 28.
- **`omp_get_num_threads`** routine, see Section 3.2.2 on page 111.
- **`omp_get_dynamic`** routine, see Section 3.2.8 on page 118.
- **`OMP_DYNAMIC`** environment variable, see Section 4.3 on page 148.

3.2.8 `omp_get_dynamic`

Summary

The **`omp_get_dynamic`** routine returns the value of the *dyn-var* ICV, which determines whether dynamic adjustment of the number of threads is enabled or disabled.

Format

C/C++

```
int omp_get_dynamic(void);
```

C/C++

Fortran

```
logical function omp_get_dynamic()
```

Fortran

Binding

The binding task set for an **`omp_get_dynamic`** region is the generating task.

Effect

This routine returns *true* if dynamic adjustment of the number of threads is enabled; it returns *false*, otherwise. If an implementation does not support dynamic adjustment of the number of threads, then this routine always returns *false*.

118 OpenMP API • Version 3.0 May 2008

See Section 2.4.1 on page 35 for the rules governing the number of threads used to execute a **parallel** region.

Cross References

- *dyn-var* ICV, see Section 2.3 on page 28.
- **omp_set_dynamic** routine, see Section 3.2.7 on page 117.
- **OMP_DYNAMIC** environment variable, see Section 4.3 on page 148.

3.2.9 omp_set_nested

Summary

The **omp_set_nested** routine enables or disables nested parallelism, by setting the *nest-var* ICV.

Format

C/C++

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

C/C++

Fortran

```
subroutine omp_set_nested (nested)
logical nested
```

Fortran

Binding

The binding task set for an **omp_set_nested** region is the generating task.

See Section 2.4.1 on page 35 for the rules governing the number of threads used to execute a **parallel** region.

Cross References

- *dyn-var* ICV, see Section 2.3 on page 28.
- **omp_set_dynamic** routine, see Section 3.2.7 on page 117.
- **OMP_DYNAMIC** environment variable, see Section 4.3 on page 148.

3.2.9 omp_set_nested

Summary

The **omp_set_nested** routine enables or disables nested parallelism, by setting the *nest-var* ICV.

Format

C/C++

void omp_set_nested(int nested);

C/C++

Fortran

subroutine omp_set_nested (nested)
logical nested

Fortran

Binding

The binding task set for an **omp_set_nested** region is the generating task.

Chapter 3 Runtime Library Routines 119

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 35 for the rules governing the number of threads used to execute a **`parallel`** region.

Cross References

- *nest-var* ICV, see Section 2.3 on page 28.
- **`omp_set_max_active_levels`** routine, see Section 3.2.14 on page 126.
- **`omp_get_max_active_levels`** routine, see Section 3.2.15 on page 127.
- **`omp_get_nested`** routine, see Section 3.2.10 on page 120.
- **`OMP_NESTED`** environment variable, see Section 4.4 on page 148.

3.2.10 omp_get_nested

Summary

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

Format

C/C++

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

C/C++

Fortran

```
logical function omp_get_nested()
```

Fortran

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 35 for the rules governing the number of threads used to execute a parallel region.

Cross References

- nest-var ICV, see Section 2.3 on page 28.
- omp_set_max_active_levels routine, see Section 3.2.14 on page 126.
- omp_get_max_active_levels routine, see Section 3.2.15 on page 127.
- omp_get_nested routine, see Section 3.2.10 on page 120.
- OMP_NESTED environment variable, see Section 4.4 on page 148.

3.2.10 omp_get_nested

Summary

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

Format

C/C++

int omp_get_nested(void);

C/C++

Fortran

logical function omp_get_nested()

Fortran

120 OpenMP API • Version 3.0 May 2008

Binding

The binding task set for an **`omp_get_nested`** region is the generating task.

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 35 for the rules governing the number of threads used to execute a **`parallel`** region.

Cross References

- *nest-var* ICV, see Section 2.3 on page 28.
- **`omp_set_nested`** routine, see Section 3.2.9 on page 119.
- **`OMP_NESTED`** environment variable, see Section 4.4 on page 148.

3.2.11 omp_set_schedule

Summary

The **`omp_set_schedule`** routine affects the schedule that is applied when **`runtime`** is used as schedule kind, by setting the value of the *run-sched-var* ICV.

Format

C/C++

```
void omp_set_schedule(omp_sched_t kind, int modifier);
```

C/C++

Binding

The binding task set for an **`omp_get_nested`** region is the generating task.

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 35 for the rules governing the number of threads used to execute a **`parallel`** region.

Cross References

- *nest-var* ICV, see Section 2.3 on page 28.
- **`omp_set_nested`** routine, see Section 3.2.9 on page 119.
- **`OMP_NESTED`** environment variable, see Section 4.4 on page 148.

3.2.11 omp_set_schedule

Summary

The **`omp_set_schedule`** routine affects the schedule that is applied when **`runtime`** is used as schedule kind, by setting the value of the *run-sched-var* ICV.

Format

C/C++

```
void omp_set_schedule(omp_sched_t kind, int modifier);
```

C/C++

Chapter 3 Runtime Library Routines 121

Fortran

```
subroutine omp_set_schedule(kind, modifier)
integer (kind=omp_sched_kind) kind
integer modifier
```

Fortran

Constraints on Arguments

The first argument passed to this routine can be one of the valid OpenMP schedule kinds (except for **runtime**) or any implementation specific schedule. The C/C++ header file (**omp.h**) and the Fortran include file (**omp_lib.h**) and/or Fortran 90 module file (**omp_lib**) define the valid constants. The valid constants must include the following, which can be extended with implementation specific values:

C/C++

```
typedef enum omp_sched_t {
    omp_sched_static = 1,
    omp_sched_dynamic = 2,
    omp_sched_guided = 3,
    omp_sched_auto = 4
} omp_sched_t;
```

C/C++

Fortran

```
integer(kind=omp_sched_kind), parameter :: omp_sched_static = 1
integer(kind=omp_sched_kind), parameter :: omp_sched_dynamic = 2
integer(kind=omp_sched_kind), parameter :: omp_sched_guided = 3
integer(kind=omp_sched_kind), parameter :: omp_sched_auto = 4
```

Fortran

Fortran

```
subroutine omp_set_schedule(kind, modifier)
integer (kind=omp_sched_kind) kind
integer modifier
```

Fortran

Constraints on Arguments

The first argument passed to this routine can be one of the valid OpenMP schedule kinds (except for **runtime**) or any implementation specific schedule. The C/C++ header file (**omp.h**) and the Fortran include file (**omp_lib.h**) and/or Fortran 90 module file (**omp_lib**) define the valid constants. The valid constants must include the following, which can be extended with implementation specific values:

C/C++

```
typedef enum omp_sched_t {
    omp_sched_static = 1,
    omp_sched_dynamic = 2,
    omp_sched_guided = 3,
    omp_sched_auto = 4
} omp_sched_t;
```

C/C++

Fortran

```
integer(kind=omp_sched_kind), parameter :: omp_sched_static = 1
integer(kind=omp_sched_kind), parameter :: omp_sched_dynamic = 2
integer(kind=omp_sched_kind), parameter :: omp_sched_guided = 3
integer(kind=omp_sched_kind), parameter :: omp_sched_auto = 4
```

Fortran

122 OpenMP API • Version 3.0 May 2008

Binding

The binding task set for an **`omp_set_schedule`** region is the generating task.

Effect

The effect of this routine is to set the value of the *run-sched-var* ICV to the values specified in the two arguments. The schedule is set to the schedule type specified by the first argument **`kind`**. It can be any of the standard schedule types or any other implementation specific one. For the schedule types **`static`**, **`dynamic`**, and **`guided`** the *chunk_size* is set to the value of the second argument, or to the default *chunk_size* if the value of the second argument is less than 1; for the schedule type **`auto`** the second argument has no meaning; for implementation specific schedule types, the values and associated meanings of the second argument are implementation defined.

Cross References

- *run-sched-var* ICV, see Section 2.3 on page 28.
- **`omp_get_schedule`** routine, see Section 3.2.12 on page 123.
- **`OMP_SCHEDULE`** environment variable, see Section 4.1 on page 146.
- Determining the schedule of a worksharing loop, see Section 2.5.1.1 on page 45.

3.2.12 `omp_get_schedule`

Summary

The **`omp_get_schedule`** routine returns the schedule that is applied when the **`runtime`** schedule is used.

Binding

The binding task set for an **`omp_set_schedule`** region is the generating task.

Effect

The effect of this routine is to set the value of the *run-sched-var* ICV to the values specified in the two arguments. The schedule is set to the schedule type specified by the first argument **`kind`**. It can be any of the standard schedule types or any other implementation specific one. For the schedule types **`static`**, **`dynamic`**, and **`guided`** the *chunk_size* is set to the value of the second argument, or to the default *chunk_size* if the value of the second argument is less than 1; for the schedule type **`auto`** the second argument has no meaning; for implementation specific schedule types, the values and associated meanings of the second argument are implementation defined.

Cross References

- *run-sched-var* ICV, see Section 2.3 on page 28.
- **`omp_get_schedule`** routine, see Section 3.2.12 on page 123.
- **`OMP_SCHEDULE`** environment variable, see Section 4.1 on page 146.
- Determining the schedule of a worksharing loop, see Section 2.5.1.1 on page 45.

3.2.12 `omp_get_schedule`

Summary

The **`omp_get_schedule`** routine returns the schedule that is applied when the **`runtime`** schedule is used.

Chapter 3 Runtime Library Routines 123

Format

C/C++

```
void omp_get_schedule(omp_sched_t * kind, int * modifier );
```

C/C++

Fortran

```
subroutine omp_get_schedule(kind, modifier)
integer (kind=omp_sched_kind) kind
integer modifier
```

Fortran

Binding

The binding task set for an **`omp_get_schedule`** region is the generating task.

Effect

This routine returns the *run-sched-var* ICV in the team executing the **`parallel`** region to which the routine binds. The first argument **`kind`** returns the schedule to be used. It can be any of the standard schedule types as defined in Section 3.2.11 on page 121, or any implementation specific schedule type. The second argument is interpreted as in the **`omp_set_schedule`** call, defined in Section 3.2.11 on page 121.

Cross References

- *run-sched-var* ICV, see Section 2.3 on page 28.
- **`omp_set_schedule`** routine, see Section 3.2.11 on page 121.
- **`OMP_SCHEDULE`** environment variable, see Section 4.1 on page 146.
- Determining the schedule of a worksharing loop, see Section 2.5.1.1 on page 45.

Format

C/C++

```
void omp_get_schedule(omp_sched_t * kind, int * modifier );
```

C/C++

Fortran

```
subroutine omp_get_schedule(kind, modifier)
integer (kind=omp_sched_kind) kind
integer modifier
```

Fortran

Binding

The binding task set for an **`omp_get_schedule`** region is the generating task.

Effect

This routine returns the *run-sched-var* ICV in the team executing the **`parallel`** region to which the routine binds. The first argument **`kind`** returns the schedule to be used. It can be any of the standard schedule types as defined in Section 3.2.11 on page 121, or any implementation specific schedule type. The second argument is interpreted as in the **`omp_set_schedule`** call, defined in Section 3.2.11 on page 121.

Cross References

- *run-sched-var* ICV, see Section 2.3 on page 28.
- **`omp_set_schedule`** routine, see Section 3.2.11 on page 121.
- **`OMP_SCHEDULE`** environment variable, see Section 4.1 on page 146.
- Determining the schedule of a worksharing loop, see Section 2.5.1.1 on page 45.

124 OpenMP API • Version 3.0 May 2008

3.2.13 `omp_get_thread_limit`

Summary

The **`omp_get_thread_limit`** routine returns the maximum number of OpenMP threads available to the program.

Format

C/C++

```
int omp_get_thread_limit(void)
```

C/C++

Fortran

```
integer function omp_get_thread_limit()
```

Fortran

Binding

The binding thread set for an **`omp_get_thread_limit`** 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_thread_limit`** routine returns the maximum number of OpenMP threads available to the program as stored in the ICV *thread-limit-var*.

3.2.13 `omp_get_thread_limit`

Summary

The **`omp_get_thread_limit`** routine returns the maximum number of OpenMP threads available to the program.

Format

C/C++

```
int omp_get_thread_limit(void)
```

C/C++

Fortran

```
integer function omp_get_thread_limit()
```

Fortran

Binding

The binding thread set for an **`omp_get_thread_limit`** 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_thread_limit`** routine returns the maximum number of OpenMP threads available to the program as stored in the ICV *thread-limit-var*.

Chapter 3 Runtime Library Routines 125

Cross References

- *thread-limit-var* ICV, see Section 2.3 on page 28.
- **`OMP_THREAD_LIMIT`** environment variable, see Section 4.8 on page 151.

3.2.14 omp_set_max_active_levels

Summary

The **`omp_set_max_active_levels`** routine limits the number of nested active parallel regions, by setting the *max-active-levels-var* ICV.

Format

C/C++

```
void omp_set_max_active_levels (int max_levels)
```

C/C++

Fortran

```
subroutine omp_set_max_active_levels (max_levels)
integer max_levels
```

Fortran

Constraints on Arguments

The value of the argument passed to this routine must evaluate to a non-negative integer, or else the behavior of this routine is implementation defined.

Cross References

- *thread-limit-var* ICV, see Section 2.3 on page 28.
- **`OMP_THREAD_LIMIT`** environment variable, see Section 4.8 on page 151.

3.2.14 omp_set_max_active_levels

Summary

The **`omp_set_max_active_levels`** routine limits the number of nested active parallel regions, by setting the *max-active-levels-var* ICV.

Format

C/C++

```
void omp_set_max_active_levels (int max_levels)
```

C/C++

Fortran

```
subroutine omp_set_max_active_levels (max_levels)
integer max_levels
```

Fortran

Constraints on Arguments

The value of the argument passed to this routine must evaluate to a non-negative integer, or else the behavior of this routine is implementation defined.

126 OpenMP API • Version 3.0 May 2008

Binding

When called from the sequential part of the program, the binding thread set for an **`omp_set_max_active_levels`** 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_max_active_levels`** region is implementation defined.

Effect

The effect of this routine is to set the value of the *max-active-levels-var* ICV to the value specified in the argument.

If the number of parallel levels requested exceeds the number of levels of parallelism supported by the implementation, the value of the *max-active-levels-var* ICV will be set to the number of parallel levels support by the implementation.

This routine has the described effect only when called from the sequential part of the program. When called from within an explicit **`parallel`** region, the effect of this routine is implementation defined.

Cross References

- *thread-limit-var* ICV, see Section 2.3 on page 28.
- **`omp_get_max_active_levels`** routine, see Section 3.2.15 on page 127.
- **`OMP_MAX_ACTIVE_LEVELS`** environment variable, see Section 4.7 on page 150.

3.2.15 `omp_get_max_active_levels`

Summary

The **`omp_get_max_active_levels`** routine returns the value of the *max-active-levels-var ICV*, which determines the maximum number of nested active parallel regions.

Binding

When called from the sequential part of the program, the binding thread set for an **`omp_set_max_active_levels`** 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_max_active_levels`** region is implementation defined.

Effect

The effect of this routine is to set the value of the *max-active-levels-var* ICV to the value specified in the argument.

If the number of parallel levels requested exceeds the number of levels of parallelism supported by the implementation, the value of the *max-active-levels-var* ICV will be set to the number of parallel levels support by the implementation.

This routine has the described effect only when called from the sequential part of the program. When called from within an explicit **`parallel`** region, the effect of this routine is implementation defined.

Cross References

- *thread-limit-var* ICV, see Section 2.3 on page 28.
- **`omp_get_max_active_levels`** routine, see Section 3.2.15 on page 127.
- **`OMP_MAX_ACTIVE_LEVELS`** environment variable, see Section 4.7 on page 150.

3.2.15 `omp_get_max_active_levels`

Summary

The **`omp_get_max_active_levels`** routine returns the value of the *max-active-levels-var ICV*, which determines the maximum number of nested active parallel regions.

Chapter 3 Runtime Library Routines 127

Format

C/C++

```
int omp_get_max_active_levels(void)
```

C/C++

Fortran

```
integer function omp_get_max_active_levels()
```

Fortran

Binding

When called from the sequential part of the program, the binding thread set for an **`omp_get_max_active_levels`** 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_active_levels`** region is implementation defined.

Effect

The **`omp_get_max_active_levels`** routine returns the value of the *max-active-levels-var ICV*, which determines the maximum number of nested active parallel regions.

Cross References

- *thread-limit-var* ICV, see Section 2.3 on page 28.
- **`omp_set_max_active_levels`** routine, see Section 3.2.14 on page 126.
- **`OMP_MAX_ACTIVE_LEVELS`** environment variable, see Section 4.7 on page 150.

Format

C/C++

int omp_get_max_active_levels(void)

C/C++

Fortran

integer function omp_get_max_active_levels()

Fortran

Binding

When called from the sequential part of the program, the binding thread set for an **`omp_get_max_active_levels`** 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_active_levels`** region is implementation defined.

Effect

The **`omp_get_max_active_levels`** routine returns the value of the *max-active-levels-var ICV*, which determines the maximum number of nested active parallel regions.

Cross References

- *thread-limit-var* ICV, see Section 2.3 on page 28.
- **`omp_set_max_active_levels`** routine, see Section 3.2.14 on page 126.
- **`OMP_MAX_ACTIVE_LEVELS`** environment variable, see Section 4.7 on page 150.

128 OpenMP API • Version 3.0 May 2008

3.2.16 omp_get_level

Summary

The **omp_get_level** routine returns the number of nested **parallel** regions enclosing the task that contains the call.

Format

C/C++

```
int omp_get_level(void)
```

C/C++

Fortran

```
integer function omp_get_level()
```

Fortran

Binding

The binding task set for an **omp_get_level** region is the generating task. The binding region for an **omp_get_level** region is the innermost enclosing parallel region.

Effect

The **omp_get_level** routine returns the number of nested **parallel** regions (whether active or inactive) enclosing the task that contains the call, not including the implicit parallel region. The routine always returns a non-negative integer, and returns 0 if it is called from the sequential part of the program.

3.2.16 omp_get_level

Summary

The **omp_get_level** routine returns the number of nested **parallel** regions enclosing the task that contains the call.

Format

C/C++

```
int omp_get_level(void)
```

C/C++

Fortran

```
integer function omp_get_level()
```

Fortran

Binding

The binding task set for an **omp_get_level** region is the generating task. The binding region for an **omp_get_level** region is the innermost enclosing parallel region.

Effect

The **omp_get_level** routine returns the number of nested **parallel** regions (whether active or inactive) enclosing the task that contains the call, not including the implicit parallel region. The routine always returns a non-negative integer, and returns 0 if it is called from the sequential part of the program.

Chapter 3 Runtime Library Routines 129

Cross References

- `omp_get_active_level` routine, see Section 3.2.19 on page 133.
- `OMP_MAX_ACTIVE_LEVELS` environment variable, see Section 4.7 on page 150.

3.2.17 omp_get_ancestor_thread_num

Summary

The **omp_get_ancestor_thread_num** routine returns, for a given nested level of the current thread, the thread number of the ancestor or the current thread.

Format

C/C++

```
int omp_get_ancestor_thread_num(int level)
```

C/C++

Fortran

```
integer function omp_get_ancestor_thread_num(level)
integer level
```

Fortran

Binding

The binding thread set for an **omp_get_ancestor_thread_num** region is the encountering thread. The binding region for an **omp_get_ancestor_thread_num** region is the innermost enclosing **parallel** region.

Cross References

- `omp_get_active_level` routine, see Section 3.2.19 on page 133.
- `OMP_MAX_ACTIVE_LEVELS` environment variable, see Section 4.7 on page 150.

3.2.17 omp_get_ancestor_thread_num

Summary

The **omp_get_ancestor_thread_num** routine returns, for a given nested level of the current thread, the thread number of the ancestor or the current thread.

Format

C/C++

```
int omp_get_ancestor_thread_num(int level)
```

C/C++

Fortran

```
integer function omp_get_ancestor_thread_num(level)
integer level
```

Fortran

Binding

The binding thread set for an **omp_get_ancestor_thread_num** region is the encountering thread. The binding region for an **omp_get_ancestor_thread_num** region is the innermost enclosing **parallel** region.

130 OpenMP API • Version 3.0 May 2008

Effect

The **`omp_get_ancestor_thread_num`** routine returns the thread number of the ancestor at a given nest level of the current thread or the thread number of the current thread. If the requested nest level is outside the range of 0 and the nest level of the current thread, as returned by the **`omp_get_level`** routine, the routine returns -1.

Note – When the **`omp_get_ancestor_thread_num`** routine is called with a value of **`level`**=0, the routine always returns 0. If **`level`**=**`omp_get_level`**(), the routine has the same effect as the **`omp_get_thread_num`** routine.

Cross References

- **`omp_get_level`** routine, see Section 3.2.16 on page 129.
- **`omp_get_thread_num`** routine, see Section 3.2.4 on page 113.
- **`omp_get_team_size`** routine, see Section 3.2.18 on page 131.

3.2.18 `omp_get_team_size`

Summary

The **`omp_get_team_size`** routine returns, for a given nested level of the current thread, the size of the thread team to which the ancestor or the current thread belongs.

Effect

The **`omp_get_ancestor_thread_num`** routine returns the thread number of the ancestor at a given nest level of the current thread or the thread number of the current thread. If the requested nest level is outside the range of 0 and the nest level of the current thread, as returned by the **`omp_get_level`** routine, the routine returns -1.

Note – When the **`omp_get_ancestor_thread_num`** routine is called with a value of **`level`**=0, the routine always returns 0. If **`level`**=**`omp_get_level`**(), the routine has the same effect as the **`omp_get_thread_num`** routine.

Cross References

- **`omp_get_level`** routine, see Section 3.2.16 on page 129.
- **`omp_get_thread_num`** routine, see Section 3.2.4 on page 113.
- **`omp_get_team_size`** routine, see Section 3.2.18 on page 131.

3.2.18 `omp_get_team_size`

Summary

The **`omp_get_team_size`** routine returns, for a given nested level of the current thread, the size of the thread team to which the ancestor or the current thread belongs.

Chapter 3 Runtime Library Routines 131

Format

C/C++

```
int omp_get_team_size(int level)
```

C/C++

Fortran

```
integer function omp_get_team_size(level)
integer level
```

Fortran

Binding

The binding thread set for an **omp_get_team_size** region is the encountering thread. The binding region for an **omp_get_team_size** region is the innermost enclosing **parallel** region.

Effect

The **omp_get_team_size** routine returns the size of the thread team to which the ancestor or the current thread belongs. If the requested nested level is outside the range of 0 and the nested level of the current thread, as returned by the **omp_get_level** routine, the routine returns -1. Inactive parallel regions are regarded like active parallel regions executed with one thread.

Note – When the **omp_get_team_size** routine is called with a value of **level**=0, the routine always returns 1. If **level**=**omp_get_level**(), the routine has the same effect as the **omp_get_num_threads** routine.

Format

C/C++

```
int omp_get_team_size(int level)
```

C/C++

Fortran

```
integer function omp_get_team_size(level)
integer level
```

Fortran

Binding

The binding thread set for an **omp_get_team_size** region is the encountering thread. The binding region for an **omp_get_team_size** region is the innermost enclosing **parallel** region.

Effect

The **omp_get_team_size** routine returns the size of the thread team to which the ancestor or the current thread belongs. If the requested nested level is outside the range of 0 and the nested level of the current thread, as returned by the **omp_get_level** routine, the routine returns -1. Inactive parallel regions are regarded like active parallel regions executed with one thread.

Note – When the **omp_get_team_size** routine is called with a value of **level**=0, the routine always returns 1. If **level**=**omp_get_level**(), the routine has the same effect as the **omp_get_num_threads** routine.

132 OpenMP API • Version 3.0 May 2008

Cross References

- **omp_get_num_threads** routine, see Section 3.2.2 on page 111.
- **omp_get_level** routine, see Section 3.2.16 on page 129.
- **omp_get_ancestor_thread_num** routine, see Section 3.2.17 on page 130.

3.2.19 omp_get_active_level

Summary

The **omp_get_active_level** routine returns the number of nested, active **parallel** regions enclosing the task that contains the call.

Format

C/C++

```
int omp_get_active_level(void)
```

C/C++

Fortran

```
integer function omp_get_active_level()
```

Fortran

Binding

The binding task set for the an **omp_get_active_level** region is the generating task. The binding region for an **omp_get_active_level** region is the innermost enclosing **parallel** region.

Cross References
• omp_get_num_threads routine, see Section 3.2.2 on page 111.
• omp_get_level routine, see Section 3.2.16 on page 129.
• omp_get_ancestor_thread_num routine, see Section 3.2.17 on page 130.
3.2.19 omp_get_active_level
Summary
The omp_get_active_level routine returns the number of nested, active parallel regions enclosing the task that contains the call.
Format
C/C++
int omp_get_active_level(void)
C/C++
Fortran
integer function omp_get_active_level()
Fortran
Binding
The binding task set for the an omp_get_active_level region is the generating task. The binding region for an omp_get_active_level region is the innermost enclosing parallel region.
Chapter 3 Runtime Library Routines 133

Effect

The **`omp_get_active_level`** routine returns the number of nested, active parallel regions enclosing the task that contains the call. The routine always returns a non-negative integer, and always returns 0 if it is called from the sequential part of the program.

Cross References

- **`omp_get_level`** routine, see Section 3.2.16 on page 129.

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; programs that otherwise access OpenMP lock variables are non-conforming.

An OpenMP lock may be in one of the following states: *uninitialized*, *unlocked*, or *locked*. If a lock is in the unlocked state, a task may *set* the lock, which changes its state to locked. The task which sets the lock is then said to *own* the lock. A task which owns a lock may *unset* that lock, returning it to the unlocked state. A program in which a task unsets a lock that is owned by another task is non-conforming.

Two types of locks are supported: *simple locks* and *nestable locks*. A nestable lock may be set multiple times by the same task before being unset; a simple lock may not be set if it is already owned by the task 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. The lock routines include a flush with no list; the read and update to the lock variable must be implemented as if they are atomic with the flush. 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 tasks.

Effect

The **`omp_get_active_level`** routine returns the number of nested, active parallel regions enclosing the task that contains the call. The routine always returns a non-negative integer, and always returns 0 if it is called from the sequential part of the program.

Cross References

- **`omp_get_level`** routine, see Section 3.2.16 on page 129.

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; programs that otherwise access OpenMP lock variables are non-conforming.

An OpenMP lock may be in one of the following states: *uninitialized*, *unlocked*, or *locked*. If a lock is in the unlocked state, a task may *set* the lock, which changes its state to locked. The task which sets the lock is then said to *own* the lock. A task which owns a lock may *unset* that lock, returning it to the unlocked state. A program in which a task unsets a lock that is owned by another task is non-conforming.

Two types of locks are supported: *simple locks* and *nestable locks*. A nestable lock may be set multiple times by the same task before being unset; a simple lock may not be set if it is already owned by the task 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. The lock routines include a flush with no list; the read and update to the lock variable must be implemented as if they are atomic with the flush. 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 tasks.

134 OpenMP API • Version 3.0 May 2008

See Section A.44 on page 271 and Section A.45 on page 274, 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. As a consequence, for each OpenMP lock, the lock routine effects relate to all tasks that call the routines, without regard to which team(s) the threads executing the tasks belong.

Simple Lock Routines

C/C++

The type **`omp_lock_t`** is an data type capable of representing a simple lock. For the following routines, a simple 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`**.

C/C++

Fortran

For the following routines, a simple lock variable must be an integer variable of **`kind=omp_lock_kind`**.

Fortran

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.

See Section A.44 on page 271 and Section A.45 on page 274, 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. As a consequence, for each OpenMP lock, the lock routine effects relate to all tasks that call the routines, without regard to which team(s) the threads executing the tasks belong.

Simple Lock Routines

C/C++

The type omp_lock_t is an data type capable of representing a simple lock. For the following routines, a simple 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.

C/C++

Fortran

For the following routines, a simple lock variable must be an integer variable of kind=omp_lock_kind.

Fortran

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.

Chapter 3 Runtime Library Routines 135

Nestable Lock Routines:

C/C++

The type **`omp_nest_lock_t`** is an data type capable of representing a nestable lock. For the following routines, a nested 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`**.

C/C++

Fortran

For the following routines, a nested lock variable must be an integer variable of **`kind=omp_nest_lock_kind`**.

Fortran

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.
- The **`omp_unset_nest_lock`** routine unsets a nestable lock.
- The **`omp_test_nest_lock`** routine tests a nestable lock, and sets it if it is available.

3.3.1 `omp_init_lock` and `omp_init_nest_lock`

Summary

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

Nestable Lock Routines:

C/C++

The type **`omp_nest_lock_t`** is an data type capable of representing a nestable lock. For the following routines, a nested 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`**.

C/C++

Fortran

For the following routines, a nested lock variable must be an integer variable of **`kind=omp_nest_lock_kind`**.

Fortran

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.
- The **`omp_unset_nest_lock`** routine unsets a nestable lock.
- The **`omp_test_nest_lock`** routine tests a nestable lock, and sets it if it is available.

3.3.1 `omp_init_lock` and `omp_init_nest_lock`

Summary

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

136 OpenMP API • Version 3.0 May 2008

Format

C/C++

```
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

C/C++

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
```

Fortran

Constraints on Arguments

A program that accesses a lock that is not in the uninitialized state through either routine is non-conforming.

Effect

The effect of these routines is to initialize the lock to the unlocked state (that is, no task 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.42 on page 269.

3.3.2 omp_destroy_lock and omp_destroy_nest_lock

Summary

These routines ensure that the OpenMP lock is uninitialized.

Format

C/C++

void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);

C/C++

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

Fortran

Constraints on Arguments

A program that accesses a lock that is not in the uninitialized state through either routine is non-conforming.

Effect

The effect of these routines is to initialize the lock to the unlocked state (that is, no task 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.42 on page 269.

3.3.2 omp_destroy_lock and omp_destroy_nest_lock

Summary

These routines ensure that the OpenMP lock is uninitialized.

Chapter 3 Runtime Library Routines 137

Format

C/C++

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

C/C++

Fortran

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

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

Fortran

Constraints on Arguments

A program that accesses a lock that is not in the unlocked state through either routine is non-conforming.

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 task region is suspended until the lock is set.

Format

C/C++

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

C/C++

Fortran

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

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

Fortran

Constraints on Arguments

A program that accesses a lock that is not in the unlocked state through either routine is non-conforming.

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 task region is suspended until the lock is set.

138 OpenMP API • Version 3.0 May 2008

Format

C/C++

```
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

C/C++

Fortran

```
subroutine omp_set_lock(svar)
integer (kind=omp_lock_kind) svar

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

Fortran

Constraints on Arguments

A program that accesses a lock that is in the uninitialized state through either routine is non-conforming. A simple lock accessed by **omp_set_lock** that is in the locked state must not be owned by the task that contains the call or deadlock will result.

Effect

Each of these routines causes suspension of the task 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 task executing the routine.

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

Format

C/C++

```
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

C/C++

Fortran

```
subroutine omp_set_lock(svar)
integer (kind=omp_lock_kind) svar

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

Fortran

Constraints on Arguments

A program that accesses a lock that is in the uninitialized state through either routine is non-conforming. A simple lock accessed by **omp_set_lock** that is in the locked state must not be owned by the task that contains the call or deadlock will result.

Effect

Each of these routines causes suspension of the task 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 task executing the routine.

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

Chapter 3 Runtime Library Routines 139

3.3.4 omp_unset_lock and omp_unset_nest_lock

Summary

These routines provide the means of unsetting an OpenMP lock.

Format

C/C++

```
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

C/C++

Fortran

```
subroutine omp_unset_lock(svar)
integer (kind=omp_lock_kind) svar

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

Fortran

Constraints on Arguments

A program that accesses a lock that is not in the locked state or that is not owned by the task that contains the call through either routine is non-conforming.

Effect

For a simple lock, the **omp_unset_lock** routine causes the lock to become unlocked.

For a nestable lock, the **omp_unset_nest_lock** routine decrements the nesting count, and causes the lock to become unlocked if the resulting nesting count is zero.

For either routine, if the lock becomes unlocked, and if one or more tasks regions were suspended because the lock was unavailable, the effect is that one task is chosen and given ownership of the lock.

3.3.4 omp_unset_lock and omp_unset_nest_lock

Summary

These routines provide the means of unsetting an OpenMP lock.

Format

C/C++

```
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

C/C++

Fortran

```
subroutine omp_unset_lock(svar)
integer (kind=omp_lock_kind) svar

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

Fortran

Constraints on Arguments

A program that accesses a lock that is not in the locked state or that is not owned by the task that contains the call through either routine is non-conforming.

Effect

For a simple lock, the **omp_unset_lock** routine causes the lock to become unlocked.

For a nestable lock, the **omp_unset_nest_lock** routine decrements the nesting count, and causes the lock to become unlocked if the resulting nesting count is zero.

For either routine, if the lock becomes unlocked, and if one or more tasks regions were suspended because the lock was unavailable, the effect is that one task is chosen and given ownership of the lock.

140 OpenMP API • Version 3.0 May 2008

3.3.5 `omp_test_lock` and `omp_test_nest_lock`

Summary

These routines attempt to set an OpenMP lock but do not suspend execution of the task executing the routine.

Format

C/C++

```
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

C/C++

Fortran

```
logical function omp_test_lock(svar)
integer (kind=omp_lock_kind) svar

integer function omp_test_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Fortran

Constraints on Arguments

A program that accesses a lock that is in the uninitialized state through either routine is non-conforming. The behavior is unspecified if a simple lock accessed by **`omp_test_lock`** that is in the locked state is owned by the task that contains the call.

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 suspend execution of the task executing the routine.

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

3.3.5 `omp_test_lock` and `omp_test_nest_lock`

Summary

These routines attempt to set an OpenMP lock but do not suspend execution of the task executing the routine.

Format

C/C++

```
int omp_test_lock(omp_lock_t *lock);
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

C/C++

Fortran

```
logical function omp_test_lock(svar)
integer (kind=omp_lock_kind) svar

integer function omp_test_nest_lock(nvar)
integer (kind=omp_nest_lock_kind) nvar
```

Fortran

Constraints on Arguments

A program that accesses a lock that is in the uninitialized state through either routine is non-conforming. The behavior is unspecified if a simple lock accessed by **`omp_test_lock`** that is in the locked state is owned by the task that contains the call.

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 suspend execution of the task executing the routine.

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

Chapter 3 Runtime Library Routines 141

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);
```

C/C++

Fortran

```
double precision function omp_get_wtime()
```

Fortran

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.

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);

C/C++

Fortran

double precision function omp_get_wtime()

Fortran

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.

142 OpenMP API • Version 3.0 May 2008

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/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);
```

C/C++

Fortran

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

Fortran

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/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);
```

C/C++

Fortran

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

Fortran

Chapter 3 Runtime Library Routines 143

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/C++

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

C/C++

Fortran

```
double precision function omp_get_wtick()
```

Fortran

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**.

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/C++

double omp_get_wtick(void);

C/C++

Fortran

double precision function omp_get_wtick()

Fortran

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.

144 OpenMP API • Version 3.0 May 2008

CHAPTER 4

Environment Variables

This chapter describes the OpenMP environment variables that specify the settings of the ICVs that affect the execution of OpenMP programs (see Section 2.3 on page 28). The names of the environment variables must be upper case. 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 some of the ICVs 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* ICV for the runtime schedule type and chunk size. It can be set to any of the valid OpenMP schedule types (i.e., **`static`**, **`dynamic`**, **`guided`**, and **`auto`**).
- **`OMP_NUM_THREADS`** sets the *nthreads-var* ICV for the number of threads to use for **`parallel`** regions.
- **`OMP_DYNAMIC`** sets the *dyn-var* ICV for the dynamic adjustment of threads to use for **`parallel`** regions.
- **`OMP_NESTED`** sets the *nest-var* ICV to enable or to disable nested parallelism.
- **`OMP_STACKSIZE`** sets the *stacksize-var* ICV that specifies the size of the stack for threads created by the OpenMP implementation.
- **`OMP_WAIT_POLICY`** sets the *wait-policy-var* ICV that controls the desired behavior of waiting threads.
- **`OMP_MAX_ACTIVE_LEVELS`** sets the *max-active-levels-var* ICV that controls the maximum number of nested active parallel regions.
- **`OMP_THREAD_LIMIT`** sets the *thread-limit-var* ICV that controls the maximum number of threads participating in the OpenMP program.

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:

CHAPTER 4

Environment Variables

This chapter describes the OpenMP environment variables that specify the settings of the ICVs that affect the execution of OpenMP programs (see Section 2.3 on page 28). The names of the environment variables must be upper case. 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 some of the ICVs 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 ICV for the runtime schedule type and chunk size. It can be set to any of the valid OpenMP schedule types (i.e., static, dynamic, guided, and auto).
- OMP_NUM_THREADS sets the nthreads-var ICV for the number of threads to use for parallel regions.
- OMP_DYNAMIC sets the dyn-var ICV for the dynamic adjustment of threads to use for parallel regions.
- OMP_NESTED sets the nest-var ICV to enable or to disable nested parallelism.
- OMP_STACKSIZE sets the stacksize-var ICV that specifies the size of the stack for threads created by the OpenMP implementation.
- OMP_WAIT_POLICY sets the wait-policy-var ICV that controls the desired behavior of waiting threads.
- OMP_MAX_ACTIVE_LEVELS sets the max-active-levels-var ICV that controls the maximum number of nested active parallel regions.
- OMP_THREAD_LIMIT sets the thread-limit-var ICV that controls the maximum number of threads participating in the OpenMP program.

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:

145

- csh:

```
setenv OMP_SCHEDULE "dynamic"
```

- ksh:

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

- DOS:

```
set 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* ICV.

The value of this environment variable takes the form:

type[,*chunk*]

where

- *type* is one of **static**, **dynamic**, **guided**, or **auto**
- *chunk* is an optional positive integer that 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 38 for a detailed description of the schedule types.

The behavior of the program is implementation defined if the value of **OMP_SCHEDULE** does not conform to the above format.

Implementation specific schedules cannot be specified in **OMP_SCHEDULE**. They can only be specified by calling **omp_set_schedule**, described in Section 3.2.11 on page 121.

Example:

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


- csh:

```
setenv OMP_SCHEDULE "dynamic"
```

- ksh:

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

- DOS:

```
set 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* ICV.

The value of this environment variable takes the form:

type[,*chunk*]

where

- *type* is one of **static**, **dynamic**, **guided**, or **auto**
- *chunk* is an optional positive integer that 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 38 for a detailed description of the schedule types.

The behavior of the program is implementation defined if the value of **OMP_SCHEDULE** does not conform to the above format.

Implementation specific schedules cannot be specified in **OMP_SCHEDULE**. They can only be specified by calling **omp_set_schedule**, described in Section 3.2.11 on page 121.

Example:

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

146 OpenMP API • Version 3.0 May 2008

Cross References

- *run-sched-var* ICV, see Section 2.3 on page 28.
- Loop construct, see Section 2.5.1 on page 38.
- Parallel loop construct, see Section 2.6.1 on page 54.
- **`omp_set_schedule`** routine, see Section 3.2.11 on page 121.
- **`omp_get_schedule`** routine, see Section 3.2.12 on page 123.

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* ICV. 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.

Example:

```
setenv OMP_NUM_THREADS 16
```

Cross References:

- *nthreads-var* ICV, see Section 2.3 on page 28.
- **`num_threads`** clause, Section 2.4 on page 32.
- **`omp_set_num_threads`** routine, see Section 3.2.1 on page 110.
- **`omp_get_num_threads`** routine, see Section 3.2.2 on page 111.
- **`omp_get_max_threads`** routine, see Section 3.2.3 on page 112.
- **`omp_get_team_size`** routine, see Section 3.2.18 on page 131.

Cross References

- *run-sched-var* ICV, see Section 2.3 on page 28.
- Loop construct, see Section 2.5.1 on page 38.
- Parallel loop construct, see Section 2.6.1 on page 54.
- **`omp_set_schedule`** routine, see Section 3.2.11 on page 121.
- **`omp_get_schedule`** routine, see Section 3.2.12 on page 123.

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* ICV. 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.

Example:

```
setenv OMP_NUM_THREADS 16
```

Cross References:

- *nthreads-var* ICV, see Section 2.3 on page 28.
- **`num_threads`** clause, Section 2.4 on page 32.
- **`omp_set_num_threads`** routine, see Section 3.2.1 on page 110.
- **`omp_get_num_threads`** routine, see Section 3.2.2 on page 111.
- **`omp_get_max_threads`** routine, see Section 3.2.3 on page 112.
- **`omp_get_team_size`** routine, see Section 3.2.18 on page 131.

Chapter 4 Environment Variables 147

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* ICV. 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. The behavior of the program is implementation defined if the value of **OMP_DYNAMIC** is neither **true** nor **false**.

Example:

```
setenv OMP_DYNAMIC true
```

Cross References:

- *dyn-var* ICV, see Section 2.3 on page 28.
- **omp_set_dynamic** routine, see Section 3.2.7 on page 117.
- **omp_get_dynamic** routine, see Section 3.2.8 on page 118.

4.4 OMP_NESTED

The **OMP_NESTED** environment variable controls nested parallelism by setting the initial value of the *nest-var* ICV. 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. The behavior of the program is implementation defined if the value of **OMP_NESTED** is neither **true** nor **false**.

Example:

```
setenv OMP_NESTED false
```

Cross References

- *nest-var* ICV, see Section 2.3 on page 28.
- **omp_set_nested** routine, see Section 3.2.9 on page 119.

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* ICV. 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. The behavior of the program is implementation defined if the value of **OMP_DYNAMIC** is neither **true** nor **false**.

Example:

```
setenv OMP_DYNAMIC true
```

Cross References:

- *dyn-var* ICV, see Section 2.3 on page 28.
- **omp_set_dynamic** routine, see Section 3.2.7 on page 117.
- **omp_get_dynamic** routine, see Section 3.2.8 on page 118.

4.4 OMP_NESTED

The **OMP_NESTED** environment variable controls nested parallelism by setting the initial value of the *nest-var* ICV. 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. The behavior of the program is implementation defined if the value of **OMP_NESTED** is neither **true** nor **false**.

Example:

```
setenv OMP_NESTED false
```

Cross References

- *nest-var* ICV, see Section 2.3 on page 28.
- **omp_set_nested** routine, see Section 3.2.9 on page 119.

148 OpenMP API • Version 3.0 May 2008

- **omp_get_nested** routine, see Section 3.2.18 on page 131.

4.5 OMP_STACKSIZE

The **OMP_STACKSIZE** environment variable controls the size of the stack for threads created by the OpenMP implementation, by setting the value of the *stacksize-var* ICV. The environment variable does not control the size of the stack for the initial thread.

The value of this environment variable takes the form:

size | *size***B** | *size***K** | *size***M** | *size***G**

where:

- *size* is a positive integer that specifies the size of the stack for threads that are created by the OpenMP implementation.
- **B**, **K**, **M**, and **G** are letters that specify whether the given size is in Bytes, Kilobytes, Megabytes, or Gigabytes, respectively. If one of these letters is present, there may be white space between *size* and the letter.

If only *size* is specified and none of **B**, **K**, **M**, or **G** is specified, then *size* is assumed to be in Kilobytes.

The behavior of the program is implementation defined if **OMP_STACKSIZE** does not conform to the above format, or if the implementation cannot provide a stack with the requested size.

Examples:

```
setenv OMP_STACKSIZE 2000500B
setenv OMP_STACKSIZE "3000 k "
setenv OMP_STACKSIZE 10M
setenv OMP_STACKSIZE " 10 M "
setenv OMP_STACKSIZE "20 m "
setenv OMP_STACKSIZE " 1G"
setenv OMP_STACKSIZE 20000
```

Cross References

- *stacksize-var* ICV, see Section 2.3 on page 28.

- **omp_get_nested** routine, see Section 3.2.18 on page 131.

4.5 OMP_STACKSIZE

The **OMP_STACKSIZE** environment variable controls the size of the stack for threads created by the OpenMP implementation, by setting the value of the *stacksize-var* ICV. The environment variable does not control the size of the stack for the initial thread.

The value of this environment variable takes the form:

size | *size***B** | *size***K** | *size***M** | *size***G**

where:

- *size* is a positive integer that specifies the size of the stack for threads that are created by the OpenMP implementation.
- **B**, **K**, **M**, and **G** are letters that specify whether the given size is in Bytes, Kilobytes, Megabytes, or Gigabytes, respectively. If one of these letters is present, there may be white space between *size* and the letter.

If only *size* is specified and none of **B**, **K**, **M**, or **G** is specified, then *size* is assumed to be in Kilobytes.

The behavior of the program is implementation defined if **OMP_STACKSIZE** does not conform to the above format, or if the implementation cannot provide a stack with the requested size.

Examples:

```
setenv OMP_STACKSIZE 2000500B
setenv OMP_STACKSIZE "3000 k "
setenv OMP_STACKSIZE 10M
setenv OMP_STACKSIZE " 10 M "
setenv OMP_STACKSIZE "20 m "
setenv OMP_STACKSIZE " 1G"
setenv OMP_STACKSIZE 20000
```

Cross References

- *stacksize-var* ICV, see Section 2.3 on page 28.

Chapter 4 Environment Variables 149

4.6 OMP_WAIT_POLICY

The **OMP_WAIT_POLICY** environment variable provides a hint to an OpenMP implementation about the desired behavior of waiting threads by setting the *wait-policy-var* ICV. A compliant OpenMP implementation may or may not abide by the setting of the environment variable.

The value of this environment variable takes the form:

ACTIVE | **PASSIVE**

The **ACTIVE** value specifies that waiting threads should mostly be active, i.e., consume processor cycles, while waiting. An OpenMP implementation may, for example, make waiting threads spin.

The **PASSIVE** value specifies that waiting threads should mostly be passive, i.e., not consume processor cycles, while waiting. An OpenMP implementation, may for example, make waiting threads yield the processor to other threads or go to sleep.

The details of the **ACTIVE** and **PASSIVE** behaviors are implementation defined.

Examples:

```
setenv OMP_WAIT_POLICY ACTIVE
setenv OMP_WAIT_POLICY active
setenv OMP_WAIT_POLICY PASSIVE
setenv OMP_WAIT_POLICY passive
```

Cross References

- *wait-policy-var* ICV, see Section 2.3 on page 24.

4.7 OMP_MAX_ACTIVE_LEVELS

The **OMP_MAX_ACTIVE_LEVELS** environment variable controls the maximum number of nested active parallel regions by setting the initial value of the *max-active-levels-var* ICV.

4.6 OMP_WAIT_POLICY

The OMP_WAIT_POLICY environment variable provides a hint to an OpenMP implementation about the desired behavior of waiting threads by setting the wait-policy-var ICV. A compliant OpenMP implementation may or may not abide by the setting of the environment variable.

The value of this environment variable takes the form:

ACTIVE | PASSIVE

The ACTIVE value specifies that waiting threads should mostly be active, i.e., consume processor cycles, while waiting. An OpenMP implementation may, for example, make waiting threads spin.

The PASSIVE value specifies that waiting threads should mostly be passive, i.e., not consume processor cycles, while waiting. An OpenMP implementation, may for example, make waiting threads yield the processor to other threads or go to sleep.

The details of the ACTIVE and PASSIVE behaviors are implementation defined.

Examples:

setenv OMP_WAIT_POLICY ACTIVE
setenv OMP_WAIT_POLICY active
setenv OMP_WAIT_POLICY PASSIVE
setenv OMP_WAIT_POLICY passive

Cross References

• wait-policy-var ICV, see Section 2.3 on page 24.

4.7 OMP_MAX_ACTIVE_LEVELS

The OMP_MAX_ACTIVE_LEVELS environment variable controls the maximum number of nested active parallel regions by setting the initial value of the max-active-levels-var ICV.

150 OpenMP API • Version 3.0 May 2008

The value of this environment variable must be a non-negative integer. The behavior of the program is implementation defined if the requested value of **`OMP_MAX_ACTIVE_LEVELS`** is greater than the maximum number of nested active parallel levels an implementation can support, or if the value is not a non-negative integer.

Cross References

- *max-active-levels-var* ICV, see Section 2.3 on page 28.
- **`omp_set_max_active_levels`** routine, see Section 3.2.14 on page 126.
- **`omp_get_max_active_levels`** routine, see Section 3.2.15 on page 127.

4.8 OMP_THREAD_LIMIT

The **`OMP_THREAD_LIMIT`** environment variable sets the number of OpenMP threads to use for the whole OpenMP program by setting the *thread-limit-var* ICV.

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_THREAD_LIMIT`** is greater than the number of threads an implementation can support, or if the value is not a positive integer.

Cross References

- *thread-limit-var* ICV, see Section 2.3 on page 28.
- **`omp_get_thread_limit`** routine

The value of this environment variable must be a non-negative integer. The behavior of the program is implementation defined if the requested value of **`OMP_MAX_ACTIVE_LEVELS`** is greater than the maximum number of nested active parallel levels an implementation can support, or if the value is not a non-negative integer.

Cross References

- *max-active-levels-var* ICV, see Section 2.3 on page 28.
- **`omp_set_max_active_levels`** routine, see Section 3.2.14 on page 126.
- **`omp_get_max_active_levels`** routine, see Section 3.2.15 on page 127.

4.8 OMP_THREAD_LIMIT

The **`OMP_THREAD_LIMIT`** environment variable sets the number of OpenMP threads to use for the whole OpenMP program by setting the *thread-limit-var* ICV.

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_THREAD_LIMIT`** is greater than the number of threads an implementation can support, or if the value is not a positive integer.

Cross References

- *thread-limit-var* ICV, see Section 2.3 on page 28.
- **`omp_get_thread_limit`** routine

Chapter 4 Environment Variables 151

152 OpenMP API • Version 3.0 May 2008

APPENDIX A

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.

C/C++

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 54). The loop iteration variable is private by default, so it is not necessary to specify it explicitly in a **`private`** clause.

C/C++

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;
}
```

C/C++

APPENDIX A

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.

C/C++

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 54). The loop iteration variable is private by default, so it is not necessary to specify it explicitly in a **`private`** clause.

C/C++

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;
}
```

C/C++

153

Fortran

Example A.1.1f

```
      SUBROUTINE A1(N, A, B)

        INTEGER I, N
        REAL B(N), A(N)

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

      END SUBROUTINE A1
```

Fortran

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.

Fortran

Example A.1.1f

```
      SUBROUTINE A1(N, A, B)

        INTEGER I, N
        REAL B(N), A(N)

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

      END SUBROUTINE A1
```

Fortran

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.

154 OpenMP API • Version 3.0 May 2008

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/C++

Example A.2.1c

```
#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 );
    }

    #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;
}
```

C/C++

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/C++

Example A.2.1c

```
#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 );
    }

    #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;
}
```

C/C++

Appendix A Examples 155

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
```

Fortran

The following example demonstrates why synchronization is difficult to perform correctly through variables. The value of flag is undefined in both prints on thread 1 and the value of data is only well-defined in the second print.

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
```

Fortran

The following example demonstrates why synchronization is difficult to perform correctly through variables. The value of flag is undefined in both prints on thread 1 and the value of data is only well-defined in the second print.

156 OpenMP API • Version 3.0 May 2008

C/C++

Example A.2.2c

```
#include <omp.h>
#include <stdio.h>
int main()
{
    int data;
    int flag=0;
    #pragma omp parallel
     {
       if (omp_get_thread_num()==0)
        {
            /* Write to the data buffer that will be
            read by thread */
            data = 42;
            /* Flush data to thread 1 and strictly order
            the write to data
            relative to the write to the flag */
            #pragma omp flush(flag, data)
            /* Set flag to release thread 1 */
            flag = 1;
            /* Flush flag to ensure that thread 1 sees
            the change */
            #pragma omp flush(flag)
        }
       else if(omp_get_thread_num()==1)
        {
            /* Loop until we see the update to the flag */
            #pragma omp flush(flag, data)
            while (flag < 1)
              {
                #pragma omp flush(flag, data)
              }
            /* Values of flag and data are undefined */
            printf("flag=%d data=%d\n", flag, data);
            #pragma omp flush(flag, data)
            /* Values data will be 42, value of flag
            still undefined */
            printf("flag=%d data=%d\n", flag, data);
        }
    }
}
```

C/C++

C/C++

Example A.2.2c

```
#include <omp.h>
#include <stdio.h>
int main()
{
    int data;
    int flag=0;
    #pragma omp parallel
     {
       if (omp_get_thread_num()==0)
        {
            /* Write to the data buffer that will be
            read by thread */
            data = 42;
            /* Flush data to thread 1 and strictly order
            the write to data
            relative to the write to the flag */
            #pragma omp flush(flag, data)
            /* Set flag to release thread 1 */
            flag = 1;
            /* Flush flag to ensure that thread 1 sees
            the change */
            #pragma omp flush(flag)
        }
       else if(omp_get_thread_num()==1)
        {
            /* Loop until we see the update to the flag */
            #pragma omp flush(flag, data)
            while (flag < 1)
              {
                #pragma omp flush(flag, data)
              }
            /* Values of flag and data are undefined */
            printf("flag=%d data=%d\n", flag, data);
            #pragma omp flush(flag, data)
            /* Values data will be 42, value of flag
            still undefined */
            printf("flag=%d data=%d\n", flag, data);
        }
    }
}
```

C/C++

Appendix A Examples 157

Fortran

Example A.2.2f

```
      PROGRAM EXAMPLE
      INCLUDE "omp_lib.h" ! or USE OMP_LIB
      INTEGER DATA
      INTEGER FLAG

!$OMP PARALLEL
        IF(OMP_GET_THREAD_NUM() .EQ. 0) THEN
                ! Write to the data buffer that will be read by thread 1
                DATA = 42
               ! Flush DATA to thread 1 and strictly order the write to DATA
                ! relative to the write to the FLAG
                !$OMP FLUSH(FLAG, DATA)
                ! Set FLAG to release thread 1
                FLAG = 1;
                ! Flush FLAG to ensure that thread 1 sees the change */
                !$OMP FLUSH(FLAG)
        ELSE IF(OMP_GET_THREAD_NUM() .EQ. 1) THEN
                ! Loop until we see the update to the FLAG
                !$OMP FLUSH(FLAG, DATA)
                DO WHILE(FLAG .LT. 1)
                        !$OMP FLUSH(FLAG, DATA)
                ENDDO

                ! Values of FLAG and DATA are undefined
                PRINT *, 'FLAG=', FLAG, ' DATA=', DATA
                !$OMP FLUSH(FLAG, DATA)

                !Values DATA will be 42, value of FLAG still undefined */
                PRINT *, 'FLAG=', FLAG, ' DATA=', DATA
        ENDIF
!$OMP END PARALLEL
      END
```

Fortran

This example demonstrates why synchronization is difficult to perform correctly through variables. The statements on thread 1 and thread 2 may execute in either order.

Fortran

Example A.2.2f

```
      PROGRAM EXAMPLE
      INCLUDE "omp_lib.h" ! or USE OMP_LIB
      INTEGER DATA
      INTEGER FLAG

!$OMP PARALLEL
        IF(OMP_GET_THREAD_NUM() .EQ. 0) THEN
                ! Write to the data buffer that will be read by thread 1
                DATA = 42
               ! Flush DATA to thread 1 and strictly order the write to DATA
                ! relative to the write to the FLAG
                !$OMP FLUSH(FLAG, DATA)
                ! Set FLAG to release thread 1
                FLAG = 1;
                ! Flush FLAG to ensure that thread 1 sees the change */
                !$OMP FLUSH(FLAG)
        ELSE IF(OMP_GET_THREAD_NUM() .EQ. 1) THEN
                ! Loop until we see the update to the FLAG
                !$OMP FLUSH(FLAG, DATA)
                DO WHILE(FLAG .LT. 1)
                        !$OMP FLUSH(FLAG, DATA)
                ENDDO

                ! Values of FLAG and DATA are undefined
                PRINT *, 'FLAG=', FLAG, ' DATA=', DATA
                !$OMP FLUSH(FLAG, DATA)

                !Values DATA will be 42, value of FLAG still undefined */
                PRINT *, 'FLAG=', FLAG, ' DATA=', DATA
        ENDIF
!$OMP END PARALLEL
      END
```

Fortran

This example demonstrates why synchronization is difficult to perform correctly through variables. The statements on thread 1 and thread 2 may execute in either order.

158 OpenMP API • Version 3.0 May 2008

C/C++

Example A.2.3c

```c
#include <omp.h>
#include <stdio.h>
int main()
{
        int flag=0;

        #pragma omp parallel
        {
                if(omp_get_thread_num()==0)
                {
                        /* Set flag to release thread 1 */
                        #pragma omp atomic
                        flag++;
                      /* Flush of flag is implied by the atomic directive */
                }
                else if(omp_get_thread_num()==1)
                {
                        /* Loop until we see that flag reaches 1*/
                        #pragma omp flush(flag)
                        while(flag < 1)
                        {
                                #pragma omp flush(flag)
                        }
                        printf("Thread 1 awoken\n");

                        /* Set flag to release thread 2 */
                        #pragma omp atomic
                        flag++;
                      /* Flush of flag is implied by the atomic directive */
                }
                else if(omp_get_thread_num()==2)
                {
                        /* Loop until we see that flag reaches 2 */
                        #pragma omp flush(flag)
                        while(flag < 2)
                        {
                                #pragma omp flush(flag)
                        }
                        printf("Thread 2 awoken\n");
                }
        }
}
```

C/C++

C/C++

Example A.2.3c

```c
#include <omp.h>
#include <stdio.h>
int main()
{
        int flag=0;

        #pragma omp parallel
        {
                if(omp_get_thread_num()==0)
                {
                        /* Set flag to release thread 1 */
                        #pragma omp atomic
                        flag++;
                      /* Flush of flag is implied by the atomic directive */
                }
                else if(omp_get_thread_num()==1)
                {
                        /* Loop until we see that flag reaches 1*/
                        #pragma omp flush(flag)
                        while(flag < 1)
                        {
                                #pragma omp flush(flag)
                        }
                        printf("Thread 1 awoken\n");

                        /* Set flag to release thread 2 */
                        #pragma omp atomic
                        flag++;
                      /* Flush of flag is implied by the atomic directive */
                }
                else if(omp_get_thread_num()==2)
                {
                        /* Loop until we see that flag reaches 2 */
                        #pragma omp flush(flag)
                        while(flag < 2)
                        {
                                #pragma omp flush(flag)
                        }
                        printf("Thread 2 awoken\n");
                }
        }
}
```

C/C++

Appendix A Examples 159

Fortran

Example A.2.3f

```fortran
      PROGRAM EXAMPLE
      INCLUDE "omp_lib.h" ! or USE OMP_LIB
      INTEGER FLAG

!$OMP PARALLEL
        IF(OMP_GET_THREAD_NUM() .EQ. 0) THEN
                ! Set flag to release thread 1
                !$OMP ATOMIC
                        FLAG = FLAG + 1
                !Flush of FLAG is implied by the atomic directive
        ELSE IF(OMP_GET_THREAD_NUM() .EQ. 1) THEN
                        ! Loop until we see that FLAG reaches 1
                        !$OMP FLUSH(FLAG, DATA)
                        DO WHILE(FLAG .LT. 1)
                                !$OMP FLUSH(FLAG, DATA)
                        ENDDO

                        PRINT *, 'Thread 1 awoken'

                        ! Set FLAG to release thread 2
                        !$OMP ATOMIC
                                FLAG = FLAG + 1
                        !Flush of FLAG is implied by the atomic directive
        ELSE IF(OMP_GET_THREAD_NUM() .EQ. 2) THEN
                        ! Loop until we see that FLAG reaches 2
                        !$OMP FLUSH(FLAG, DATA)
                        DO WHILE(FLAG .LT. 2)
                                !$OMP FLUSH(FLAG,    DATA)
                        ENDDO

                        PRINT *, 'Thread 2 awoken'
        ENDIF
!$OMP END PARALLEL
      END
```

Fortran

Fortran

Example A.2.3f

```fortran
      PROGRAM EXAMPLE
      INCLUDE "omp_lib.h" ! or USE OMP_LIB
      INTEGER FLAG

!$OMP PARALLEL
        IF(OMP_GET_THREAD_NUM() .EQ. 0) THEN
                ! Set flag to release thread 1
                !$OMP ATOMIC
                        FLAG = FLAG + 1
                !Flush of FLAG is implied by the atomic directive
        ELSE IF(OMP_GET_THREAD_NUM() .EQ. 1) THEN
                        ! Loop until we see that FLAG reaches 1
                        !$OMP FLUSH(FLAG, DATA)
                        DO WHILE(FLAG .LT. 1)
                                !$OMP FLUSH(FLAG, DATA)
                        ENDDO

                        PRINT *, 'Thread 1 awoken'

                        ! Set FLAG to release thread 2
                        !$OMP ATOMIC
                                FLAG = FLAG + 1
                        !Flush of FLAG is implied by the atomic directive
        ELSE IF(OMP_GET_THREAD_NUM() .EQ. 2) THEN
                        ! Loop until we see that FLAG reaches 2
                        !$OMP FLUSH(FLAG, DATA)
                        DO WHILE(FLAG .LT. 2)
                                !$OMP FLUSH(FLAG,    DATA)
                        ENDDO

                        PRINT *, 'Thread 2 awoken'
        ENDIF
!$OMP END PARALLEL
      END
```

Fortran

160 OpenMP API • Version 3.0 May 2008

A.3 Conditional Compilation

C/C++

The following example illustrates the use of conditional compilation using the OpenMP macro **_OPENMP** (Section 2.2 on page 26). With OpenMP compilation, the **_OPENMP** macro becomes defined.

C/C++

Example A.3.1c

```
#include <stdio.h>

int main()
{

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

    return 0;
}
```

C/C++

Fortran

The following example illustrates the use of the conditional compilation sentinel (see Section 2.2 on page 26). 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.

Fortran

Example A.3.1f

```
      PROGRAM A3

C234567890
!$    PRINT *, "Compiled by an OpenMP-compliant implementation."

      END PROGRAM A3
```

Fortran

A.3 Conditional Compilation

C/C++

The following example illustrates the use of conditional compilation using the OpenMP macro **_OPENMP** (Section 2.2 on page 26). With OpenMP compilation, the **_OPENMP** macro becomes defined.

C/C++

Example A.3.1c

```
#include <stdio.h>

int main()
{

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

    return 0;
}
```

C/C++

Fortran

The following example illustrates the use of the conditional compilation sentinel (see Section 2.2 on page 26). 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.

Fortran

Example A.3.1f

```
      PROGRAM A3

C234567890
!$    PRINT *, "Compiled by an OpenMP-compliant implementation."

      END PROGRAM A3
```

Fortran

Appendix A Examples 161

A.4 Internal Control Variables

According to Section 2.3 on page 28, an OpenMP implementation must act as if there are ICVs that control the behavior of the program. This example illustrates two ICVs, *nthreads-var* and *max_active-levels-var*. The *nthreads-var* ICV controls the number of threads requested for encountered parallel regions; there is one copy of this ICV per task. The *max-active-levels-var* ICV controls the maximum number of nested active parallel regions; there is one copy of this ICV for the whole program.

In the following example, the value of the *nthreads-var* ICV is changed via a call to **`omp_set_num_threads`**. The new value of *nthreads-var* applies only to the implicit tasks that execute the parallel region and make the call to **`omp_set_num_threads`**. The *max-active-levels-var* ICV is global; so its value is the same for all tasks.

A.4 Internal Control Variables

According to Section 2.3 on page 28, an OpenMP implementation must act as if there are ICVs that control the behavior of the program. This example illustrates two ICVs, *nthreads-var* and *max_active-levels-var*. The *nthreads-var* ICV controls the number of threads requested for encountered parallel regions; there is one copy of this ICV per task. The *max-active-levels-var* ICV controls the maximum number of nested active parallel regions; there is one copy of this ICV for the whole program.

In the following example, the value of the *nthreads-var* ICV is changed via a call to **`omp_set_num_threads`**. The new value of *nthreads-var* applies only to the implicit tasks that execute the parallel region and make the call to **`omp_set_num_threads`**. The *max-active-levels-var* ICV is global; so its value is the same for all tasks.

162 OpenMP API • Version 3.0 May 2008

C/C++

Example A.4.1c

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

int main (void)
{
  omp_set_nested(1);
  omp_set_max_active_levels(8);
  omp_set_dynamic(0);
  omp_set_num_threads(2);
  #pragma omp parallel
    {
      omp_set_num_threads(3);

      #pragma omp parallel
        {
          omp_set_num_threads(4);
          #pragma omp single
            {
              /*
               * The following should print:
               * Inner: max_act_lev=8, num_thds=3, max_thds=4
               * Inner: max_act_lev=8, num_thds=3, max_thds=4
               */
              printf ("Inner: max_act_lev=%d, num_thds=%d, max_thds=%d\n",
              omp_get_max_active_levels(), omp_get_num_threads(),
              omp_get_max_threads());
            }
        }

      #pragma omp barrier
      #pragma omp single
        {
          /*
           * The following should print:
           * Outer: max_act_lev=8, num_thds=2, max_thds=3
           */
          printf ("Outer: max_act_lev=%d, num_thds=%d, max_thds=%d\n",
                  omp_get_max_active_levels(), omp_get_num_threads(),
                  omp_get_max_threads());
        }
    }
}
```

C/C++

C/C++

Example A.4.1c

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

int main (void)
{
  omp_set_nested(1);
  omp_set_max_active_levels(8);
  omp_set_dynamic(0);
  omp_set_num_threads(2);
  #pragma omp parallel
    {
      omp_set_num_threads(3);

      #pragma omp parallel
        {
          omp_set_num_threads(4);
          #pragma omp single
            {
              /*
               * The following should print:
               * Inner: max_act_lev=8, num_thds=3, max_thds=4
               * Inner: max_act_lev=8, num_thds=3, max_thds=4
               */
              printf ("Inner: max_act_lev=%d, num_thds=%d, max_thds=%d\n",
              omp_get_max_active_levels(), omp_get_num_threads(),
              omp_get_max_threads());
            }
        }

      #pragma omp barrier
      #pragma omp single
        {
          /*
           * The following should print:
           * Outer: max_act_lev=8, num_thds=2, max_thds=3
           */
          printf ("Outer: max_act_lev=%d, num_thds=%d, max_thds=%d\n",
                  omp_get_max_active_levels(), omp_get_num_threads(),
                  omp_get_max_threads());
        }
    }
}
```

C/C++

Appendix A Examples 163

Fortran

Example A.4.1f

```
      program icv
      use omp_lib

      call omp_set_nested(.true.)
      call omp_set_max_active_levels(8)
      call omp_set_dynamic(.false.)
      call omp_set_num_threads(2)

!$omp parallel
      call omp_set_num_threads(3)

!$omp parallel
      call omp_set_num_threads(4)
!$omp single!
!      The following should print:
!      Inner: max_act_lev= 8 , num_thds= 3 , max_thds= 4
!      Inner: max_act_lev= 8 , num_thds= 3 , max_thds= 4
       print *, ("Inner: max_act_lev=", omp_get_max_active_levels(),
     &           ", num_thds=", omp_get_num_threads(),
     &           ", max_thds=", omp_get_max_threads())
!$omp end single
!$omp end parallel

!$omp barrier
!$omp single
!      The following should print:
!      Outer: max_act_lev= 8 , num_thds= 2 , max_thds= 3
       print *, ("Outer: max_act_lev=", omp_get_max_active_levels(),
     &           ", num_thds=", omp_get_num_threads(),
     &           ", max_thds=", omp_get_max_threads())
!$omp end single
!$omp end parallel
       end
```

Fortran

A.5 The `parallel` Construct

The **`parallel`** construct (Section 2.4 on page 32) 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:

Fortran

Example A.4.1f

```
      program icv
      use omp_lib

      call omp_set_nested(.true.)
      call omp_set_max_active_levels(8)
      call omp_set_dynamic(.false.)
      call omp_set_num_threads(2)

!$omp parallel
      call omp_set_num_threads(3)

!$omp parallel
      call omp_set_num_threads(4)
!$omp single!
!      The following should print:
!      Inner: max_act_lev= 8 , num_thds= 3 , max_thds= 4
!      Inner: max_act_lev= 8 , num_thds= 3 , max_thds= 4
       print *, ("Inner: max_act_lev=", omp_get_max_active_levels(),
     &           ", num_thds=", omp_get_num_threads(),
     &           ", max_thds=", omp_get_max_threads())
!$omp end single
!$omp end parallel

!$omp barrier
!$omp single
!      The following should print:
!      Outer: max_act_lev= 8 , num_thds= 2 , max_thds= 3
       print *, ("Outer: max_act_lev=", omp_get_max_active_levels(),
     &           ", num_thds=", omp_get_num_threads(),
     &           ", max_thds=", omp_get_max_threads())
!$omp end single
!$omp end parallel
       end
```

Fortran

A.5 The `parallel` Construct

The **`parallel`** construct (Section 2.4 on page 32) 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:

164 OpenMP API • Version 3.0 May 2008

C/C++

Example A.5.1c

```
#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;
}
```

C/C++

C/C++

Example A.5.1c

```
#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;
}
```

C/C++

Appendix A Examples **165**

Fortran

Example A.5.1f

```fortran
      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 A5
          REAL ARRAY(10000)
          CALL SUB(ARRAY, 10000)
      END PROGRAM A5
```

Fortran

A.6 The `num_threads` Clause

The following example demonstrates the **`num_threads`** clause (Section 2.4 on page 32). The parallel region is executed with a maximum of 10 threads.

Fortran

Example A.5.1f

```fortran
      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 A5
          REAL ARRAY(10000)
          CALL SUB(ARRAY, 10000)
      END PROGRAM A5
```

Fortran

A.6 The `num_threads` Clause

The following example demonstrates the **`num_threads`** clause (Section 2.4 on page 32). The parallel region is executed with a maximum of 10 threads.

166 OpenMP API • Version 3.0 May 2008

C/C++

Example A.6.1c

```
#include <omp.h>
int main()
{
  omp_set_dynamic(1);

  #pragma omp parallel num_threads(10)
  {
    /* do work here */
  }
  return 0;
}
```

C/C++

Fortran

Example A.6.1f

```
      PROGRAM A6
        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 A6
```

Fortran

Fortran

A.7 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 38. The following example contains correct usages of loop constructs:

C/C++

Example A.6.1c

```
#include <omp.h>
int main()
{
  omp_set_dynamic(1);

  #pragma omp parallel num_threads(10)
  {
    /* do work here */
  }
  return 0;
}
```

C/C++

Fortran

Example A.6.1f

```
      PROGRAM A6
        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 A6
```

Fortran

Fortran

A.7 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 38. The following example contains correct usages of loop constructs:

Appendix A Examples 167

Example A.7.1f

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

      SUBROUTINE A7_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 A7_GOOD
```

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

Example A.7.2f

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

      SUBROUTINE A7_WRONG
        INTEGER I, J

        DO 100 I = 1,10
!$OMP     DO
          DO 100 J = 1,10
            CALL WORK(I,J)
100     CONTINUE
!$OMP   ENDDO
      END SUBROUTINE A7_WRONG
```

Fortran

Example A.7.1f

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

      SUBROUTINE A7_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 A7_GOOD
```

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

Example A.7.2f

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

      SUBROUTINE A7_WRONG
        INTEGER I, J

        DO 100 I = 1,10
!$OMP     DO
          DO 100 J = 1,10
            CALL WORK(I,J)
100     CONTINUE
!$OMP   ENDDO
      END SUBROUTINE A7_WRONG
```

Fortran

168 OpenMP API • Version 3.0 May 2008

A.8 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 38 and Section 2.9.1 on page 77). In the following example of a sequential loop in a **`parallel`** construct the loop iteration variable *I* will be private.

Example A.8.1f

```
SUBROUTINE A8_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 A8_1
```


Fortran

A.8 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 38 and Section 2.9.1 on page 77). In the following example of a sequential loop in a parallel construct the loop iteration variable I will be private.

Example A.8.1f

SUBROUTINE A8_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 A8_1

Appendix A Examples 169

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

Example A.8.2f

```
SUBROUTINE A8_2(A,B,N,I1,I2)
REAL A(*), B(*)
INTEGER I1, I2, N

!$OMP PARALLEL SHARED(A,B,I1,I2)
!$OMP SECTIONS
!$OMP SECTION
     DO I1 = I1, N
       IF (A(I1).NE.0.0) EXIT
     ENDDO
!$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 A8_2
```

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

Fortran

A.9 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 38) to avoid the implied barrier at the end of the loop construct, as follows:

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

Example A.8.2f

```
SUBROUTINE A8_2(A,B,N,I1,I2)
REAL A(*), B(*)
INTEGER I1, I2, N

!$OMP PARALLEL SHARED(A,B,I1,I2)
!$OMP SECTIONS
!$OMP SECTION
     DO I1 = I1, N
       IF (A(I1).NE.0.0) EXIT
     ENDDO
!$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 A8_2
```

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

Fortran

A.9 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 38) to avoid the implied barrier at the end of the loop construct, as follows:

170 OpenMP API • Version 3.0 May 2008

C/C++

Example A.9.1c

```c
#include <math.h>

void a9(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]);
  }
}
```

C/C++

Fortran

Example A.9.1f

```fortran
        SUBROUTINE A9(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 A9
```

Fortran

C/C++

Example A.9.1c

```c
#include <math.h>

void a9(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]);
  }
}
```

C/C++

Fortran

Example A.9.1f

```fortran
        SUBROUTINE A9(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 A9
```

Fortran

Appendix A Examples 171

The following examples show the use of **`nowait`** with static scheduling.

C/C++

Example A.9.2c

```c
#include <math.h>
void a92(int n, float *a, float *b, float *c, float *y, float *z)
{
   int i;
#pragma omp parallel
   {
#pragma omp for schedule(static) nowait
   for (i=0; i<n; i++)
      c[i] = (a[i] + b[i]) / 2.0;
#pragma omp for schedule(static) nowait
   for (i=0; i<n; i++)
      z[i] = sqrt(c[i]);
#pragma omp for schedule(static) nowait
   for (i=1; i<=n; i++)
      y[i] = z[i-1] + a[i];
   }
}
```

C/C++

Fortran

Example A.9.2f

```fortran
   SUBROUTINE A92(N, A, B, C, Y, Z)
   INTEGER N
   REAL A(*), B(*), C(*), Y(*), Z(*)
   INTEGER I
!$OMP PARALLEL
!$OMP DO SCHEDULE(STATIC)
   DO I=1,N
      C(I) = (A(I) + B(I)) / 2.0
   ENDDO
!$OMP END DO NOWAIT
!$OMP DO SCHEDULE(STATIC)
   DO I=1,N
      Z(I) = SQRT(C(I))
   ENDDO
!$OMP END DO NOWAIT
!$OMP DO SCHEDULE(STATIC)
   DO I=2,N+1
      Y(I) = Z(I-1) + A(I)
   ENDDO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
   END SUBROUTINE A92
```

Fortran

The following examples show the use of **`nowait`** with static scheduling.

C/C++

Example A.9.2c

```c
#include <math.h>
void a92(int n, float *a, float *b, float *c, float *y, float *z)
{
   int i;
#pragma omp parallel
   {
#pragma omp for schedule(static) nowait
   for (i=0; i<n; i++)
      c[i] = (a[i] + b[i]) / 2.0;
#pragma omp for schedule(static) nowait
   for (i=0; i<n; i++)
      z[i] = sqrt(c[i]);
#pragma omp for schedule(static) nowait
   for (i=1; i<=n; i++)
      y[i] = z[i-1] + a[i];
   }
}
```

C/C++

Fortran

Example A.9.2f

```fortran
   SUBROUTINE A92(N, A, B, C, Y, Z)
   INTEGER N
   REAL A(*), B(*), C(*), Y(*), Z(*)
   INTEGER I
!$OMP PARALLEL
!$OMP DO SCHEDULE(STATIC)
   DO I=1,N
      C(I) = (A(I) + B(I)) / 2.0
   ENDDO
!$OMP END DO NOWAIT
!$OMP DO SCHEDULE(STATIC)
   DO I=1,N
      Z(I) = SQRT(C(I))
   ENDDO
!$OMP END DO NOWAIT
!$OMP DO SCHEDULE(STATIC)
   DO I=2,N+1
      Y(I) = Z(I-1) + A(I)
   ENDDO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
   END SUBROUTINE A92
```

Fortran

44 **172** OpenMP API • Version 3.0 May 2008

A.10 The `collapse` clause

In the next example, the loops over *k* and *j* are collapsed and their iteration space is executed by all threads of the current team.

Fortran

Example A.10.1f

```fortran
      subroutine sub()
!$omp do collapse(2) private(i,j,k)
       do k = kl, ku, ks
         do j = jl, ju, js
           do i = il, iu, is
             call bar(a,i,j,k)
          enddo
        enddo
      enddo
!$omp end do
      end subroutine
```

Fortran

In the next example, the loops over *k* and *j* are collapsed and their iteration space is executed by all threads of the current team. The example prints: 2   3.

Fortran

Example A.10.2f

```fortran
      program test
!$omp parallel
!$omp do private(j,k) collapse(2) lastprivate(jlast, klast)
      do k = 1,2
        do j = 1,3
          jlast=j
          klast=k
        enddo
      enddo
!$omp end do
!$omp single
                print *, klast, jlast
!$omp end single
!$omp end parallel
      end program test
```

Fortran

A.10 The `collapse` clause

In the next example, the loops over *k* and *j* are collapsed and their iteration space is executed by all threads of the current team.

Fortran

Example A.10.1f

```fortran
      subroutine sub()
!$omp do collapse(2) private(i,j,k)
       do k = kl, ku, ks
         do j = jl, ju, js
           do i = il, iu, is
             call bar(a,i,j,k)
          enddo
        enddo
      enddo
!$omp end do
      end subroutine
```

Fortran

In the next example, the loops over *k* and *j* are collapsed and their iteration space is executed by all threads of the current team. The example prints: 2   3.

Fortran

Example A.10.2f

```fortran
      program test
!$omp parallel
!$omp do private(j,k) collapse(2) lastprivate(jlast, klast)
      do k = 1,2
        do j = 1,3
          jlast=j
          klast=k
        enddo
      enddo
!$omp end do
!$omp single
                print *, klast, jlast
!$omp end single
!$omp end parallel
      end program test
```

Fortran

Appendix A Examples 173

The next example illustrates use of the **`ordered`** construct with the **`collapse`** construct. Since both loops are collapsed into one, the **`ordered`** construct has to be inside all loops associated with the **`do`** construct. Since an iteration may not execute more than one **`ordered`** region this program would be wrong without the **`collapse`**(2) clause. The code prints

0 1 1

0 1 2

0 2 1

1 2 2

1 3 1

1 3 2

Fortran

Example A.10.3f

```
      program test
      include 'omp_lib.h'
!$omp parallel num_threads(2)
!$omp do collapse(2) ordered private(j,k) schedule(static,3)
      do k = 1,3
        do j = 1,2
!$omp ordered
          print *, omp_get_thread_num(), k, j
!$omp end ordered
          call work(a,j,k)
        enddo
      enddo
!$omp end do
!$omp end parallel
      end program test
```

Fortran

A.11 The `parallel sections` Construct

In the following example (for Section 2.5.2 on page 47) routines *xaxis*, *yaxis*, and *zaxis* can be executed concurrently. The first **`section`** directive is optional. Note that all **`section`** directives need to appear in the **`parallel sections`** construct.

The next example illustrates use of the **`ordered`** construct with the **`collapse`** construct. Since both loops are collapsed into one, the **`ordered`** construct has to be inside all loops associated with the **`do`** construct. Since an iteration may not execute more than one **`ordered`** region this program would be wrong without the **`collapse`**(2) clause. The code prints

0 1 1

0 1 2

0 2 1

1 2 2

1 3 1

1 3 2

Fortran

Example A.10.3f

```
      program test
      include 'omp_lib.h'
!$omp parallel num_threads(2)
!$omp do collapse(2) ordered private(j,k) schedule(static,3)
      do k = 1,3
        do j = 1,2
!$omp ordered
          print *, omp_get_thread_num(), k, j
!$omp end ordered
          call work(a,j,k)
        enddo
      enddo
!$omp end do
!$omp end parallel
      end program test
```

Fortran

A.11 The `parallel sections` Construct

In the following example (for Section 2.5.2 on page 47) routines *xaxis*, *yaxis*, and *zaxis* can be executed concurrently. The first **`section`** directive is optional. Note that all **`section`** directives need to appear in the **`parallel sections`** construct.

174 OpenMP API • Version 3.0 May 2008

C/C++

Example A.11.1c

```c
void XAXIS();
void YAXIS();
void ZAXIS();

void a11()
{
  #pragma omp parallel sections
  {
    #pragma omp section
      XAXIS();

    #pragma omp section
      YAXIS();

    #pragma omp section
      ZAXIS();
  }
}
```

C/C++

Fortran

Example A.11.1f

```fortran
      SUBROUTINE A11()

!$OMP PARALLEL SECTIONS
!$OMP SECTION
        CALL XAXIS()

!$OMP SECTION
        CALL YAXIS()

!$OMP SECTION
        CALL ZAXIS()

!$OMP END PARALLEL SECTIONS

      END SUBROUTINE A11
```

Fortran

C/C++

Example A.11.1c

```c
void XAXIS();
void YAXIS();
void ZAXIS();

void a11()
{
  #pragma omp parallel sections
  {
    #pragma omp section
      XAXIS();

    #pragma omp section
      YAXIS();

    #pragma omp section
      ZAXIS();
  }
}
```

C/C++

Fortran

Example A.11.1f

```fortran
      SUBROUTINE A11()

!$OMP PARALLEL SECTIONS
!$OMP SECTION
        CALL XAXIS()

!$OMP SECTION
        CALL YAXIS()

!$OMP SECTION
        CALL ZAXIS()

!$OMP END PARALLEL SECTIONS

      END SUBROUTINE A11
```

Fortran

Appendix A Examples 175

A.12 The `single` Construct

The following example demonstrates the **`single`** construct (Section 2.5.3 on page 49). 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.

C/C++

Example A.12.1c

```
#include <stdio.h>

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

void a12()
{
  #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();
  }
}
```

C/C++

A.12 The `single` Construct

The following example demonstrates the **`single`** construct (Section 2.5.3 on page 49). 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.

C/C++

Example A.12.1c

```
#include <stdio.h>

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

void a12()
{
  #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();
  }
}
```

C/C++

176 OpenMP API • Version 3.0 May 2008

Fortran

Example A.12.1f

```
      SUBROUTINE WORK1()
      END SUBROUTINE WORK1

      SUBROUTINE WORK2()
      END SUBROUTINE WORK2

      PROGRAM A12
!$OMP PARALLEL

!$OMP SINGLE
        print *, "Beginning work1."
!$OMP END SINGLE

        CALL WORK1()

!$OMP SINGLE
        print *, "Finishing work1."
!$OMP END SINGLE

!$OMP SINGLE
        print *, "Finished work1 and beginning work2."
!$OMP END SINGLE NOWAIT

        CALL WORK2()

!$OMP END PARALLEL

      END PROGRAM A12
```

Fortran

A.13 Tasking Constructs

The following example shows how to traverse a tree-like structure using explicit tasks. Note that the *traverse* function should be called from within a parallel region for the different specified tasks to be executed in parallel. Also, note that the tasks will be executed in no specified order because there are no synchronization directives. Thus, assuming that the traversal will be done in post order, as in the sequential code, is wrong.

Fortran

Example A.12.1f

```
      SUBROUTINE WORK1()
      END SUBROUTINE WORK1

      SUBROUTINE WORK2()
      END SUBROUTINE WORK2

      PROGRAM A12
!$OMP PARALLEL

!$OMP SINGLE
        print *, "Beginning work1."
!$OMP END SINGLE

        CALL WORK1()

!$OMP SINGLE
        print *, "Finishing work1."
!$OMP END SINGLE

!$OMP SINGLE
        print *, "Finished work1 and beginning work2."
!$OMP END SINGLE NOWAIT

        CALL WORK2()

!$OMP END PARALLEL

      END PROGRAM A12
```

Fortran

A.13 Tasking Constructs

The following example shows how to traverse a tree-like structure using explicit tasks. Note that the *traverse* function should be called from within a parallel region for the different specified tasks to be executed in parallel. Also, note that the tasks will be executed in no specified order because there are no synchronization directives. Thus, assuming that the traversal will be done in post order, as in the sequential code, is wrong.

Appendix A Examples 177

C/C++

Example A.13.1c

```c
struct node {
  struct node *left;
  struct node *right;
};
extern void process(struct node *);
void traverse( struct node *p ) {
  if (p->left)
#pragma omp task   // p is firstprivate by default
      traverse(p->left);
  if (p->right)
#pragma omp task    // p is firstprivate by default
      traverse(p->right);
  process(p);
}
```

C/C++

Fortran

Example A.13.1f

```fortran
       RECURSIVE SUBROUTINE traverse ( P )
          TYPE Node
            TYPE(Node), POINTER :: left, right
          END TYPE Node
          TYPE(Node) :: P
          IF (associated(P%left)) THEN
                  !$OMP TASK    ! P is firstprivate by default
                      call traverse(P%left)
                  !$OMP END TASK
          ENDIF
          IF (associated(P%right)) THEN
                  !$OMP TASK     ! P is firstprivate by default
                      call traverse(P%right)
                  !$OMP END TASK
          ENDIF
          CALL process ( P )
        END SUBROUTINE
```

Fortran

C/C++

Example A.13.1c

```c
struct node {
  struct node *left;
  struct node *right;
};
extern void process(struct node *);
void traverse( struct node *p ) {
  if (p->left)
#pragma omp task   // p is firstprivate by default
      traverse(p->left);
  if (p->right)
#pragma omp task    // p is firstprivate by default
      traverse(p->right);
  process(p);
}
```

C/C++

Fortran

Example A.13.1f

```fortran
       RECURSIVE SUBROUTINE traverse ( P )
          TYPE Node
            TYPE(Node), POINTER :: left, right
          END TYPE Node
          TYPE(Node) :: P
          IF (associated(P%left)) THEN
                  !$OMP TASK    ! P is firstprivate by default
                      call traverse(P%left)
                  !$OMP END TASK
          ENDIF
          IF (associated(P%right)) THEN
                  !$OMP TASK     ! P is firstprivate by default
                      call traverse(P%right)
                  !$OMP END TASK
          ENDIF
          CALL process ( P )
        END SUBROUTINE
```

Fortran

37 **178** OpenMP API • Version 3.0 May 2008

In the next example, we force a postorder traversal of the tree by adding a **`taskwait`** directive. Now, we can safely assume that the left and right sons have been executed before we process the current node.

C/C++

Example A.13.2c

```c
struct node {
  struct node *left;
  struct node *right;
};
extern void process(struct node *);
void postorder_traverse( struct node *p ) {
    if (p->left)
       #pragma omp task    // p is firstprivate by default
           postorder_traverse(p->left);
    if (p->right)
       #pragma omp task   // p is firstprivate by default
           postorder_traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

C/C++

Fortran

Example A.13.2f

```fortran
         RECURSIVE SUBROUTINE traverse ( P )
            TYPE Node
                TYPE(Node), POINTER :: left, right
             END TYPE Node
             TYPE(Node) :: P
             IF (associated(P%left)) THEN
                  !$OMP TASK   ! P is firstprivate by default
                      call traverse(P%left)
                  !$OMP END TASK
             ENDIF
             IF (associated(P%right)) THEN
                  !$OMP TASK    ! P is firstprivate by default
                      call traverse(P%right)
                  !$OMP END TASK
             ENDIF
             !$OMP TASKWAIT
             CALL process ( P )
          END SUBROUTINE
```

Fortran

In the next example, we force a postorder traversal of the tree by adding a **`taskwait`** directive. Now, we can safely assume that the left and right sons have been executed before we process the current node.

C/C++

Example A.13.2c

```c
struct node {
  struct node *left;
  struct node *right;
};
extern void process(struct node *);
void postorder_traverse( struct node *p ) {
    if (p->left)
       #pragma omp task    // p is firstprivate by default
           postorder_traverse(p->left);
    if (p->right)
       #pragma omp task   // p is firstprivate by default
           postorder_traverse(p->right);
    #pragma omp taskwait
    process(p);
}
```

C/C++

Fortran

Example A.13.2f

```fortran
         RECURSIVE SUBROUTINE traverse ( P )
            TYPE Node
                TYPE(Node), POINTER :: left, right
             END TYPE Node
             TYPE(Node) :: P
             IF (associated(P%left)) THEN
                  !$OMP TASK   ! P is firstprivate by default
                      call traverse(P%left)
                  !$OMP END TASK
             ENDIF
             IF (associated(P%right)) THEN
                  !$OMP TASK    ! P is firstprivate by default
                      call traverse(P%right)
                  !$OMP END TASK
             ENDIF
             !$OMP TASKWAIT
             CALL process ( P )
          END SUBROUTINE
```

Fortran

Appendix A Examples 179

The following example demonstrates how to use the **task** construct to process elements of a linked list in parallel. The pointer *p* is firstprivate by default on the **task** construct so it is not necessary to specify it in a **firstprivate** clause.

C/C++

Example A.13.3c

```
typedef struct node node;
struct node {
      int data;
      node * next;
};

void process(node * p)
{
    /* do work here */
}
void increment_list_items(node * head)
{
    #pragma omp parallel
    {
        #pragma omp single
            {
               node * p = head;
               while (p) {
                    #pragma omp task
                     // p is firstprivate by default
                           process(p);
                     p = p->next;
                  }
            }
     }
}
```

C/C++

The following example demonstrates how to use the **task** construct to process elements of a linked list in parallel. The pointer *p* is firstprivate by default on the **task** construct so it is not necessary to specify it in a **firstprivate** clause.

C/C++

Example A.13.3c

```
typedef struct node node;
struct node {
      int data;
      node * next;
};

void process(node * p)
{
    /* do work here */
}
void increment_list_items(node * head)
{
    #pragma omp parallel
    {
        #pragma omp single
            {
               node * p = head;
               while (p) {
                    #pragma omp task
                     // p is firstprivate by default
                           process(p);
                     p = p->next;
                  }
            }
     }
}
```

C/C++

180 OpenMP API • Version 3.0 May 2008

Fortran

Example A.13.3f

```
MODULE LIST
   TYPE NODE
       INTEGER :: PAYLOAD
       TYPE (NODE), POINTER :: NEXT
   END TYPE NODE
CONTAINS
    SUBROUTINE PROCESS(p)
       TYPE (NODE), POINTER :: P
           ! do work here
    END SUBROUTINE
    SUBROUTINE INCREMENT_LIST_ITEMS (HEAD)
        TYPE (NODE), POINTER :: HEAD
        TYPE (NODE), POINTER :: P
        !$OMP PARALLEL PRIVATE(P)
           !$OMP SINGLE
                P => HEAD
                DO
                   !$OMP TASK
                       ! P is firstprivate by default
                       CALL PROCESS(P)
                   !$OMP END TASK
                   P => P%NEXT
                   IF ( .NOT. ASSOCIATED (P) ) EXIT
                END DO
          !$OMP END SINGLE
       !$OMP END PARALLEL
    END SUBROUTINE
 END MODULE
```

Fortran

Fortran

Example A.13.3f

```
MODULE LIST
   TYPE NODE
       INTEGER :: PAYLOAD
       TYPE (NODE), POINTER :: NEXT
   END TYPE NODE
CONTAINS
    SUBROUTINE PROCESS(p)
       TYPE (NODE), POINTER :: P
           ! do work here
    END SUBROUTINE
    SUBROUTINE INCREMENT_LIST_ITEMS (HEAD)
        TYPE (NODE), POINTER :: HEAD
        TYPE (NODE), POINTER :: P
        !$OMP PARALLEL PRIVATE(P)
           !$OMP SINGLE
                P => HEAD
                DO
                   !$OMP TASK
                       ! P is firstprivate by default
                       CALL PROCESS(P)
                   !$OMP END TASK
                   P => P%NEXT
                   IF ( .NOT. ASSOCIATED (P) ) EXIT
                END DO
          !$OMP END SINGLE
       !$OMP END PARALLEL
    END SUBROUTINE
 END MODULE
```

Fortran

Appendix A Examples 181

This example calculates a Fibonacci number. If a call to this function is encountered by a single thread in a parallel region, a nested task region will be spawned to carry out the computation in parallel.

C/C++

Example A.13.4c

```
int fib(int n) {
   int i, j;
   if (n<2)
     return n;
   else {
      #pragma omp task shared(i)
         i=fib(n-1);
      #pragma omp task shared(j)
         j=fib(n-2);
      #pragma omp taskwait
         return i+j;
   }
}
```

C/C++

Fortran

Example A.13.4f

```
      RECURSIVE INTEGER FUNCTION fib(n)
      INTEGER n, i, j
      IF ( n .LT. 2) THEN
        fib = n
      ELSE
!$OMP TASK SHARED(i)
        i = fib( n-1 )
!$OMP END TASK
!$OMP TASK SHARED(j)
        j = fib( n-2 )
!$OMP END TASK
!$OMP END TASKWAIT
        fib = i+j
      END IF
      END FUNCTION
```

Fortran

Note: Fibonacci number computation is a classic computer science example for showing recursion, but it is also a classic example showing that a simple algorithm may not be efficient. A more efficient method is through the calculation of exponentiation over an integer matrix. Here we used the classic recursion algorithm for illustrative purposes.

This example calculates a Fibonacci number. If a call to this function is encountered by a single thread in a parallel region, a nested task region will be spawned to carry out the computation in parallel.

C/C++

Example A.13.4c

```
int fib(int n) {
   int i, j;
   if (n<2)
     return n;
   else {
      #pragma omp task shared(i)
         i=fib(n-1);
      #pragma omp task shared(j)
         j=fib(n-2);
      #pragma omp taskwait
         return i+j;
   }
}
```

C/C++

Fortran

Example A.13.4f

```
      RECURSIVE INTEGER FUNCTION fib(n)
      INTEGER n, i, j
      IF ( n .LT. 2) THEN
        fib = n
      ELSE
!$OMP TASK SHARED(i)
        i = fib( n-1 )
!$OMP END TASK
!$OMP TASK SHARED(j)
        j = fib( n-2 )
!$OMP END TASK
!$OMP END TASKWAIT
        fib = i+j
      END IF
      END FUNCTION
```

Fortran

Note: Fibonacci number computation is a classic computer science example for showing recursion, but it is also a classic example showing that a simple algorithm may not be efficient. A more efficient method is through the calculation of exponentiation over an integer matrix. Here we used the classic recursion algorithm for illustrative purposes.

182 OpenMP API • Version 3.0 May 2008

The following example demonstrates a way to generate a large number of tasks with one thread and execute them with the threads in the parallel team. While generating these tasks, the implementation may reach its limit on unassigned tasks. If it does, the implementation is allowed to cause the thread executing the task generating loop to suspend its task at the task scheduling point in the **task** directive, and start executing unassigned tasks. Once the number of unassigned tasks is sufficiently low, the thread may resume execution of the task generating loop.

C/C++

Example A.13.5c

```
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];
extern void process(double);

int main() {
#pragma omp parallel
  {
    #pragma omp single
    {
      int i;
      for (i=0; i<LARGE_NUMBER; i++)
             #pragma omp task    // i is firstprivate, item is shared
                  process(item[i]);
    }
  }
}
```

C/C++

Fortran

Example A.13.5f

```
       real*8 item(10000000)
       integer i

!$omp parallel
!$omp single ! loop iteration variable i is private
       do i=1,10000000
!$omp task
         ! i is firstprivate, item is shared
          call process(item(i))
!$omp end task
       end do
!$omp end single
!$omp end parallel
       end
```

Fortran

The following example demonstrates a way to generate a large number of tasks with one thread and execute them with the threads in the parallel team. While generating these tasks, the implementation may reach its limit on unassigned tasks. If it does, the implementation is allowed to cause the thread executing the task generating loop to suspend its task at the task scheduling point in the **task** directive, and start executing unassigned tasks. Once the number of unassigned tasks is sufficiently low, the thread may resume execution of the task generating loop.

C/C++

Example A.13.5c

```
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];
extern void process(double);

int main() {
#pragma omp parallel
  {
    #pragma omp single
    {
      int i;
      for (i=0; i<LARGE_NUMBER; i++)
             #pragma omp task    // i is firstprivate, item is shared
                  process(item[i]);
    }
  }
}
```

C/C++

Fortran

Example A.13.5f

```
       real*8 item(10000000)
       integer i

!$omp parallel
!$omp single ! loop iteration variable i is private
       do i=1,10000000
!$omp task
         ! i is firstprivate, item is shared
          call process(item(i))
!$omp end task
       end do
!$omp end single
!$omp end parallel
       end
```

Fortran

Appendix A Examples 183

The following example is the same as the previous one, except that the tasks are generated in an untied task. While generating the tasks, the implementation may reach its limit on unassigned tasks. If it does, the implementation is allowed to cause the thread executing the task generating loop to suspend its task at the task scheduling point in the **`task`** directive, and start executing unassigned tasks. If that thread begins execution of a task that takes a long time to complete, the other threads may complete all the other tasks before it is finished.

In this case, since the loop is in an untied task, any other thread is eligible to resume the task generating loop. In the previous examples, the other threads would be forced to wait idly until the generating thread finishes its long task, since the task generating loop was in a tied task.

C/C++

Example A.13.6c

```
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];
extern void process(double);
int main() {
#pragma omp parallel
  {
    #pragma omp single
    {
      int i;
      #pragma omp task untied
      // i is firstprivate, item is shared
      {
         for (i=0; i<LARGE_NUMBER; i++)
             #pragma omp task
                  process(item[i]);
      }
    }
  }
}
```

C/C++

The following example is the same as the previous one, except that the tasks are generated in an untied task. While generating the tasks, the implementation may reach its limit on unassigned tasks. If it does, the implementation is allowed to cause the thread executing the task generating loop to suspend its task at the task scheduling point in the **`task`** directive, and start executing unassigned tasks. If that thread begins execution of a task that takes a long time to complete, the other threads may complete all the other tasks before it is finished.

In this case, since the loop is in an untied task, any other thread is eligible to resume the task generating loop. In the previous examples, the other threads would be forced to wait idly until the generating thread finishes its long task, since the task generating loop was in a tied task.

C/C++

Example A.13.6c

```
#define LARGE_NUMBER 10000000
double item[LARGE_NUMBER];
extern void process(double);
int main() {
#pragma omp parallel
  {
    #pragma omp single
    {
      int i;
      #pragma omp task untied
      // i is firstprivate, item is shared
      {
         for (i=0; i<LARGE_NUMBER; i++)
             #pragma omp task
                  process(item[i]);
      }
    }
  }
}
```

C/C++

33 **184** OpenMP API • Version 3.0 May 2008

Fortran

Example A.13.6f

```
      real*8 item(10000000)
!$omp parallel
!$omp single
!$omp task untied
      ! loop iteration variable i is private
      do i=1,10000000
!$omp task ! i is firstprivate, item is shared
         call process(item(i))
!$omp end task
      end do
!$omp end task
!$omp end single
!$omp end parallel
      end
```

Fortran

The following two examples demonstrate how the scheduling rules illustrated in Section 2.7.1 on page 62 affect the usage of threadprivate variables in tasks. The value of a threadprivate variable will change across task scheduling points if the executing thread executes a part of another schedulable task that modifies the variable. In tied tasks, the user can control where task scheduling points appear in the code.

A single thread may execute both of the task regions that modify *tp*. The parts of these task regions in which *tp* is modified may be executed in any order so the resulting value of *var* can be either 1 or 2.

Fortran

Example A.13.6f

```
      real*8 item(10000000)
!$omp parallel
!$omp single
!$omp task untied
      ! loop iteration variable i is private
      do i=1,10000000
!$omp task ! i is firstprivate, item is shared
         call process(item(i))
!$omp end task
      end do
!$omp end task
!$omp end single
!$omp end parallel
      end
```

Fortran

The following two examples demonstrate how the scheduling rules illustrated in Section 2.7.1 on page 62 affect the usage of threadprivate variables in tasks. The value of a threadprivate variable will change across task scheduling points if the executing thread executes a part of another schedulable task that modifies the variable. In tied tasks, the user can control where task scheduling points appear in the code.

A single thread may execute both of the task regions that modify *tp*. The parts of these task regions in which *tp* is modified may be executed in any order so the resulting value of *var* can be either 1 or 2.

Appendix A Examples **185**

C/C++

Example A.13.7c

```c
int tp;
#pragma omp threadprivate(tp)
int var;
void work()
{
#pragma omp task
    {
        /* do work here */
#pragma omp task
        {
            tp = 1;
            /* do work here */
#pragma omp task
            {
                /* no modification of tp */
            }
            var = tp; //value of tp can be 1 or 2
        }
        tp = 2;
    }
}
```

C/C++

Fortran

Example A.13.7f

```fortran
      module example
      integer tp
!$omp threadprivate(tp)
      integer var
      contains
      subroutine work
      use globals
!$omp task
         ! do work here
!$omp task
         tp = 1
         ! do work here
!$omp task
           ! no modification of tp
!$omp end task
         var = tp    ! value of var can be 1 or 2
!$omp end task
        tp = 2
!$omp end task
      end subroutine
      end module
```

Fortran

C/C++

Example A.13.7c

```c
int tp;
#pragma omp threadprivate(tp)
int var;
void work()
{
#pragma omp task
    {
        /* do work here */
#pragma omp task
        {
            tp = 1;
            /* do work here */
#pragma omp task
            {
                /* no modification of tp */
            }
            var = tp; //value of tp can be 1 or 2
        }
        tp = 2;
    }
}
```

C/C++

Fortran

Example A.13.7f

```fortran
      module example
      integer tp
!$omp threadprivate(tp)
      integer var
      contains
      subroutine work
      use globals
!$omp task
         ! do work here
!$omp task
         tp = 1
         ! do work here
!$omp task
           ! no modification of tp
!$omp end task
         var = tp    ! value of var can be 1 or 2
!$omp end task
        tp = 2
!$omp end task
      end subroutine
      end module
```

Fortran

186 OpenMP API • Version 3.0 May 2008

In this example, scheduling rules prohibit a single thread from scheduling a new task that modifies *tp* while another such task region is suspended. Therefore, the value written will persist across the task scheduling point.

C/C++

Example A.13.8c

```c
#include <omp.h>
int tp;
#pragma omp threadprivate(tp)
int var;
void work()
{
#pragma omp parallel
    {
        /* do work here */
#pragma omp task
        {
            tp++;
            /* do work here */
#pragma omp task
            {
                /* do work here but don't modify tp */
            }
            var = tp; //Value does not change after write above
        }
    }
}
```

C/C++

Fortran

Example A.13.8f

```fortran
      module example
      integer tp
!$omp threadprivate(tp)
      integer var
      contains
      subroutine work
!$omp parallel
         ! do work here
!$omp task
         tp = tp + 1
         ! do work here
!$omp task
           ! do work here but don't modify tp
!$omp end task
         var = tp    ! value does not change after write above
!$omp end task
!$omp end parallel
      end subroutine
      end module
```

Fortran

In this example, scheduling rules prohibit a single thread from scheduling a new task that modifies *tp* while another such task region is suspended. Therefore, the value written will persist across the task scheduling point.

C/C++

Example A.13.8c

```c
#include <omp.h>
int tp;
#pragma omp threadprivate(tp)
int var;
void work()
{
#pragma omp parallel
    {
        /* do work here */
#pragma omp task
        {
            tp++;
            /* do work here */
#pragma omp task
            {
                /* do work here but don't modify tp */
            }
            var = tp; //Value does not change after write above
        }
    }
}
```

C/C++

Fortran

Example A.13.8f

```fortran
      module example
      integer tp
!$omp threadprivate(tp)
      integer var
      contains
      subroutine work
!$omp parallel
         ! do work here
!$omp task
         tp = tp + 1
         ! do work here
!$omp task
           ! do work here but don't modify tp
!$omp end task
         var = tp    ! value does not change after write above
!$omp end task
!$omp end parallel
      end subroutine
      end module
```

Fortran

Appendix A Examples 187

The following two examples demonstrate how the scheduling rules illustrated in Section 2.7.1 on page 62 affect the usage of locks and critical sections in tasks. If a lock is held across a task scheduling point, no attempt should be made to acquire the same lock in any code that may be interleaved. Otherwise, a deadlock is possible.

In the example below, suppose the thread executing task 1 defers task 2. When it encounters the task scheduling point at task 3, it could suspend task 1 and begin task 2 which will result in a deadlock when it tries to enter critical region 1.

C/C++

Example A.13.9c

```
void work()
{
   #pragma omp task
   { //Task 1
       #pragma omp task
       { //Task 2
            #pragma omp critical //Critical region 1
            {/*do work here */ }
       }
       #pragma omp critical //Critical Region 2
       {
           //Capture data for the following task
           #pragma omp task
           { /* do work here */ } //Task 3
       }
   }
}
```

C/C++

The following two examples demonstrate how the scheduling rules illustrated in Section 2.7.1 on page 62 affect the usage of locks and critical sections in tasks. If a lock is held across a task scheduling point, no attempt should be made to acquire the same lock in any code that may be interleaved. Otherwise, a deadlock is possible.

In the example below, suppose the thread executing task 1 defers task 2. When it encounters the task scheduling point at task 3, it could suspend task 1 and begin task 2 which will result in a deadlock when it tries to enter critical region 1.

C/C++

Example A.13.9c

```
void work()
{
   #pragma omp task
   { //Task 1
       #pragma omp task
       { //Task 2
            #pragma omp critical //Critical region 1
            {/*do work here */ }
       }
       #pragma omp critical //Critical Region 2
       {
           //Capture data for the following task
           #pragma omp task
           { /* do work here */ } //Task 3
       }
   }
}
```

C/C++

27 188 OpenMP API • Version 3.0 May 2008

Fortran

Example A.13.9f

```fortran
       module example
       contains
       subroutine work
!$omp task
       ! Task 1
!$omp task
       ! Task 2
!$omp critical
       ! Critical region 1
       ! do work here
!$omp end critical
!$omp end task
!$omp critical
       ! Critical region 2
       ! Capture data for the following task
!$omp task
       !Task 3
       ! do work here
!$omp end task
!$omp end critical
!$omp end task
      end subroutine
      end module
```

Fortran

Fortran

Example A.13.9f

```fortran
       module example
       contains
       subroutine work
!$omp task
       ! Task 1
!$omp task
       ! Task 2
!$omp critical
       ! Critical region 1
       ! do work here
!$omp end critical
!$omp end task
!$omp critical
       ! Critical region 2
       ! Capture data for the following task
!$omp task
       !Task 3
       ! do work here
!$omp end task
!$omp end critical
!$omp end task
      end subroutine
      end module
```

Fortran

Appendix A Examples 189

In the following example, *lock* is held across a task scheduling point. However, according to the scheduling restrictions outlined in Section 2.7.1 on page 62, the executing thread can't begin executing one of the non-descendant tasks that also acquires *lock* before the task region is complete. Therefore, no deadlock is possible.

C/C++

Example A.13.10c

```
#include <omp.h>
void work() {
    omp_lock_t lock;
#pragma omp parallel
    {
        int i;
#pragma omp for
        for (i = 0; i < 100; i++) {
#pragma omp task
            {
                    // lock is shared by default in the task
                    omp_set_lock(&lock);
                // Capture data for the following task
#pragma omp task
                    // Task Scheduling Point 1
                { /* do work here */ }
                omp_unset_lock(&lock);
            }
        }
    }
}
```

C/C++

In the following example, *lock* is held across a task scheduling point. However, according to the scheduling restrictions outlined in Section 2.7.1 on page 62, the executing thread can't begin executing one of the non-descendant tasks that also acquires *lock* before the task region is complete. Therefore, no deadlock is possible.

C/C++

Example A.13.10c

```
#include <omp.h>
void work() {
    omp_lock_t lock;
#pragma omp parallel
    {
        int i;
#pragma omp for
        for (i = 0; i < 100; i++) {
#pragma omp task
            {
                    // lock is shared by default in the task
                    omp_set_lock(&lock);
                // Capture data for the following task
#pragma omp task
                    // Task Scheduling Point 1
                { /* do work here */ }
                omp_unset_lock(&lock);
            }
        }
    }
}
```

C/C++

28 **190** OpenMP API • Version 3.0 May 2008

Fortran

Example A.13.10f

```fortran
       module example
       include 'omp_lib.h'
       integer (kind=omp_lock_kind) lock
       integer i
       contains
       subroutine work
!$omp parallel
     !$omp do
      do i=1,100
         !$omp task
              ! Outer task
              call omp_set_lock(lock)    ! lock is shared by
                                         ! default in the task
                     ! Capture data for the following task
                     !$omp task     ! Task Scheduling Point 1
                              ! do work here
                     !$omp end task
               call omp_unset_lock(lock)
         !$omp end task
      end do
!$omp end parallel
      end subroutine
      end module
```

Fortran

Fortran

A.14 The **workshare** Construct

The following are examples of the **workshare** construct (see Section 2.5.4 on page 51).

Fortran

Example A.13.10f

```fortran
       module example
       include 'omp_lib.h'
       integer (kind=omp_lock_kind) lock
       integer i
       contains
       subroutine work
!$omp parallel
     !$omp do
      do i=1,100
         !$omp task
              ! Outer task
              call omp_set_lock(lock)    ! lock is shared by
                                         ! default in the task
                     ! Capture data for the following task
                     !$omp task     ! Task Scheduling Point 1
                              ! do work here
                     !$omp end task
               call omp_unset_lock(lock)
         !$omp end task
      end do
!$omp end parallel
      end subroutine
      end module
```

Fortran

Fortran

A.14 The workshare Construct

The following are examples of the **workshare** construct (see Section 2.5.4 on page 51).

Appendix A Examples 191

Fortran (cont.)

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.14.1f

```
      SUBROUTINE A14_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 A14_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`.

Example A.14.2f

```
      SUBROUTINE A14_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 A14_2
```


Fortran (cont.)

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.14.1f

```
      SUBROUTINE A14_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 A14_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`.

Example A.14.2f

```
      SUBROUTINE A14_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 A14_2
```

192 OpenMP API • Version 3.0 May 2008

Fortran (cont.)

The following example shows the use of an **atomic** directive inside a **workshare** construct. The computation of **SUM(AA)** is workshared, but the update to **R** is atomic.

Example A.14.3f

```
      SUBROUTINE A14_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 A14_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:

AA = BB then

CC = DD then

EE .ne. 0 then

FF = 1 / EE then

GG = HH

Fortran (cont.)

The following example shows the use of an **atomic** directive inside a **workshare** construct. The computation of **SUM(AA)** is workshared, but the update to **R** is atomic.

Example A.14.3f

```
      SUBROUTINE A14_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 A14_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:

AA = BB then

CC = DD then

EE .ne. 0 then

FF = 1 / EE then

GG = HH

Appendix A Examples **193**

Fortran (cont.)

Example A.14.4f

```
      SUBROUTINE A14_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 A14_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.14.5f

```
      SUBROUTINE A14_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
!$OMP     END WORKSHARE
!$OMP   END PARALLEL

      END SUBROUTINE A14_5
```

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.

Fortran (cont.)

Example A.14.4f

```
      SUBROUTINE A14_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 A14_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.14.5f

```
      SUBROUTINE A14_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
!$OMP     END WORKSHARE
!$OMP   END PARALLEL

      END SUBROUTINE A14_5
```

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.

194 OpenMP API • Version 3.0 May 2008

Example A.14.6f

```
      SUBROUTINE A14_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 A14_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.14.7f

```
      SUBROUTINE A14_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)
!$OMP     END WORKSHARE
!$OMP   END PARALLEL

      END SUBROUTINE A14_7
```

Fortran

A.15 The `master` Construct

The following example demonstrates the master construct (Section 2.8.1 on page 63). 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.14.6f

```
      SUBROUTINE A14_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 A14_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.14.7f

```
      SUBROUTINE A14_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)
!$OMP     END WORKSHARE
!$OMP   END PARALLEL

      END SUBROUTINE A14_7
```

Fortran

A.15 The `master` Construct

The following example demonstrates the master construct (Section 2.8.1 on page 63). 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.

Appendix A Examples 195

C/C++

Example A.15.1c

```c
#include <stdio.h>

extern float average(float,float,float);

void a15( 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 );
      }
    }while( toobig > 0 );
  }
}
```

C/C++

C/C++

Example A.15.1c

```c
#include <stdio.h>

extern float average(float,float,float);

void a15( 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 );
      }
    }while( toobig > 0 );
  }
}
```

C/C++

36 **196** OpenMP API • Version 3.0 May 2008

Fortran

Example A.15.1f

```fortran
      SUBROUTINE A15( 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 A15
```

Fortran

A.16 The `critical` Construct

The following example includes several **`critical`** constructs (Section 2.8.2 on page 65). 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*.

Fortran

Example A.15.1f

```fortran
      SUBROUTINE A15( 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 A15
```

Fortran

A.16 The critical Construct

The following example includes several **`critical`** constructs (Section 2.8.2 on page 65). 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*.

Appendix A Examples 197

C/C++

Example A.16.1c

```c
int dequeue(float *a);
void work(int i, float *a);

void a16(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);
  }

}
```

C/C++

Fortran

Example A.16.1f

```fortran
      SUBROUTINE A16(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)

!$OMP END PARALLEL

      END SUBROUTINE A16
```

Fortran

C/C++

Example A.16.1c

```c
int dequeue(float *a);
void work(int i, float *a);

void a16(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);
  }

}
```

C/C++

Fortran

Example A.16.1f

```fortran
      SUBROUTINE A16(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)

!$OMP END PARALLEL

      END SUBROUTINE A16
```

Fortran

198 OpenMP API • Version 3.0 May 2008

A.17 worksharing Constructs Inside a `critical` Construct

The following example demonstrates using a worksharing construct inside a **`critical`** construct (see Section 2.8.2 on page 65). This example is conforming because the **`single`** region and the **`critical`** region are not closely nested (see Section 2.10 on page 104).

C/C++

Example A.17.1c

```
void a17()
{
  int i = 1;
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      #pragma omp critical (name)
      {
        #pragma omp parallel
        {
          #pragma omp single
          {
            i++;
          }
        }
      }
    }
  }
}
```

C/C++

A.17 worksharing Constructs Inside a `critical` Construct

The following example demonstrates using a worksharing construct inside a **`critical`** construct (see Section 2.8.2 on page 65). This example is conforming because the **`single`** region and the **`critical`** region are not closely nested (see Section 2.10 on page 104).

C/C++

Example A.17.1c

```
void a17()
{
  int i = 1;
  #pragma omp parallel sections
  {
    #pragma omp section
    {
      #pragma omp critical (name)
      {
        #pragma omp parallel
        {
          #pragma omp single
          {
            i++;
          }
        }
      }
    }
  }
}
```

C/C++

Appendix A Examples 199

Fortran

Example A.17.1f

```
      SUBROUTINE A17()

        INTEGER I
        I = 1

!$OMP   PARALLEL SECTIONS
!$OMP     SECTION
!$OMP       CRITICAL (NAME)
!$OMP         PARALLEL
!$OMP           SINGLE
                  I = I + 1
!$OMP           END SINGLE
!$OMP         END PARALLEL
!$OMP       END CRITICAL (NAME)
!$OMP   END PARALLEL SECTIONS
      END SUBROUTINE A17
```

Fortran

A.18 Binding of `barrier` Regions

The binding rules call for a **`barrier`** region to bind to the closest enclosing **`parallel`** region (see Section 2.8.3 on page 66).

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*.

Fortran

Example A.17.1f

```
      SUBROUTINE A17()

        INTEGER I
        I = 1

!$OMP   PARALLEL SECTIONS
!$OMP     SECTION
!$OMP       CRITICAL (NAME)
!$OMP         PARALLEL
!$OMP           SINGLE
                  I = I + 1
!$OMP           END SINGLE
!$OMP         END PARALLEL
!$OMP       END CRITICAL (NAME)
!$OMP   END PARALLEL SECTIONS
      END SUBROUTINE A17
```

Fortran

A.18 Binding of `barrier` Regions

The binding rules call for a **`barrier`** region to bind to the closest enclosing **`parallel`** region (see Section 2.8.3 on page 66).

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*.

200 OpenMP API • Version 3.0 May 2008

C/C++

Example A.18.1c

```c
void work(int n) {}

void sub3(int n)
{
  work(n);
  #pragma omp barrier
  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;
}
```

C/C++

C/C++

Example A.18.1c

```c
void work(int n) {}

void sub3(int n)
{
  work(n);
  #pragma omp barrier
  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;
}
```

C/C++

Appendix A Examples 201

Fortran

Example A.18.1f

```
      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


      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 A18
        CALL SUB1(2)
        CALL SUB2(2)
        CALL SUB3(2)
      END PROGRAM A18
```

Fortran

A.19 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 2.8.5 on page 69).

Fortran

Example A.18.1f

```
      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


      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 A18
        CALL SUB1(2)
        CALL SUB2(2)
        CALL SUB3(2)
      END PROGRAM A18
```

Fortran

A.19 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 2.8.5 on page 69).

202 OpenMP API • Version 3.0 May 2008

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 2.8.2 on page 65) 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.

C/C++

Example A.19.1c

```
float work1(int i)
{
  return 1.0 * i;
}

float work2(int i)
{
   return 2.0 * i;
}

void a19(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;
  a19(x, y, index, 10000);
  return 0;
}
```

C/C++

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 2.8.2 on page 65) 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.

C/C++

Example A.19.1c

```
float work1(int i)
{
  return 1.0 * i;
}

float work2(int i)
{
   return 2.0 * i;
}

void a19(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;
  a19(x, y, index, 10000);
  return 0;
}
```

C/C++

Appendix A Examples 203

Fortran

Example A.19.1f

```fortran
      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 SUBA19(X, Y, INDEX, N)
        REAL X(*), Y(*)
        INTEGER INDEX(*), N

        INTEGER I

!$OMP   PARALLEL DO SHARED(X, Y, INDEX, N)
          DO I=1,N
!$OMP       ATOMIC
              X(INDEX(I)) = X(INDEX(I)) + WORK1(I)
            Y(I) = Y(I) + WORK2(I)
          ENDDO

      END SUBROUTINE SUBA19

      PROGRAM A19
        REAL X(1000), Y(10000)
        INTEGER INDEX(10000)
        INTEGER I

        DO I=1,10000
          INDEX(I) = MOD(I, 1000) + 1
          Y(I) = 0.0
        ENDDO

        DO I = 1,1000
          X(I) = 0.0
        ENDDO

        CALL SUBA19(X, Y, INDEX, 10000)

      END PROGRAM A19
```

Fortran

Fortran

Example A.19.1f

```fortran
      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 SUBA19(X, Y, INDEX, N)
        REAL X(*), Y(*)
        INTEGER INDEX(*), N

        INTEGER I

!$OMP   PARALLEL DO SHARED(X, Y, INDEX, N)
          DO I=1,N
!$OMP       ATOMIC
              X(INDEX(I)) = X(INDEX(I)) + WORK1(I)
            Y(I) = Y(I) + WORK2(I)
          ENDDO

      END SUBROUTINE SUBA19

      PROGRAM A19
        REAL X(1000), Y(10000)
        INTEGER INDEX(10000)
        INTEGER I

        DO I=1,10000
          INDEX(I) = MOD(I, 1000) + 1
          Y(I) = 0.0
        ENDDO

        DO I = 1,1000
          X(I) = 0.0
        ENDDO

        CALL SUBA19(X, Y, INDEX, 10000)

      END PROGRAM A19
```

Fortran

204 OpenMP API • Version 3.0 May 2008

A.20 Restrictions on the `atomic` Construct

The following examples illustrate the restrictions on the atomic construct. For more information, see Section 2.8.5 on page 69.

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.

C/C++

Fortran

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.

Fortran

The following are some non-conforming examples:

C/C++

Example A.20.1c

```
void a20_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 */
  }
}
```

C/C++

A.20 Restrictions on the `atomic` Construct

The following examples illustrate the restrictions on the atomic construct. For more information, see Section 2.8.5 on page 69.

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.

C/C++

Fortran

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.

Fortran

The following are some non-conforming examples:

C/C++

Example A.20.1c

```
void a20_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 */
  }
}
```

C/C++

Appendix A Examples 205

Fortran

Example A.20.1f

```fortran
      SUBROUTINE A20_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 A20_1_WRONG
```

Fortran

C/C++

Example A.20.2c

```c
void a20_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 */

  }
}
```

C/C++

Fortran

Example A.20.1f

```fortran
      SUBROUTINE A20_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 A20_1_WRONG
```

Fortran

C/C++

Example A.20.2c

```c
void a20_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 */

  }
}
```

C/C++

36 206 OpenMP API • Version 3.0 May 2008

Fortran

The following example is non-conforming because *I* and *R* reference the same location but have different types.

Example A.20.2f

```
      SUBROUTINE SUB()
        COMMON /BLK/ R
        REAL R

!$OMP   ATOMIC
          R = R + 1.0
      END SUBROUTINE SUB

      SUBROUTINE A20_2_WRONG()
        COMMON /BLK/ I
        INTEGER I

!$OMP   PARALLEL

!$OMP     ATOMIC
            I = I + 1
          CALL SUB()
!$OMP   END PARALLEL
      END SUBROUTINE A20_2_WRONG
```


Fortran

The following example is non-conforming because *I* and *R* reference the same location but have different types.

Example A.20.2f

```
      SUBROUTINE SUB()
        COMMON /BLK/ R
        REAL R

!$OMP   ATOMIC
          R = R + 1.0
      END SUBROUTINE SUB

      SUBROUTINE A20_2_WRONG()
        COMMON /BLK/ I
        INTEGER I

!$OMP   PARALLEL

!$OMP     ATOMIC
            I = I + 1
          CALL SUB()
!$OMP   END PARALLEL
      END SUBROUTINE A20_2_WRONG
```

Appendix A Examples 207

Although the following example might work on some implementations, this is also non-conforming:

Example A.20.3f

```
      SUBROUTINE A20_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 A20_3_WRONG
```

Fortran

A.21 The `flush` Construct with a List

The following example uses the **`flush`** construct (see Section 2.8.6 on page 72) for point-to-point synchronization of specific variables between pairs of threads:

C/C++

Example A.21.1c

```
#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;
```


Although the following example might work on some implementations, this is also non-conforming:

Example A.20.3f

```
      SUBROUTINE A20_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 A20_3_WRONG
```

Fortran

A.21 The flush Construct with a List

The following example uses the **`flush`** construct (see Section 2.8.6 on page 72) for point-to-point synchronization of specific variables between pairs of threads:

C/C++

Example A.21.1c

```
#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;
```

208 OpenMP API • Version 3.0 May 2008

```c
}

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)
     }

     #pragma omp flush(work,synch)

     /* Read neighbor's values of work array */
     result[iam] = fn2(work[neighbor], work[iam]);
  }

  /* output result here */

  return 0;
}
```

C/C++


```c
}

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)
     }

     #pragma omp flush(work,synch)

     /* Read neighbor's values of work array */
     result[iam] = fn2(work[neighbor], work[iam]);
  }

  /* output result here */

  return 0;
}
```

C/C++

Appendix A Examples 209

Fortran

Example A.21.1f

```
      REAL FUNCTION FN1(I)
        INTEGER I
        FN1 = I * 2.0
        RETURN
      END FUNCTION FN1
      REAL FUNCTION FN2(A, B)
        REAL A, B
        FN2 = A + B
        RETURN
      END FUNCTION FN2

      PROGRAM A21
        INCLUDE "omp_lib.h"     ! or USE OMP_LIB
        INTEGER ISYNC(256)
        REAL    WORK(256)
        REAL    RESULT(256)
        INTEGER IAM, NEIGHBOR

!$OMP   PARALLEL PRIVATE(IAM, NEIGHBOR) SHARED(WORK, ISYNC)
          IAM = OMP_GET_THREAD_NUM() + 1
          ISYNC(IAM) = 0
!$OMP BARRIER
C Do computation into my portion of work array
          WORK(IAM) = FN1(IAM)
C Announce that I am done with my work.
C The first flush ensures that my work is made visible before
C synch. The second flush ensures that synch is made visible.
!$OMP     FLUSH(WORK,ISYNC)
          ISYNC(IAM) = 1
!$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
      END PROGRAM A21
```

Fortran

Fortran

Example A.21.1f

```
      REAL FUNCTION FN1(I)
        INTEGER I
        FN1 = I * 2.0
        RETURN
      END FUNCTION FN1
      REAL FUNCTION FN2(A, B)
        REAL A, B
        FN2 = A + B
        RETURN
      END FUNCTION FN2

      PROGRAM A21
        INCLUDE "omp_lib.h"     ! or USE OMP_LIB
        INTEGER ISYNC(256)
        REAL    WORK(256)
        REAL    RESULT(256)
        INTEGER IAM, NEIGHBOR

!$OMP   PARALLEL PRIVATE(IAM, NEIGHBOR) SHARED(WORK, ISYNC)
          IAM = OMP_GET_THREAD_NUM() + 1
          ISYNC(IAM) = 0
!$OMP BARRIER
C Do computation into my portion of work array
          WORK(IAM) = FN1(IAM)
C Announce that I am done with my work.
C The first flush ensures that my work is made visible before
C synch. The second flush ensures that synch is made visible.
!$OMP     FLUSH(WORK,ISYNC)
          ISYNC(IAM) = 1
!$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
      END PROGRAM A21
```

Fortran

210 OpenMP API • Version 3.0 May 2008

A.22 The `flush` Construct without a List

The following example (for Section 2.8.6 on page 72) distinguishes the shared variables affected by a **`flush`** construct with no list from the shared objects that are not affected:

C/C++

Example A.22.1c

```
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 */
```


A.22 The `flush` Construct without a List

The following example (for Section 2.8.6 on page 72) distinguishes the shared variables affected by a **`flush`** construct with no list from the shared objects that are not affected:

C/C++

Example A.22.1c

```
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 */
```

Appendix A Examples 211

```
    sum += i + j + *p + n;
  }
  return sum;
}

int main()
{
  int result = g(7);
  return result;
}
```

C/C++

Fortran

Example A.22.1f

```
      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
```



```
    sum += i + j + *p + n;
  }
  return sum;
}

int main()
{
  int result = g(7);
  return result;
}
```

C/C++

Fortran

Example A.22.1f

```
      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
```

212 OpenMP API • Version 3.0 May 2008

```
!$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
!$OMP   END PARALLEL

        G = SUM
      END FUNCTION G

      PROGRAM A22
        COMMON /DATA/ X, P
        INTEGER, TARGET  :: X
        INTEGER, POINTER :: P
        INTEGER RESULT, G

        P => X
        RESULT = G(7)
        PRINT *, RESULT
      END PROGRAM A22
```

Fortran


```
!$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
!$OMP   END PARALLEL

        G = SUM
      END FUNCTION G

      PROGRAM A22
        COMMON /DATA/ X, P
        INTEGER, TARGET  :: X
        INTEGER, POINTER :: P
        INTEGER RESULT, G

        P => X
        RESULT = G(7)
        PRINT *, RESULT
      END PROGRAM A22
```

Fortran

Appendix A Examples 213

A.23 Placement of `flush`, `barrier`, and `taskwait` Directives

The following example is non-conforming, because the **`flush`**, **`barrier`**, and **`taskwait`** directives cannot be the immediate substatement of an **`if`** statement. See Section 2.8.3 on page 66, Section 2.8.6 on page 72, and Section 2.8.4 on page 68.

Example A.23.1c

```
void a23_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 */

if (a != 0)
  #pragma omp taskwait
/* incorrect as taskwait cannot be immediate substatement
   of if statement */

  }
}
```

The following version of the above example is conforming because the **`flush`**, **`barrier`**, and **`taskwait`** directives are enclosed in a compound statement.

C/C++

A.23 Placement of flush, barrier, and taskwait Directives

The following example is non-conforming, because the flush, barrier, and taskwait directives cannot be the immediate substatement of an if statement. See Section 2.8.3 on page 66, Section 2.8.6 on page 72, and Section 2.8.4 on page 68.

Example A.23.1c

void a23_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 */

if (a != 0)
  #pragma omp taskwait
/* incorrect as taskwait cannot be immediate substatement
   of if statement */

  }
}

The following version of the above example is conforming because the flush, barrier, and taskwait directives are enclosed in a compound statement.

214 OpenMP API • Version 3.0 May 2008

Example A.23.2c

```
void a23()
{
  int a = 1;

  #pragma omp parallel
  {
     if (a != 0) {
  #pragma omp flush(a)
     }
     if (a != 0) {
  #pragma omp barrier
     }
     if (a != 0) {
  #pragma omp taskwait
     }
  }
}
```

C/C++

A.24 The `ordered` Clause and the `ordered` Construct

Ordered constructs (Section 2.8.7 on page 75) are useful for sequentially ordering the output from work that is done in parallel. The following program prints out the indices in sequential order:

Example A.23.2c

```
void a23()
{
  int a = 1;

  #pragma omp parallel
  {
     if (a != 0) {
  #pragma omp flush(a)
     }
     if (a != 0) {
  #pragma omp barrier
     }
     if (a != 0) {
  #pragma omp taskwait
     }
  }
}
```

C/C++

A.24 The `ordered` Clause and the `ordered` Construct

Ordered constructs (Section 2.8.7 on page 75) are useful for sequentially ordering the output from work that is done in parallel. The following program prints out the indices in sequential order:

Appendix A Examples 215

C/C++

Example A.24.1c

```
#include <stdio.h>

void work(int k)
{
  #pragma omp ordered
    printf(" %d\n", k);
}

void a24(int lb, int ub, int stride)
{
  int i;

  #pragma omp parallel for ordered schedule(dynamic)
  for (i=lb; i<ub; i+=stride)
    work(i);
}

int main()
{
  a24(0, 100, 5);
  return 0;
}
```

C/C++

C/C++

Example A.24.1c

```
#include <stdio.h>

void work(int k)
{
  #pragma omp ordered
    printf(" %d\n", k);
}

void a24(int lb, int ub, int stride)
{
  int i;

  #pragma omp parallel for ordered schedule(dynamic)
  for (i=lb; i<ub; i+=stride)
    work(i);
}

int main()
{
  a24(0, 100, 5);
  return 0;
}
```

C/C++

21 **216** OpenMP API • Version 3.0 May 2008

Fortran

Example A.24.1f

```
      SUBROUTINE WORK(K)
        INTEGER k

!$OMP ORDERED
        WRITE(*,*) K
!$OMP END ORDERED

      END SUBROUTINE WORK

      SUBROUTINE SUBA24(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 SUBA24

      PROGRAM A24
        CALL SUBA24(1,100,5)
      END PROGRAM A24
```

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:

Fortran

Example A.24.1f

```
      SUBROUTINE WORK(K)
        INTEGER k

!$OMP ORDERED
        WRITE(*,*) K
!$OMP END ORDERED

      END SUBROUTINE WORK

      SUBROUTINE SUBA24(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 SUBA24

      PROGRAM A24
        CALL SUBA24(1,100,5)
      END PROGRAM A24
```

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:

Appendix A Examples 217

C/C++

Example A.24.2c

```c
void work(int i) {}

void a24_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);
  }
}
```

C/C++

Fortran

Example A.24.2f

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

      SUBROUTINE A24_WRONG(N)
      INTEGER N

        INTEGER I
!$OMP   DO ORDERED
        DO I = 1, N
! incorrect because an iteration may not execute more than one
! ordered region
!$OMP     ORDERED
            CALL WORK(I)
!$OMP     END ORDERED

!$OMP     ORDERED
            CALL WORK(I+1)
!$OMP     END ORDERED
        END DO
      END SUBROUTINE A24_WRONG
```

Fortran

C/C++

Example A.24.2c

```c
void work(int i) {}

void a24_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);
  }
}
```

C/C++

Fortran

Example A.24.2f

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

      SUBROUTINE A24_WRONG(N)
      INTEGER N

        INTEGER I
!$OMP   DO ORDERED
        DO I = 1, N
! incorrect because an iteration may not execute more than one
! ordered region
!$OMP     ORDERED
            CALL WORK(I)
!$OMP     END ORDERED

!$OMP     ORDERED
            CALL WORK(I+1)
!$OMP     END ORDERED
        END DO
      END SUBROUTINE A24_WRONG
```

Fortran

218 OpenMP API • Version 3.0 May 2008

The following is a conforming example with more than one **ordered** construct. Each iteration will execute only one **ordered** region:

C/C++

Example A.24.3c

```c
void work(int i) {}
void a24_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);
    }
  }
}
```

C/C++

Fortran

Example A.24.3f

```fortran
      SUBROUTINE A24_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
          ENDIF
        ENDDO
      END SUBROUTINE A24_GOOD
```

Fortran

The following is a conforming example with more than one **ordered** construct. Each iteration will execute only one **ordered** region:

C/C++

Example A.24.3c

```c
void work(int i) {}
void a24_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);
    }
  }
}
```

C/C++

Fortran

Example A.24.3f

```fortran
      SUBROUTINE A24_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
          ENDIF
        ENDDO
      END SUBROUTINE A24_GOOD
```

Fortran

Appendix A Examples 219

A.25 The `threadprivate` Directive

The following examples demonstrate how to use the **`threadprivate`** directive (Section 2.9.2 on page 81) to give each thread a separate counter.

C/C++

Example A.25.1c

```c
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
  counter++;
  return(counter);
}
```

C/C++

Fortran

Example A.25.1f

```fortran
      INTEGER FUNCTION INCREMENT_COUNTER()
        COMMON/A25_COMMON/COUNTER
!$OMP   THREADPRIVATE(/A25_COMMON/)

        COUNTER = COUNTER +1
        INCREMENT_COUNTER = COUNTER
        RETURN
      END FUNCTION INCREMENT_COUNTER
```

Fortran

C/C++

The following example uses **`threadprivate`** on a static variable:

Example A.25.2c

```c
int increment_counter_2()
{
  static int counter = 0;
  #pragma omp threadprivate(counter)
  counter++;
  return(counter);
}
```


A.25 The `threadprivate` Directive

The following examples demonstrate how to use the **`threadprivate`** directive (Section 2.9.2 on page 81) to give each thread a separate counter.

C/C++

Example A.25.1c

```c
int counter = 0;
#pragma omp threadprivate(counter)

int increment_counter()
{
  counter++;
  return(counter);
}
```

C/C++

Fortran

Example A.25.1f

```fortran
      INTEGER FUNCTION INCREMENT_COUNTER()
        COMMON/A25_COMMON/COUNTER
!$OMP   THREADPRIVATE(/A25_COMMON/)

        COUNTER = COUNTER +1
        INCREMENT_COUNTER = COUNTER
        RETURN
      END FUNCTION INCREMENT_COUNTER
```

Fortran

C/C++

The following example uses **`threadprivate`** on a static variable:

Example A.25.2c

```c
int increment_counter_2()
{
  static int counter = 0;
  #pragma omp threadprivate(counter)
  counter++;
  return(counter);
}
```

220 OpenMP API • Version 3.0 May 2008

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 variable and a copy-constructor.

Example A.25.3c

```
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:
    * a is constructed from x (with value 1 or 2?)
    * 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 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 variable and a copy-constructor.

Example A.25.3c

```
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:
    * a is constructed from x (with value 1 or 2?)
    * b is copy-constructed from b_aux
    */

   for (int i=0; i<n; i++) {
       g(a, b); /* Value of a is unspecified. */
   }
}
```

C/C++

Appendix A Examples 221

Fortran

The following examples show non-conforming uses and correct uses of the **threadprivate** directive. For more information, see Section 2.9.2 on page 81 and Section 2.9.4.1 on page 101.

The following example is non-conforming because the common block is not declared local to the subroutine that refers to it:

Example A.25.2f

```
      MODULE A25_MODULE
        COMMON /T/ A
      END MODULE A25_MODULE

      SUBROUTINE A25_2_WRONG()
        USE A25_MODULE
!$OMP   THREADPRIVATE(/T/)
      !non-conforming because /T/ not declared in A25_4_WRONG
      END SUBROUTINE A25_2_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.25.3f

```
      SUBROUTINE A25_3_WRONG()
        COMMON /T/ A
!$OMP   THREADPRIVATE(/T/)

        CONTAINS
          SUBROUTINE A25_3S_WRONG()
!$OMP       PARALLEL COPYIN(/T/)
      !non-conforming because /T/ not declared in A35_5S_WRONG
!$OMP       END PARALLEL
          END SUBROUTINE A25_3S_WRONG
      END SUBROUTINE A25_3_WRONG
```


Fortran

The following examples show non-conforming uses and correct uses of the **threadprivate** directive. For more information, see Section 2.9.2 on page 81 and Section 2.9.4.1 on page 101.

The following example is non-conforming because the common block is not declared local to the subroutine that refers to it:

Example A.25.2f

```
      MODULE A25_MODULE
        COMMON /T/ A
      END MODULE A25_MODULE

      SUBROUTINE A25_2_WRONG()
        USE A25_MODULE
!$OMP   THREADPRIVATE(/T/)
      !non-conforming because /T/ not declared in A25_4_WRONG
      END SUBROUTINE A25_2_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.25.3f

```
      SUBROUTINE A25_3_WRONG()
        COMMON /T/ A
!$OMP   THREADPRIVATE(/T/)

        CONTAINS
          SUBROUTINE A25_3S_WRONG()
!$OMP       PARALLEL COPYIN(/T/)
      !non-conforming because /T/ not declared in A35_5S_WRONG
!$OMP       END PARALLEL
          END SUBROUTINE A25_3S_WRONG
      END SUBROUTINE A25_3_WRONG
```

222 OpenMP API • Version 3.0 May 2008

Fortran (cont.)

The following example is a correct rewrite of the previous example:

Example A.25.4f

```
      SUBROUTINE A25_4_GOOD()
       COMMON /T/ A
!$OMP   THREADPRIVATE(/T/)

       CONTAINS
         SUBROUTINE A25_4S_GOOD()
           COMMON /T/ A
!$OMP       THREADPRIVATE(/T/)

!$OMP       PARALLEL COPYIN(/T/)
!$OMP       END PARALLEL
        END SUBROUTINE A25_4S_GOOD
      END SUBROUTINE A25_4_GOOD
```


Fortran (cont.)

The following example is a correct rewrite of the previous example:

Example A.25.4f

```
      SUBROUTINE A25_4_GOOD()
       COMMON /T/ A
!$OMP   THREADPRIVATE(/T/)

       CONTAINS
         SUBROUTINE A25_4S_GOOD()
           COMMON /T/ A
!$OMP       THREADPRIVATE(/T/)

!$OMP       PARALLEL COPYIN(/T/)
!$OMP       END PARALLEL
        END SUBROUTINE A25_4S_GOOD
      END SUBROUTINE A25_4_GOOD
```

Appendix A Examples **223**

The following is an example of the use of **threadprivate** for local variables:

Example A.25.5f

```
      PROGRAM A25_5_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
            PRINT *

!$OMP     END CRITICAL
!$OMP   END PARALLEL
      END PROGRAM A25_5_GOOD
```

The above program, if executed by two threads, will print one of the following two sets of output:

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

A is not allocated
```


Fortran (cont.)

The following is an example of the use of **threadprivate** for local variables:

Example A.25.5f

```
      PROGRAM A25_5_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
            PRINT *

!$OMP     END CRITICAL
!$OMP   END PARALLEL
      END PROGRAM A25_5_GOOD
```

The above program, if executed by two threads, will print one of the following two sets of output:

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

A is not allocated
```

224 OpenMP API • Version 3.0 May 2008

```
ptr = 4
i = 5
```

or

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

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

The following is an example of the use of **`threadprivate`** for module variables:

Example A.25.6f

```
      MODULE A25_MODULE6
        REAL, POINTER :: WORK(:)
        SAVE WORK
!$OMP   THREADPRIVATE(WORK)
      END MODULE A25_MODULE6

      SUBROUTINE SUB1(N)
      USE A25_MODULE6
!$OMP   PARALLEL PRIVATE(THE_SUM)
        ALLOCATE(WORK(N))
        CALL SUB2(THE_SUM)
       WRITE(*,*)THE_SUM
!$OMP   END PARALLEL
      END SUBROUTINE SUB1

      SUBROUTINE SUB2(THE_SUM)
        USE A25_MODULE6
        WORK(:) = 10
        THE_SUM=SUM(WORK)
      END SUBROUTINE SUB2

      PROGRAM A25_6_GOOD
        N = 10
        CALL SUB1(N)
      END PROGRAM A25_6_GOOD
```

Fortran

C/C++

The following example illustrates initialization of threadprivate variables for class-type *T*. *t1* is default constructed, *t2* is constructed taking a constructor accepting one argument of integer type, *t3* is copy constructed with argument *f()*:

ptr = 4
i = 5

or

A is not allocated
ptr = 4
i = 15

a = 1 2 3
ptr = 4
i = 5

The following is an example of the use of threadprivate for module variables:

Example A.25.6f

      MODULE A25_MODULE6
        REAL, POINTER :: WORK(:)
        SAVE WORK
!$OMP   THREADPRIVATE(WORK)
      END MODULE A25_MODULE6

      SUBROUTINE SUB1(N)
      USE A25_MODULE6
!$OMP   PARALLEL PRIVATE(THE_SUM)
        ALLOCATE(WORK(N))
        CALL SUB2(THE_SUM)
       WRITE(*,*)THE_SUM
!$OMP   END PARALLEL
      END SUBROUTINE SUB1

      SUBROUTINE SUB2(THE_SUM)
        USE A25_MODULE6
        WORK(:) = 10
        THE_SUM=SUM(WORK)
      END SUBROUTINE SUB2

      PROGRAM A25_6_GOOD
        N = 10
        CALL SUB1(N)
      END PROGRAM A25_6_GOOD

Fortran

C/C++

The following example illustrates initialization of threadprivate variables for class-type T. t1 is default constructed, t2 is constructed taking a constructor accepting one argument of integer type, t3 is copy constructed with argument f():

Appendix A Examples 225

Example A.25.4c

```
static T t1;
#pragma omp threadprivate(t1)
static T t2( 23 );
#pragma omp threadprivate(t2)
static T t3 = f();
#pragma omp threadprivate(t3)
```

The following example illustrates the use of threadprivate for static class members. The threadprivate directive for a static class member must be placed inside the class definition.

Example A.25.5c

```
class T {
 public:
  static int i;
#pragma omp threadprivate(i)
};
```

C/C++

C/C++

A.26 Parallel Random Access Iterator Loop

The following example shows a parallel Random access iterator loop.

Example A.26.1c

```
#include <vector>
void iterator_example()
{
  std::vector<int> vec(23);
  std::vector<int>::iterator it;
#pragma omp parallel for default(none) shared(vec)
  for (it = vec.begin(); it < vec.end(); it++)
  {
    // do work with *it //
  }
}
```

C/C++

Example A.25.4c

```
static T t1;
#pragma omp threadprivate(t1)
static T t2( 23 );
#pragma omp threadprivate(t2)
static T t3 = f();
#pragma omp threadprivate(t3)
```

The following example illustrates the use of threadprivate for static class members. The threadprivate directive for a static class member must be placed inside the class definition.

Example A.25.5c

```
class T {
 public:
  static int i;
#pragma omp threadprivate(i)
};
```

C/C++

C/C++

A.26 Parallel Random Access Iterator Loop

The following example shows a parallel Random access iterator loop.

Example A.26.1c

```
#include <vector>
void iterator_example()
{
  std::vector<int> vec(23);
  std::vector<int>::iterator it;
#pragma omp parallel for default(none) shared(vec)
  for (it = vec.begin(); it < vec.end(); it++)
  {
    // do work with *it //
  }
}
```

C/C++

226 OpenMP API • Version 3.0 May 2008

Fortran

A.27 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 2.9.3 on page 85.

The following example is conforming:

Example A.27.1f

```
      SUBROUTINE A27_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 A27_1_GOOD
```


Fortran

A.27 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 2.9.3 on page 85.

The following example is conforming:

Example A.27.1f

```
      SUBROUTINE A27_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 A27_1_GOOD
```

Appendix A Examples 227

The following example is also conforming:

Example A.27.2f

```
      SUBROUTINE A27_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 A27_2_GOOD
```

The following example is conforming:

Example A.27.3f

```
      SUBROUTINE A27_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 A27_3_GOOD
```


Fortran (cont.)

The following example is also conforming:

Example A.27.2f

      SUBROUTINE A27_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 A27_2_GOOD

The following example is conforming:

Example A.27.3f

      SUBROUTINE A27_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 A27_3_GOOD

228 OpenMP API • Version 3.0 May 2008

The following example is non-conforming because x is a constituent element of c:

Example A.27.4f

```
      SUBROUTINE A27_4_WRONG()
        COMMON /C/ X,Y
! Incorrect because X is a constituent element of C
!$OMP   PARALLEL PRIVATE(/C/), SHARED(X)
          ! do work here
!$OMP   END PARALLEL
      END SUBROUTINE A27_4_WRONG
```

The following example is non-conforming because a common block may not be declared both shared and private:

Example A.27.5f

```
      SUBROUTINE A27_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 A27_5_WRONG
```

Fortran

A.28 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 2.9.3.1 on page 86.

The following example is non-conforming because x is a constituent element of c:

Example A.27.4f

```
      SUBROUTINE A27_4_WRONG()
        COMMON /C/ X,Y
! Incorrect because X is a constituent element of C
!$OMP   PARALLEL PRIVATE(/C/), SHARED(X)
          ! do work here
!$OMP   END PARALLEL
      END SUBROUTINE A27_4_WRONG
```

The following example is non-conforming because a common block may not be declared both shared and private:

Example A.27.5f

```
      SUBROUTINE A27_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 A27_5_WRONG
```

Fortran

A.28 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 2.9.3.1 on page 86.

Appendix A Examples 229

C/C++

Example A.28.1c

```c
#include <omp.h>
int x, y, z[1000];
#pragma omp threadprivate(x)

void a28(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 */
  }
}
```

C/C++

C/C++

Example A.28.1c

```c
#include <omp.h>
int x, y, z[1000];
#pragma omp threadprivate(x)

void a28(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 */
  }
}
```

C/C++

230 OpenMP API • Version 3.0 May 2008

Fortran

Example A.28.1f

```
      SUBROUTINE A28(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

          Z(I) = Y    ! Error - cannot reference I or Y here
!$OMP   END PARALLEL
      END SUBROUTINE A28
```

Fortran

Fortran

A.29 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 2.9.3.2 on page 88). The subroutine call passing an array section argument may cause the compiler to copy the argument into a

Fortran

Example A.28.1f

```
      SUBROUTINE A28(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

          Z(I) = Y    ! Error - cannot reference I or Y here
!$OMP   END PARALLEL
      END SUBROUTINE A28
```

Fortran

Fortran

A.29 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 2.9.3.2 on page 88). The subroutine call passing an array section argument may cause the compiler to copy the argument into a

Appendix A Examples 231

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.29.1f

```
SUBROUTINE A29

  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 A29

SUBROUTINE SUB(X)
  REAL X(*)
  X(1:5) = 4
END SUBROUTINE SUB
```

Fortran

A.30 The `private` Clause

In the following example, the values of original list items *i* and *j* are retained on exit from the **parallel** region, while the private list items *i* and *j* are modified within the **parallel** construct. For more information on the **private** clause, see Section 2.9.3.3 on page 89.

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.29.1f

```
SUBROUTINE A29

  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 A29

SUBROUTINE SUB(X)
  REAL X(*)
  X(1:5) = 4
END SUBROUTINE SUB
```

Fortran

A.30 The private Clause

In the following example, the values of original list items *i* and *j* are retained on exit from the **parallel** region, while the private list items *i* and *j* are modified within the **parallel** construct. For more information on the **private** clause, see Section 2.9.3.3 on page 89.

232 OpenMP API • Version 3.0 May 2008

C/C++

Example A.30.1c

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

int main()
{
  int i, j;
  int *ptr_i, *ptr_j;

  i = 1;
  j = 2;

  ptr_i = &i;
  ptr_j = &j;

  #pragma omp parallel private(i) firstprivate(j)
  {
    i = 3;
    j = j + 2;
    assert (*ptr_i == 1 && *ptr_j == 2);
  }

  assert(i == 1 && j == 2);

  return 0;
}
```

C/C++

Fortran

Example A.30.1f

```fortran
      PROGRAM A30
        INTEGER I, J

        I = 1
        J = 2

!$OMP   PARALLEL PRIVATE(I) FIRSTPRIVATE(J)
          I = 3
          J = J + 2
!$OMP   END PARALLEL

        PRINT *, I, J  ! I .eq. 1 .and. J .eq. 2
      END PROGRAM A30
```

Fortran

C/C++

Example A.30.1c

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

int main()
{
  int i, j;
  int *ptr_i, *ptr_j;

  i = 1;
  j = 2;

  ptr_i = &i;
  ptr_j = &j;

  #pragma omp parallel private(i) firstprivate(j)
  {
    i = 3;
    j = j + 2;
    assert (*ptr_i == 1 && *ptr_j == 2);
  }

  assert(i == 1 && j == 2);

  return 0;
}
```

C/C++

Fortran

Example A.30.1f

```fortran
      PROGRAM A30
        INTEGER I, J

        I = 1
        J = 2

!$OMP   PARALLEL PRIVATE(I) FIRSTPRIVATE(J)
          I = 3
          J = J + 2
!$OMP   END PARALLEL

        PRINT *, I, J  ! I .eq. 1 .and. J .eq. 2
      END PROGRAM A30
```

Fortran

Appendix A Examples 233

In the following example, all uses of the variable *a* within the loop construct in the routine *f* refer to a private list item *a*, while it is unspecified whether references to *a* in the routine *g* are to a private list item or the original list item.

C/C++

Example A.30.2c

```
int a;

void g(int k) {
  a = k; /* Accessed in the region but outside of the construct;
          * therefore unspecified whether original or private list
          * item is modified. */
}


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" */
    }
}
```

C/C++

In the following example, all uses of the variable *a* within the loop construct in the routine *f* refer to a private list item *a*, while it is unspecified whether references to *a* in the routine *g* are to a private list item or the original list item.

C/C++

Example A.30.2c

```
int a;

void g(int k) {
  a = k; /* Accessed in the region but outside of the construct;
          * therefore unspecified whether original or private list
          * item is modified. */
}


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" */
    }
}
```

C/C++

21 234 OpenMP API • Version 3.0 May 2008

Fortran

Example A.30.2f

```
      MODULE A30_2
         REAL A

         CONTAINS

           SUBROUTINE G(K)
             REAL K
             A = K  ! Accessed in the region but outside of the
                    ! construct; therefore unspecified whether
                    ! original or private list item is modified.
           END SUBROUTINE G

           SUBROUTINE F(N)
           INTEGER N
           REAL A

             INTEGER I
!$OMP       PARALLEL DO PRIVATE(A)
               DO I = 1,N
                 A = I
                 CALL G(A*2)
               ENDDO
!$OMP       END PARALLEL DO
           END SUBROUTINE F

       END MODULE A30_2
```

Fortran

A.31 Reprivatization

The following example demonstrates the reprivatization of variables (see Section 2.9.3.3 on page 89). Private variables can be marked **private** again in a nested construct. They do not have to be shared in the enclosing **parallel** region.

Fortran

Example A.30.2f

```
      MODULE A30_2
         REAL A

         CONTAINS

           SUBROUTINE G(K)
             REAL K
             A = K  ! Accessed in the region but outside of the
                    ! construct; therefore unspecified whether
                    ! original or private list item is modified.
           END SUBROUTINE G

           SUBROUTINE F(N)
           INTEGER N
           REAL A

             INTEGER I
!$OMP       PARALLEL DO PRIVATE(A)
               DO I = 1,N
                 A = I
                 CALL G(A*2)
               ENDDO
!$OMP       END PARALLEL DO
           END SUBROUTINE F

       END MODULE A30_2
```

Fortran

A.31 Reprivatization

The following example demonstrates the reprivatization of variables (see Section 2.9.3.3 on page 89). Private variables can be marked **private** again in a nested construct. They do not have to be shared in the enclosing **parallel** region.

Appendix A Examples 235

C/C++

Example A.31.1c

```c
#include <assert.h>
void a31()
{
  int i, a;

  #pragma omp parallel private(a)
  {
     a = 1;
    #pragma omp parallel for private(a)
      for (i=0; i<10; i++)
     {
       a = 2;
     }
    assert(a == 1);
  }
}
```

C/C++

Fortran

Example A.31.1f

```fortran
      SUBROUTINE A31()
        INTEGER I, A

!$OMP   PARALLEL PRIVATE(A)
         A = 1
!$OMP     PARALLEL DO PRIVATE(A)
          DO I = 1, 10
            A = 2
          END DO
!$OMP     END PARALLEL DO
        PRINT *, A ! Outer A still has value 1
!$OMP   END PARALLEL
      END SUBROUTINE A31
```

Fortran

C/C++

Example A.31.1c

```c
#include <assert.h>
void a31()
{
  int i, a;

  #pragma omp parallel private(a)
  {
     a = 1;
    #pragma omp parallel for private(a)
      for (i=0; i<10; i++)
     {
       a = 2;
     }
    assert(a == 1);
  }
}
```

C/C++

Fortran

Example A.31.1f

```fortran
      SUBROUTINE A31()
        INTEGER I, A

!$OMP   PARALLEL PRIVATE(A)
         A = 1
!$OMP     PARALLEL DO PRIVATE(A)
          DO I = 1, 10
            A = 2
          END DO
!$OMP     END PARALLEL DO
        PRINT *, A ! Outer A still has value 1
!$OMP   END PARALLEL
      END SUBROUTINE A31
```

Fortran

236 OpenMP API • Version 3.0 May 2008

A.32 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 2.9.3.3 on page 89).

Example A.32.1f

```
       SUBROUTINE SUB()
       COMMON /BLOCK/ X
       PRINT *,X             ! X is undefined
       END SUBROUTINE SUB

       PROGRAM A32_1
         COMMON /BLOCK/ X
         X = 1.0
!$OMP    PARALLEL PRIVATE (X)
         X = 2.0
         CALL SUB()
!$OMP    END PARALLEL
      END PROGRAM A32_1
```

Example A.32.2f

```
      PROGRAM A32_2
        COMMON /BLOCK2/ X
        X = 1.0

!$OMP   PARALLEL PRIVATE (X)
          X = 2.0
          CALL SUB()
!$OMP   END PARALLEL

       CONTAINS

          SUBROUTINE SUB()
          COMMON /BLOCK2/ Y

          PRINT *,X              ! X is undefined
          PRINT *,Y              ! Y is undefined
          END SUBROUTINE SUB

       END PROGRAM A32_2
```


Fortran

A.32 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 2.9.3.3 on page 89).

Example A.32.1f

```
       SUBROUTINE SUB()
       COMMON /BLOCK/ X
       PRINT *,X             ! X is undefined
       END SUBROUTINE SUB

       PROGRAM A32_1
         COMMON /BLOCK/ X
         X = 1.0
!$OMP    PARALLEL PRIVATE (X)
         X = 2.0
         CALL SUB()
!$OMP    END PARALLEL
      END PROGRAM A32_1
```

Example A.32.2f

```
      PROGRAM A32_2
        COMMON /BLOCK2/ X
        X = 1.0

!$OMP   PARALLEL PRIVATE (X)
          X = 2.0
          CALL SUB()
!$OMP   END PARALLEL

       CONTAINS

          SUBROUTINE SUB()
          COMMON /BLOCK2/ Y

          PRINT *,X              ! X is undefined
          PRINT *,Y              ! Y is undefined
          END SUBROUTINE SUB

       END PROGRAM A32_2
```

Appendix A Examples 237

Example A.32.3f

```
      PROGRAM A32_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 A32_3
```

Example A.32.4f

```
    PROGRAM A32_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
        ENDDO
!$OMP END PARALLEL DO       ! The LASTPRIVATE write for A has
                            ! undefined results

         PRINT *, B    ! B is undefined since the LASTPRIVATE
                       ! write of A was not defined
     END PROGRAM A32_4
```


Fortran (cont.)

Example A.32.3f

      PROGRAM A32_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 A32_3

Example A.32.4f

    PROGRAM A32_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
        ENDDO
!$OMP END PARALLEL DO   ! The LASTPRIVATE write for A has
                            ! undefined results

         PRINT *, B   ! B is undefined since the LASTPRIVATE
                       ! write of A was not defined
     END PROGRAM A32_4

238 OpenMP API • Version 3.0 May 2008

Example A.32.5f

```
      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 A32_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 A32_5
```

Fortran

Example A.32.5f

```
      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 A32_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 A32_5
```

Fortran

Appendix A Examples 239

C/C++

A.33 C/C++ Arrays in a `firstprivate` Clause

The following example illustrates the size and value of list items of array or pointer type in a **`firstprivate`** clause (Section 2.9.3.4 on page 92). 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 **`A`** is array of two arrays of two ints.
- The type of **`B`** is adjusted to pointer to array of **`n`** ints, because it is a function parameter.
- The type of **`C`** is adjusted to pointer to int, because it is a function parameter.
- The type of **`D`** is array of two arrays of two ints.
- The type of **`E`** is array of **`n`** arrays of **`n`** ints.

Note that **`B`** and **`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.

C/C++

A.33 C/C++ Arrays in a firstprivate Clause

The following example illustrates the size and value of list items of array or pointer type in a firstprivate clause (Section 2.9.3.4 on page 92). 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 A is array of two arrays of two ints.
- The type of B is adjusted to pointer to array of n ints, because it is a function parameter.
- The type of C is adjusted to pointer to int, because it is a function parameter.
- The type of D is array of two arrays of two ints.
- The type of E is array of n arrays of n ints.

Note that B and 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.

240 OpenMP API • Version 3.0 May 2008

Example A.33.1c

```c
#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));

    /* Private B and C have values of original B and C. */
    assert(&B[1][1] == &A[1][1]);
    assert(&C[3] == &A[1][1]);
    assert(D[1][1] == 4);
    assert(E[1][1] == 4);
  }
}

int main() {
  f(2, A, A[0]);
  return 0;
}
```

C/C++

A.34 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 2.9.3.5 on page 94) so that the values of the variables are the same as when the loop is executed sequentially.

Example A.33.1c

```c
#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));

    /* Private B and C have values of original B and C. */
    assert(&B[1][1] == &A[1][1]);
    assert(&C[3] == &A[1][1]);
    assert(D[1][1] == 4);
    assert(E[1][1] == 4);
  }
}

int main() {
  f(2, A, A[0]);
  return 0;
}
```

C/C++

A.34 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 2.9.3.5 on page 94) so that the values of the variables are the same as when the loop is executed sequentially.

Appendix A Examples **241**

C/C++

Example A.34.1c

```c
void a34 (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 */
}
```

C/C++

Fortran

Example A.34.1f

```fortran
      SUBROUTINE A34(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 A34
```

Fortran

A.35 The `reduction` Clause

The following example demonstrates the **`reduction`** clause (Section 2.9.3.6 on page 96):

C/C++

Example A.34.1c

```c
void a34 (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 */
}
```

C/C++

Fortran

Example A.34.1f

```fortran
      SUBROUTINE A34(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 A34
```

Fortran

A.35 The `reduction` Clause

The following example demonstrates the **`reduction`** clause (Section 2.9.3.6 on page 96):

242 OpenMP API • Version 3.0 May 2008

C/C++

Example A.35.1c

```c
void a35_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];

  }

}
```

C/C++

Fortran

Example A.35.1f

```fortran
      SUBROUTINE A35_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))


!  Note that some reductions can be expressed in
!  other forms. For example, the MIN could be expressed as
!  IF (B > Y(I)) B = Y(I)

         END DO

      END SUBROUTINE A35_1
```

Fortran

C/C++

Example A.35.1c

```c
void a35_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];

  }

}
```

C/C++

Fortran

Example A.35.1f

```fortran
      SUBROUTINE A35_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))


!  Note that some reductions can be expressed in
!  other forms. For example, the MIN could be expressed as
!  IF (B > Y(I)) B = Y(I)

         END DO

      END SUBROUTINE A35_1
```

Fortran

Appendix A Examples 243

A common implementation of the preceding example is to treat it as if it had been written as follows:

C/C++

Example A.35.2c

```c
void a35_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;
    }

  }
}
```

C/C++

A common implementation of the preceding example is to treat it as if it had been written as follows:

C/C++

Example A.35.2c

```c
void a35_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;
    }

  }
}
```

C/C++

244 OpenMP API • Version 3.0 May 2008

Fortran

Example A.35.2f

```
      SUBROUTINE A35_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 A35_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.35.3f

```
 PROGRAM A35_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 A35_3_WRONG

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


Fortran

Example A.35.2f

```
      SUBROUTINE A35_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 A35_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.35.3f

```
 PROGRAM A35_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 A35_3_WRONG

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

Appendix A Examples 245

The following conforming program performs the reduction using the *intrinsic procedure name* **MAX** even though the intrinsic **MAX** has been renamed to **REN**.

Example A.35.4f

```
MODULE M
   INTRINSIC MAX
END MODULE M

PROGRAM A35_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 A35_4
```

The following conforming program performs the reduction using *intrinsic procedure name* **MAX** even though the intrinsic **MAX** has been renamed to **MIN**.

Example A.35.5f

```
MODULE MOD
   INTRINSIC MAX, MIN
END MODULE MOD

PROGRAM A35_5
   USE MOD, MIN=>MAX, MAX=>MIN
   REAL :: R
   R = -HUGE(0.0)

!$OMP PARALLEL DO REDUCTION(MIN: R)     ! still does MAX
   DO I = 1, 1000
      R = MIN(R, SIN(REAL(I)))
   END DO
   PRINT *, R
END PROGRAM A35_5
```

Fortran

The following example is non-conforming because the initialization (a = 0) of the original list item "a" is not synchronized with the update of "a" as a result of the reduction computation in the for loop. Therefore, the example may print an incorrect value for "a".

The following conforming program performs the reduction using the *intrinsic procedure name* **MAX** even though the intrinsic **MAX** has been renamed to **REN**.

Example A.35.4f

```
MODULE M
   INTRINSIC MAX
END MODULE M

PROGRAM A35_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 A35_4
```

The following conforming program performs the reduction using *intrinsic procedure name* **MAX** even though the intrinsic **MAX** has been renamed to **MIN**.

Example A.35.5f

```
MODULE MOD
   INTRINSIC MAX, MIN
END MODULE MOD

PROGRAM A35_5
   USE MOD, MIN=>MAX, MAX=>MIN
   REAL :: R
   R = -HUGE(0.0)

!$OMP PARALLEL DO REDUCTION(MIN: R)     ! still does MAX
   DO I = 1, 1000
      R = MIN(R, SIN(REAL(I)))
   END DO
   PRINT *, R
END PROGRAM A35_5
```

Fortran

The following example is non-conforming because the initialization (a = 0) of the original list item "a" is not synchronized with the update of "a" as a result of the reduction computation in the for loop. Therefore, the example may print an incorrect value for "a".

246 OpenMP API • Version 3.0 May 2008

To avoid this problem, the initialization of the original list item "a" should complete before any update of "a" as a result of the reduction clause. This can be achieved by adding an explicit barrier after the assignment a = 0, or by enclosing the assignment a = 0 in a single directive (which has an implied barrier), or by initializing "a" before the start of the parallel region.

C/C++

Example A.35.3c

```
#include <stdio.h>

int main (void)
{
  int a, i;

  #pragma omp parallel shared(a) private(i)
  {
    #pragma omp master
    a = 0;

    // To avoid race conditions, add a barrier here.

    #pragma omp for reduction(+:a)
    for (i = 0; i < 10; i++) {
        a += i;
    }

    #pragma omp single
    printf ("Sum is %d\n", a);
  }
}
```

C/C++

To avoid this problem, the initialization of the original list item "a" should complete before any update of "a" as a result of the reduction clause. This can be achieved by adding an explicit barrier after the assignment a = 0, or by enclosing the assignment a = 0 in a single directive (which has an implied barrier), or by initializing "a" before the start of the parallel region.

C/C++

Example A.35.3c

```
#include <stdio.h>

int main (void)
{
  int a, i;

  #pragma omp parallel shared(a) private(i)
  {
    #pragma omp master
    a = 0;

    // To avoid race conditions, add a barrier here.

    #pragma omp for reduction(+:a)
    for (i = 0; i < 10; i++) {
        a += i;
    }

    #pragma omp single
    printf ("Sum is %d\n", a);
  }
}
```

C/C++

Appendix A Examples 247

Fortran

Example A.35.6f

```
      INTEGER A, I

!$OMP PARALLEL SHARED(A) PRIVATE(I)

!$OMP MASTER
      A = 0
!$OMP END MASTER

      ! To avoid race conditions, add a barrier here.

!$OMP DO REDUCTION(+:A)
      DO I= 0, 9
         A = A + I
      END DO

!$OMP SINGLE
      PRINT *, "Sum is ", A
!$OMP END SINGLE

!$OMP END PARALLEL
      END
```

Fortran

A.36 The `copyin` Clause

The **`copyin`** clause (see Section 2.9.4.1 on page 101) 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.

Fortran

Example A.35.6f

```
      INTEGER A, I

!$OMP PARALLEL SHARED(A) PRIVATE(I)

!$OMP MASTER
      A = 0
!$OMP END MASTER

      ! To avoid race conditions, add a barrier here.

!$OMP DO REDUCTION(+:A)
      DO I= 0, 9
         A = A + I
      END DO

!$OMP SINGLE
      PRINT *, "Sum is ", A
!$OMP END SINGLE

!$OMP END PARALLEL
      END
```

Fortran

A.36 The `copyin` Clause

The **`copyin`** clause (see Section 2.9.4.1 on page 101) 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.

248 OpenMP API • Version 3.0 May 2008

C/C++

Example A.36.1c

```
#include <stdlib.h>

float* work;
int size;
float tol;

#pragma omp threadprivate(work,size,tol)

void build()
{
  int i;
  work = (float*)malloc( sizeof(float)*size );
  for( i = 0; i < size; ++i ) work[i] = tol;
}

void a36( float t, int n )
{
  tol = t;
  size = n;
  #pragma omp parallel copyin(tol,size)
  {
    build();
  }
}
```

C/C++

C/C++

Example A.36.1c

```
#include <stdlib.h>

float* work;
int size;
float tol;

#pragma omp threadprivate(work,size,tol)

void build()
{
  int i;
  work = (float*)malloc( sizeof(float)*size );
  for( i = 0; i < size; ++i ) work[i] = tol;
}

void a36( float t, int n )
{
  tol = t;
  size = n;
  #pragma omp parallel copyin(tol,size)
  {
    build();
  }
}
```

C/C++

Appendix A Examples 249

Fortran

Example A.36.1f

```
      MODULE M
        REAL, POINTER, SAVE :: WORK(:)
        INTEGER :: SIZE
        REAL :: TOL
!$OMP   THREADPRIVATE(WORK,SIZE,TOL)
      END MODULE M

      SUBROUTINE A36( T, N )
        USE M
        REAL :: T
        INTEGER :: N
        TOL = T
        SIZE = N
!$OMP   PARALLEL COPYIN(TOL,SIZE)
        CALL BUILD
!$OMP   END PARALLEL
      END SUBROUTINE A36

      SUBROUTINE BUILD
        USE M
        ALLOCATE(WORK(SIZE))
        WORK = TOL
      END SUBROUTINE BUILD
```

Fortran

A.37 The `copyprivate` Clause

The **`copyprivate`** clause (see Section 2.9.4.2 on page 102) 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 variables designated by *a*, *b*, *x*, and *y* in all threads have become defined with the values read.

Fortran

Example A.36.1f

```
      MODULE M
        REAL, POINTER, SAVE :: WORK(:)
        INTEGER :: SIZE
        REAL :: TOL
!$OMP   THREADPRIVATE(WORK,SIZE,TOL)
      END MODULE M

      SUBROUTINE A36( T, N )
        USE M
        REAL :: T
        INTEGER :: N
        TOL = T
        SIZE = N
!$OMP   PARALLEL COPYIN(TOL,SIZE)
        CALL BUILD
!$OMP   END PARALLEL
      END SUBROUTINE A36

      SUBROUTINE BUILD
        USE M
        ALLOCATE(WORK(SIZE))
        WORK = TOL
      END SUBROUTINE BUILD
```

Fortran

A.37 The `copyprivate` Clause

The **`copyprivate`** clause (see Section 2.9.4.2 on page 102) 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 variables designated by *a*, *b*, *x*, and *y* in all threads have become defined with the values read.

250 OpenMP API • Version 3.0 May 2008

C/C++

Example A.37.1c

```c
#include <stdio.h>
float x, y;
#pragma omp threadprivate(x, y)

void init(float a, float b ) {
    #pragma omp single copyprivate(a,b,x,y)
    {
        scanf("%f %f %f %f", &a, &b, &x, &y);
    }
}
```

C/C++

Fortran

Example A.37.1f

```fortran
      SUBROUTINE INIT(A,B)
      REAL A, B
        COMMON /XY/ X,Y
!$OMP   THREADPRIVATE (/XY/)

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

      END SUBROUTINE INIT
```

Fortran

C/C++

Example A.37.1c

```c
#include <stdio.h>
float x, y;
#pragma omp threadprivate(x, y)

void init(float a, float b ) {
    #pragma omp single copyprivate(a,b,x,y)
    {
        scanf("%f %f %f %f", &a, &b, &x, &y);
    }
}
```

C/C++

Fortran

Example A.37.1f

```fortran
      SUBROUTINE INIT(A,B)
      REAL A, B
        COMMON /XY/ X,Y
!$OMP   THREADPRIVATE (/XY/)

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

      END SUBROUTINE INIT
```

Fortran

Appendix A Examples **251**

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 variable.

C/C++

Example A.37.2c

```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 */


  #pragma omp master
  {
    scanf("%f", tmp);
  }

  #pragma omp barrier
  return_val = *tmp;
  #pragma omp barrier

  #pragma omp single nowait
  {
    free(tmp);
  }

  return return_val;
}
```

C/C++

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 variable.

C/C++

Example A.37.2c

#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 */

  #pragma omp master
  {
    scanf("%f", tmp);
  }

  #pragma omp barrier
  return_val = *tmp;
  #pragma omp barrier

  #pragma omp single nowait
  {
    free(tmp);
  }

  return return_val;
}

C/C++

29 252 OpenMP API • Version 3.0 May 2008

Fortran

Example A.37.2f

```fortran
      REAL FUNCTION READ_NEXT()
      REAL, POINTER :: TMP

!$OMP SINGLE
        ALLOCATE (TMP)
!$OMP END SINGLE COPYPRIVATE (TMP)  ! copies the pointer only

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

!$OMP BARRIER
        READ_NEXT = TMP
!$OMP BARRIER

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

Fortran

Suppose that the number of lock variables 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 variables that are allocated within that **parallel** region.

C/C++

Example A.37.3c

```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;
}
```

C/C++

Fortran

Example A.37.2f

```fortran
      REAL FUNCTION READ_NEXT()
      REAL, POINTER :: TMP

!$OMP SINGLE
        ALLOCATE (TMP)
!$OMP END SINGLE COPYPRIVATE (TMP)  ! copies the pointer only

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

!$OMP BARRIER
        READ_NEXT = TMP
!$OMP BARRIER

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

Fortran

Suppose that the number of lock variables 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 variables that are allocated within that **parallel** region.

C/C++

Example A.37.3c

```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;
}
```

C/C++

Appendix A Examples 253

Fortran

Example A.37.3f

```
      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.37.4f

```
      SUBROUTINE S(N)
      INTEGER N

        REAL, DIMENSION(:), ALLOCATABLE :: A
        REAL, DIMENSION(:), POINTER :: B

        ALLOCATE (A(N))
!$OMP   SINGLE
          ALLOCATE (B(N))
          READ (11) A,B
!$OMP   END SINGLE COPYPRIVATE(A,B)
        ! Variable A is private and is
        ! assigned the same value in each thread
        ! Variable B is shared

!$OMP   BARRIER
!$OMP   SINGLE
          DEALLOCATE (B)
!$OMP   END SINGLE NOWAIT
      END SUBROUTINE S
```

Fortran

Fortran

Example A.37.3f

```
      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.37.4f

```
      SUBROUTINE S(N)
      INTEGER N

        REAL, DIMENSION(:), ALLOCATABLE :: A
        REAL, DIMENSION(:), POINTER :: B

        ALLOCATE (A(N))
!$OMP   SINGLE
          ALLOCATE (B(N))
          READ (11) A,B
!$OMP   END SINGLE COPYPRIVATE(A,B)
        ! Variable A is private and is
        ! assigned the same value in each thread
        ! Variable B is shared

!$OMP   BARRIER
!$OMP   SINGLE
          DEALLOCATE (B)
!$OMP   END SINGLE NOWAIT
      END SUBROUTINE S
```

Fortran

33 254 OpenMP API • Version 3.0 May 2008

A.38 Nested Loop Constructs

The following example of loop construct nesting (see Section 2.10 on page 104) is conforming because the inner and outer loop regions bind to different **`parallel`** regions:

C/C++

Example A.38.1c

```
void work(int i, int j) {}

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);
      }
    }
  }
}
```

C/C++

A.38 Nested Loop Constructs

The following example of loop construct nesting (see Section 2.10 on page 104) is conforming because the inner and outer loop regions bind to different **`parallel`** regions:

C/C++

Example A.38.1c

```
void work(int i, int j) {}

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);
      }
    }
  }
}
```

C/C++

Appendix A Examples 255

Fortran

Example A.38.1f

```
      SUBROUTINE WORK(I, J)
      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
```

Fortran

Fortran

Example A.38.1f

```
      SUBROUTINE WORK(I, J)
      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
```

Fortran

256 OpenMP API • Version 3.0 May 2008

The following variation of the preceding example is also conforming:

C/C++

Example A.38.2c

```c
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);
  }
}
```

C/C++

The following variation of the preceding example is also conforming:

C/C++

Example A.38.2c

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);
  }
}

C/C++

Appendix A Examples 257

Fortran

Example A.38.2f

```fortran
      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

A.39 Restrictions on Nesting of Regions

The examples in this section illustrate the region nesting rules. For more information on region nesting, see Section 2.10 on page 104.

Fortran

Example A.38.2f

```fortran
      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

A.39 Restrictions on Nesting of Regions

The examples in this section illustrate the region nesting rules. For more information on region nesting, see Section 2.10 on page 104.

258 OpenMP API • Version 3.0 May 2008

The following example is non-conforming because the inner and outer loop regions are closely nested:

C/C++

Example A.39.1c

```c
void work(int i, int j) {}

void wrong1(int n)
{
  #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++

Fortran

Example A.39.1f

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

      SUBROUTINE WRONG1(N)
      INTEGER N

        INTEGER I,J
!$OMP   PARALLEL DEFAULT(SHARED)
!$OMP     DO
          DO I = 1, N
!$OMP       DO             ! incorrect nesting of loop regions
            DO J = 1, N
              CALL WORK(I,J)
            END DO
          END DO
!$OMP   END PARALLEL
      END SUBROUTINE WRONG1
```

Fortran

The following example is non-conforming because the inner and outer loop regions are closely nested:

C/C++

Example A.39.1c

```c
void work(int i, int j) {}

void wrong1(int n)
{
  #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++

Fortran

Example A.39.1f

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

      SUBROUTINE WRONG1(N)
      INTEGER N

        INTEGER I,J
!$OMP   PARALLEL DEFAULT(SHARED)
!$OMP     DO
          DO I = 1, N
!$OMP       DO             ! incorrect nesting of loop regions
            DO J = 1, N
              CALL WORK(I,J)
            END DO
          END DO
!$OMP   END PARALLEL
      END SUBROUTINE WRONG1
```

Fortran

Appendix A Examples 259

The following orphaned version of the preceding example is also non-conforming:

C/C++

Example A.39.2c

```c
void work(int i, int j) {}
void work1(int i, int n)
{
  int j;
/* incorrect nesting of loop regions */
  #pragma omp for
    for (j=0; j<n; j++)
      work(i, j);
}

void wrong2(int n)
{
  #pragma omp parallel default(shared)
  {
    int i;
    #pragma omp for
      for (i=0; i<n; i++)
         work1(i, n);
  }
}
```

C/C++

Fortran

Example A.39.2f

```fortran
       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
```

Fortran

The following orphaned version of the preceding example is also non-conforming:

C/C++

Example A.39.2c

```c
void work(int i, int j) {}
void work1(int i, int n)
{
  int j;
/* incorrect nesting of loop regions */
  #pragma omp for
    for (j=0; j<n; j++)
      work(i, j);
}

void wrong2(int n)
{
  #pragma omp parallel default(shared)
  {
    int i;
    #pragma omp for
      for (i=0; i<n; i++)
         work1(i, n);
  }
}
```

C/C++

Fortran

Example A.39.2f

```fortran
       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
```

Fortran

260 OpenMP API • Version 3.0 May 2008

The following example is non-conforming because the loop and **single** regions are closely nested:

C/C++

Example A.39.3c

```c
void work(int i, int j) {}
void wrong3(int n)
{
  #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);
      }
  }
}
```

C/C++

Fortran

Example A.39.3f

```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
```

Fortran

The following example is non-conforming because the loop and **single** regions are closely nested:

C/C++

Example A.39.3c

```c
void work(int i, int j) {}
void wrong3(int n)
{
  #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);
      }
  }
}
```

C/C++

Fortran

Example A.39.3f

```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
```

Fortran

Appendix A Examples 261

The following example is non-conforming because a **barrier** region cannot be closely nested inside a loop region:

C/C++

Example A.39.4c

```
void work(int i, int j) {}
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
        work(i, 1);
      }
  }
}
```

C/C++

Fortran

Example A.39.4f

```
      SUBROUTINE WRONG4(N)
      INTEGER N

        INTEGER I
!$OMP   PARALLEL DEFAULT(SHARED)
!$OMP     DO
          DO I = 1, N
            CALL WORK(I, 1)
! incorrect nesting of barrier region in a loop region
!$OMP       BARRIER
            CALL WORK(I, 2)
          END DO
!$OMP   END PARALLEL
      END SUBROUTINE WRONG4
```

Fortran

The following example is non-conforming because a **barrier** region cannot be closely nested inside a loop region:

C/C++

Example A.39.4c

```
void work(int i, int j) {}
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
        work(i, 1);
      }
  }
}
```

C/C++

Fortran

Example A.39.4f

```
      SUBROUTINE WRONG4(N)
      INTEGER N

        INTEGER I
!$OMP   PARALLEL DEFAULT(SHARED)
!$OMP     DO
          DO I = 1, N
            CALL WORK(I, 1)
! incorrect nesting of barrier region in a loop region
!$OMP       BARRIER
            CALL WORK(I, 2)
          END DO
!$OMP   END PARALLEL
      END SUBROUTINE WRONG4
```

Fortran

36 **262** OpenMP API • Version 3.0 May 2008

The following example is non-conforming because the **barrier** region cannot be closely nested inside the **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 **critical** region:

C/C++

Example A.39.5c

```c
void work(int i, int j) {}
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);
    }
  }
}
```

C/C++

Fortran

Example A.39.5f

```fortran
      SUBROUTINE WRONG5(N)
      INTEGER N

!$OMP   PARALLEL DEFAULT(SHARED)
!$OMP     CRITICAL
            CALL WORK(N,1)
! incorrect nesting of barrier region in a critical region
!$OMP       BARRIER
            CALL WORK(N,2)
!$OMP     END CRITICAL
!$OMP   END PARALLEL
      END SUBROUTINE WRONG5
```

Fortran

The following example is non-conforming because the **barrier** region cannot be closely nested inside the **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 **critical** region:

C/C++

Example A.39.5c

```c
void work(int i, int j) {}
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);
    }
  }
}
```

C/C++

Fortran

Example A.39.5f

```fortran
      SUBROUTINE WRONG5(N)
      INTEGER N

!$OMP   PARALLEL DEFAULT(SHARED)
!$OMP     CRITICAL
            CALL WORK(N,1)
! incorrect nesting of barrier region in a critical region
!$OMP       BARRIER
            CALL WORK(N,2)
!$OMP     END CRITICAL
!$OMP   END PARALLEL
      END SUBROUTINE WRONG5
```

Fortran

Appendix A Examples 263

The following example is non-conforming because the **barrier** region cannot be closely nested inside the **single** region. If this were permitted, it would result in deadlock due to the fact that only one thread executes the **single** region:

C/C++

Example A.39.6c

```
void work(int i, int j) {}
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);
    }
  }
}
```

C/C++

Fortran

Example A.39.6f

```
     SUBROUTINE WRONG6(N)
      INTEGER N

!$OMP   PARALLEL DEFAULT(SHARED)
!$OMP     SINGLE
            CALL WORK(N,1)
! incorrect nesting of barrier region in a single region
!$OMP       BARRIER
            CALL WORK(N,2)
!$OMP     END SINGLE
!$OMP   END PARALLEL
      END SUBROUTINE WRONG6
```

Fortran

The following example is non-conforming because the **barrier** region cannot be closely nested inside the **single** region. If this were permitted, it would result in deadlock due to the fact that only one thread executes the **single** region:

C/C++

Example A.39.6c

```
void work(int i, int j) {}
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);
    }
  }
}
```

C/C++

Fortran

Example A.39.6f

```
     SUBROUTINE WRONG6(N)
      INTEGER N

!$OMP   PARALLEL DEFAULT(SHARED)
!$OMP     SINGLE
            CALL WORK(N,1)
! incorrect nesting of barrier region in a single region
!$OMP       BARRIER
            CALL WORK(N,2)
!$OMP     END SINGLE
!$OMP   END PARALLEL
      END SUBROUTINE WRONG6
```

Fortran

34 264 OpenMP API • Version 3.0 May 2008

A.40 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 117), and **`omp_set_num_threads`** (Section 3.2.1 on page 110).

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++

Example A.40.1c

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

void do_by_16(float *x, int iam, int ipoints) {}

void a40(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);
  }
}
```

C/C++

A.40 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 117), and **`omp_set_num_threads`** (Section 3.2.1 on page 110).

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++

Example A.40.1c

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

void do_by_16(float *x, int iam, int ipoints) {}

void a40(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);
  }
}
```

C/C++

Appendix A Examples 265

Fortran

Example A.40.1f

```
      SUBROUTINE DO_BY_16(X, IAM, IPOINTS)
        REAL X(*)
        INTEGER IAM, IPOINTS
      END SUBROUTINE DO_BY_16

      SUBROUTINE SUBA40(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 SUBA40
```

Fortran

A.41 The `omp_get_num_threads` Routine

In the following example, the **`omp_get_num_threads`** call (see Section 3.2.2 on page 111) returns 1 in the sequential part of the code, so *np* will always be equal to 1. To determine the number of threads that will be deployed for the **`parallel`** region, the call should be inside the **`parallel`** region.

Fortran

Example A.40.1f

SUBROUTINE DO_BY_16(X, IAM, IPOINTS)
REAL X(*)
INTEGER IAM, IPOINTS
END SUBROUTINE DO_BY_16
SUBROUTINE SUBA40(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 SUBA40

Fortran

A.41 The omp_get_num_threads Routine

In the following example, the omp_get_num_threads call (see Section 3.2.2 on page 111) returns 1 in the sequential part of the code, so np will always be equal to 1. To determine the number of threads that will be deployed for the parallel region, the call should be inside the parallel region.

266 OpenMP API • Version 3.0 May 2008

C/C++

Example A.41.1c

```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);
}
```

C/C++

Fortran

Example A.41.1f

```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
```

Fortran

C/C++

Example A.41.1c

```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);
}
```

C/C++

Fortran

Example A.41.1f

```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
```

Fortran

Appendix A Examples 267

The following example shows how to rewrite this program without including a query for the number of threads:

C/C++

Example A.41.2c

```c
#include <omp.h>
void work(int i);

void correct()
{
  int i;

  #pragma omp parallel private(i)
  {
    i = omp_get_thread_num();
    work(i);
  }
}
```

C/C++

Fortran

Example A.41.2f

```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
```

Fortran

The following example shows how to rewrite this program without including a query for the number of threads:

C/C++

Example A.41.2c

```c
#include <omp.h>
void work(int i);

void correct()
{
  int i;

  #pragma omp parallel private(i)
  {
    i = omp_get_thread_num();
    work(i);
  }
}
```

C/C++

Fortran

Example A.41.2f

```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
```

Fortran

33 **268** OpenMP API • Version 3.0 May 2008

A.42 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 136).

C/C++

Example A.42.1c

```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;
}
```

C/C++

Fortran

Example A.42.1f

```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
```

Fortran

A.42 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 136).

C/C++

Example A.42.1c

```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;
}
```

C/C++

Fortran

Example A.42.1f

```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
```

Fortran

Appendix A Examples 269

A.43 Ownership of Locks

Ownership of locks has changed from OpenMP 2.5 to OpenMP 3.0. In OpenMP 2.5, locks are owned by threads; so a lock released by the **`omp_unset_lock`** routine must be owned by the same thread executing the routine. In OpenMP 3.0, on the other hand, locks are owned by task regions; so a lock released by the **`omp_unset_lock`** routine in a task region must be owned by the same task region.

This change in ownership requires extra care when using locks. The following program is conforming in OpenMP 2.5 because the thread that releases the lock *lck* in the parallel region is the same thread that acquired the lock in the sequential part of the program (master thread of parallel region and the initial thread are the same). However, it is not conforming in OpenMP 3.0, because the task region that releases the lock *lck* is different from the task region that acquires the lock.

C/C++

Example A.43.1c

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

int main()
{
  int x;
  omp_lock_t lck;

  omp_init_lock (&lck);
  omp_set_lock (&lck);
  x = 0;

#pragma omp parallel shared (x)
  {
    #pragma omp master
      {
        x = x + 1;
        omp_unset_lock (&lck);
      }

    /* Some more stuff. */
  }

  omp_destroy_lock (&lck);
}
```

C/C++

A.43 Ownership of Locks

Ownership of locks has changed from OpenMP 2.5 to OpenMP 3.0. In OpenMP 2.5, locks are owned by threads; so a lock released by the **`omp_unset_lock`** routine must be owned by the same thread executing the routine. In OpenMP 3.0, on the other hand, locks are owned by task regions; so a lock released by the **`omp_unset_lock`** routine in a task region must be owned by the same task region.

This change in ownership requires extra care when using locks. The following program is conforming in OpenMP 2.5 because the thread that releases the lock *lck* in the parallel region is the same thread that acquired the lock in the sequential part of the program (master thread of parallel region and the initial thread are the same). However, it is not conforming in OpenMP 3.0, because the task region that releases the lock *lck* is different from the task region that acquires the lock.

C/C++

Example A.43.1c

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

int main()
{
  int x;
  omp_lock_t lck;

  omp_init_lock (&lck);
  omp_set_lock (&lck);
  x = 0;

#pragma omp parallel shared (x)
  {
    #pragma omp master
      {
        x = x + 1;
        omp_unset_lock (&lck);
      }

    /* Some more stuff. */
  }

  omp_destroy_lock (&lck);
}
```

C/C++

270 OpenMP API • Version 3.0 May 2008

Fortran

Example A.43.1f

```
      program lock
      use omp_lib
      integer :: x
      integer (kind=omp_lock_kind) :: lck

      call omp_init_lock (lck)
      call omp_set_lock(lck)
      x = 0

!$omp parallel shared (x)
!$omp master
      x = x + 1
      call omp_unset_lock(lck)
!$omp end master

!     Some more stuff.
!$omp end parallel

      call omp_destroy_lock(lck)
      end
```

Fortran

A.44 Simple Lock Routines

In the following example (for Section 3.3 on page 134), 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.

Fortran

Example A.43.1f

```
      program lock
      use omp_lib
      integer :: x
      integer (kind=omp_lock_kind) :: lck

      call omp_init_lock (lck)
      call omp_set_lock(lck)
      x = 0

!$omp parallel shared (x)
!$omp master
      x = x + 1
      call omp_unset_lock(lck)
!$omp end master

!     Some more stuff.
!$omp end parallel

      call omp_destroy_lock(lck)
      end
```

Fortran

A.44 Simple Lock Routines

In the following example (for Section 3.3 on page 134), 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.

Appendix A Examples 271

C/C++

Note that the argument to the lock routines should have type **omp_lock_t**, and that there is no need to flush it.

Example A.44.1c

```
#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);
  }

  omp_destroy_lock(&lck);

  return 0;
}
```

C/C++

C/C++

Note that the argument to the lock routines should have type **omp_lock_t**, and that there is no need to flush it.

Example A.44.1c

```
#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);
  }

  omp_destroy_lock(&lck);

  return 0;
}
```

C/C++

33 272 OpenMP API • Version 3.0 May 2008

Fortran

Note that there is no need to flush the lock variable.

Example A.44.1f

```fortran
      SUBROUTINE SKIP(ID)
      END SUBROUTINE SKIP

      SUBROUTINE WORK(ID)
      END SUBROUTINE WORK

      PROGRAM A44

        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 )

      END PROGRAM A44
```

Fortran

Fortran

Note that there is no need to flush the lock variable.

Example A.44.1f

```fortran
      SUBROUTINE SKIP(ID)
      END SUBROUTINE SKIP

      SUBROUTINE WORK(ID)
      END SUBROUTINE WORK

      PROGRAM A44

        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 )

      END PROGRAM A44
```

Fortran

Appendix A Examples 273

A.45 Nestable Lock Routines

The following example (for Section 3.3 on page 134) demonstrates how a nestable lock can be used to synchronize updates both to a whole structure and to one of its members.

C/C++

Example A.45.1c

```
#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 a45(pair *p)
{
  #pragma omp parallel sections
  {
    #pragma omp section
      incr_pair(p, work1(), work2());
    #pragma omp section
      incr_b(p, work3());
  }
}
```

C/C++

A.45 Nestable Lock Routines

The following example (for Section 3.3 on page 134) demonstrates how a nestable lock can be used to synchronize updates both to a whole structure and to one of its members.

C/C++

Example A.45.1c

```
#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 a45(pair *p)
{
  #pragma omp parallel sections
  {
    #pragma omp section
      incr_pair(p, work1(), work2());
    #pragma omp section
      incr_b(p, work3());
  }
}
```

C/C++

274 OpenMP API • Version 3.0 May 2008

Fortran

Example A.45.1f

```
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 A45(P)
  USE OMP_LIB        ! or INCLUDE "omp_lib.h"
  USE DATA
  TYPE(LOCKED_PAIR) :: P
  INTEGER WORK1, WORK2, WORK3
  EXTERNAL WORK1, WORK2, WORK3
```


Fortran

Example A.45.1f

```
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 A45(P)
  USE OMP_LIB        ! or INCLUDE "omp_lib.h"
  USE DATA
  TYPE(LOCKED_PAIR) :: P
  INTEGER WORK1, WORK2, WORK3
  EXTERNAL WORK1, WORK2, WORK3
```

Appendix A Examples 275

```
!$OMP   PARALLEL SECTIONS

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

        END SUBROUTINE A45
```

Fortran


```
!$OMP   PARALLEL SECTIONS

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

        END SUBROUTINE A45
```

Fortran

276 OpenMP API • Version 3.0 May 2008

APPENDIX **B**

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.

In an actual implementation the lock variable might be used to hold the address of an allocated memory block, 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.

Fortran

Note – In order to be able to compile the Fortran stubs file, the include file **`omp_lib.h`** was split into two files: **`omp_lib_kinds.h`** and **`omp_lib.h`** and the **`omp_lib_kinds.h`** file included where needed. There is no requirement for the implementation to provide separate files.

Fortran

APPENDIX **B**

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.

In an actual implementation the lock variable might be used to hold the address of an allocated memory block, 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.

Fortran

Note – In order to be able to compile the Fortran stubs file, the include file **`omp_lib.h`** was split into two files: **`omp_lib_kinds.h`** and **`omp_lib.h`** and the **`omp_lib_kinds.h`** file included where needed. There is no requirement for the implementation to provide separate files.

Fortran

277

B.1 C/C++ Stub Routines

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

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;
}

int omp_in_parallel(void)
{
    return 0;
}

void omp_set_dynamic(int dynamic_threads)
{
}

int omp_get_dynamic(void)
{
    return 0;
}

void omp_set_nested(int nested)
{
}
```


B.1 C/C++ Stub Routines

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

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;
}

int omp_in_parallel(void)
{
    return 0;
}

void omp_set_dynamic(int dynamic_threads)
{
}

int omp_get_dynamic(void)
{
    return 0;
}

void omp_set_nested(int nested)
{
}
```

278 OpenMP API • Version 3.0 May 2008

```c
int omp_get_nested(void)
{
    return 0;
}

void omp_set_schedule(omp_sched_t kind, int modifier)
{
}

void omp_get_schedule(omp_sched_t *kind, int *modifier)
{
    *kind = omp_sched_static;
    *modifier = 0;
}

int omp_get_thread_limit(void)
{
    return 1;
}

void omp_set_max_active_levels(int max_active_levels)
{
}

int omp_get_max_active_levels(void)
{
    return 0;
}

int omp_get_level(void)
{
    return 0;
}

int omp_get_ancestor_thread_num(int level)
{
    if (level == 0)
    {
        return 0;
    }
    else
    {
        return -1;
    }
}
```



```c
int omp_get_nested(void)
{
    return 0;
}

void omp_set_schedule(omp_sched_t kind, int modifier)
{
}

void omp_get_schedule(omp_sched_t *kind, int *modifier)
{
    *kind = omp_sched_static;
    *modifier = 0;
}

int omp_get_thread_limit(void)
{
    return 1;
}

void omp_set_max_active_levels(int max_active_levels)
{
}

int omp_get_max_active_levels(void)
{
    return 0;
}

int omp_get_level(void)
{
    return 0;
}

int omp_get_ancestor_thread_num(int level)
{
    if (level == 0)
    {
        return 0;
    }
    else
    {
        return -1;
    }
}
```

Appendix B Stubs for Runtime Library Routines 279

```c
int omp_get_team_size(int level)
{
    if (level == 0)
    {
        return 1;
    }
    else
    {
        return -1;
    }
}

int omp_get_active_level(void)
{
    return 0;
}

struct __omp_lock
{
    int lock;
};

enum { UNLOCKED = -1, INIT, LOCKED };

void omp_init_lock(omp_lock_t *arg)
{
    struct __omp_lock *lock = (struct __omp_lock *)arg;
    lock->lock = UNLOCKED;
}

void omp_destroy_lock(omp_lock_t *arg)
{
    struct __omp_lock *lock = (struct __omp_lock *)arg;
    lock->lock = INIT;
}

void omp_set_lock(omp_lock_t *arg)
{
    struct __omp_lock *lock = (struct __omp_lock *)arg;
    if (lock->lock == UNLOCKED)
    {
        lock->lock = LOCKED;
    }
    else if (lock->lock == LOCKED)
    {
        fprintf(stderr,
           "error: deadlock in using lock variable\n");
        exit(1);
    }
    else
    {
        fprintf(stderr, "error: lock not initialized\n");
        exit(1);
    }
```



```c
int omp_get_team_size(int level)
{
    if (level == 0)
    {
        return 1;
    }
    else
    {
        return -1;
    }
}

int omp_get_active_level(void)
{
    return 0;
}

struct __omp_lock
{
    int lock;
};

enum { UNLOCKED = -1, INIT, LOCKED };

void omp_init_lock(omp_lock_t *arg)
{
    struct __omp_lock *lock = (struct __omp_lock *)arg;
    lock->lock = UNLOCKED;
}

void omp_destroy_lock(omp_lock_t *arg)
{
    struct __omp_lock *lock = (struct __omp_lock *)arg;
    lock->lock = INIT;
}

void omp_set_lock(omp_lock_t *arg)
{
    struct __omp_lock *lock = (struct __omp_lock *)arg;
    if (lock->lock == UNLOCKED)
    {
        lock->lock = LOCKED;
    }
    else if (lock->lock == LOCKED)
    {
        fprintf(stderr,
           "error: deadlock in using lock variable\n");
        exit(1);
    }
    else
    {
        fprintf(stderr, "error: lock not initialized\n");
        exit(1);
    }
```

280 OpenMP API • Version 3.0 May 2008

```c
}

void omp_unset_lock(omp_lock_t *arg)
{
    struct __omp_lock *lock = (struct __omp_lock *)arg;
    if (lock->lock == LOCKED)
    {
        lock->lock = UNLOCKED;
    }
    else if (lock->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 *arg)
{
    struct __omp_lock *lock = (struct __omp_lock *)arg;
    if (lock->lock == UNLOCKED)
    {
        lock->lock = LOCKED;
        return 1;
    }
    else if (lock->lock == LOCKED)
    {
        return 0;
    }
    else
    {
        fprintf(stderr, "error: lock not initialized\n");
        exit(1);
    }
}

struct __omp_nest_lock
{
    short owner;
    short count;
};

enum { NOOWNER = -1, MASTER = 0 };

void omp_init_nest_lock(omp_nest_lock_t *arg)
{
    struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
    nlock->owner = NOOWNER;
    nlock->count = 0;
```



```c
}

void omp_unset_lock(omp_lock_t *arg)
{
    struct __omp_lock *lock = (struct __omp_lock *)arg;
    if (lock->lock == LOCKED)
    {
        lock->lock = UNLOCKED;
    }
    else if (lock->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 *arg)
{
    struct __omp_lock *lock = (struct __omp_lock *)arg;
    if (lock->lock == UNLOCKED)
    {
        lock->lock = LOCKED;
        return 1;
    }
    else if (lock->lock == LOCKED)
    {
        return 0;
    }
    else
    {
        fprintf(stderr, "error: lock not initialized\n");
        exit(1);
    }
}

struct __omp_nest_lock
{
    short owner;
    short count;
};

enum { NOOWNER = -1, MASTER = 0 };

void omp_init_nest_lock(omp_nest_lock_t *arg)
{
    struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
    nlock->owner = NOOWNER;
    nlock->count = 0;
```

Appendix B Stubs for Runtime Library Routines 281

```
}

void omp_destroy_nest_lock(omp_nest_lock_t *arg)
{
    struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
    nlock->owner = NOOWNER;
    nlock->count = UNLOCKED;
}

void omp_set_nest_lock(omp_nest_lock_t *arg)
{
    struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
    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);
    }
}

void omp_unset_nest_lock(omp_nest_lock_t *arg)
{
    struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
    if (nlock->owner == MASTER && 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);
```



```
}

void omp_destroy_nest_lock(omp_nest_lock_t *arg)
{
    struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
    nlock->owner = NOOWNER;
    nlock->count = UNLOCKED;
}

void omp_set_nest_lock(omp_nest_lock_t *arg)
{
    struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
    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);
    }
}

void omp_unset_nest_lock(omp_nest_lock_t *arg)
{
    struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
    if (nlock->owner == MASTER && 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);
```

282 OpenMP API • Version 3.0 May 2008

```
    }
}

int omp_test_nest_lock(omp_nest_lock_t *arg)
{
    struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
    omp_set_nest_lock(arg);
    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.;
}
```



```
    }
}

int omp_test_nest_lock(omp_nest_lock_t *arg)
{
    struct __omp_nest_lock *nlock=(struct __omp_nest_lock *)arg;
    omp_set_nest_lock(arg);
    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.;
}
```

Appendix B Stubs for Runtime Library Routines 283

B.2 Fortran Stub Routines

```
C23456
      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

      logical function omp_in_parallel()
        omp_in_parallel = .false.
        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

      subroutine omp_set_nested(nested)
        logical nested
        return
      end subroutine
```


B.2 Fortran Stub Routines

```
C23456
      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

      logical function omp_in_parallel()
        omp_in_parallel = .false.
        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

      subroutine omp_set_nested(nested)
        logical nested
        return
      end subroutine
```

284 OpenMP API • Version 3.0 May 2008

```fortran
logical function omp_get_nested()
  omp_get_nested = .false.
  return
end function

subroutine omp_set_schedule(kind, modifier)
  include 'omp_lib_kinds.h'
  integer (kind=omp_sched_kind) kind
  integer modifier
  return
end subroutine

subroutine omp_get_schedule(kind, modifier)
  include 'omp_lib_kinds.h'
  integer (kind=omp_sched_kind) kind
  integer modifier

  kind = omp_sched_static
  modifier = 0
  return
end subroutine

integer function omp_get_thread_limit()
  omp_get_thread_limit = 1
  return
end function

subroutine omp_set_max_active_levels( level )
  integer level
end subroutine

integer function omp_get_max_active_levels()
  omp_get_max_active_levels = 0
  return
end function

integer function omp_get_level()
  omp_get_level = 0
  return
end function

integer function omp_get_ancestor_thread_num( level )
  integer level
  if ( level .eq. 0 ) then
     omp_get_ancestor_thread_num = 0
  else
     omp_get_ancestor_thread_num = -1
  end if
  return
end function
```



```fortran
logical function omp_get_nested()
  omp_get_nested = .false.
  return
end function

subroutine omp_set_schedule(kind, modifier)
  include 'omp_lib_kinds.h'
  integer (kind=omp_sched_kind) kind
  integer modifier
  return
end subroutine

subroutine omp_get_schedule(kind, modifier)
  include 'omp_lib_kinds.h'
  integer (kind=omp_sched_kind) kind
  integer modifier

  kind = omp_sched_static
  modifier = 0
  return
end subroutine

integer function omp_get_thread_limit()
  omp_get_thread_limit = 1
  return
end function

subroutine omp_set_max_active_levels( level )
  integer level
end subroutine

integer function omp_get_max_active_levels()
  omp_get_max_active_levels = 0
  return
end function

integer function omp_get_level()
  omp_get_level = 0
  return
end function

integer function omp_get_ancestor_thread_num( level )
  integer level
  if ( level .eq. 0 ) then
     omp_get_ancestor_thread_num = 0
  else
     omp_get_ancestor_thread_num = -1
  end if
  return
end function
```

Appendix B Stubs for Runtime Library Routines 285

```
integer function omp_get_team_size( level )
  integer level
  if ( level .eq. 0 ) then
     omp_get_team_size = 1
  else
     omp_get_team_size = -1
  end if
  return
end function

integer function omp_get_active_level()
  omp_get_active_level = 0
  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
  include 'omp_lib_kinds.h'
  integer(kind=omp_lock_kind) lock

  lock = -1
  return
end subroutine

subroutine omp_destroy_lock(lock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_lock_kind) lock

  lock = 0
  return
end subroutine

subroutine omp_set_lock(lock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_lock_kind) lock

  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
```



```
integer function omp_get_team_size( level )
  integer level
  if ( level .eq. 0 ) then
     omp_get_team_size = 1
  else
     omp_get_team_size = -1
  end if
  return
end function

integer function omp_get_active_level()
  omp_get_active_level = 0
  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
  include 'omp_lib_kinds.h'
  integer(kind=omp_lock_kind) lock

  lock = -1
  return
end subroutine

subroutine omp_destroy_lock(lock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_lock_kind) lock

  lock = 0
  return
end subroutine

subroutine omp_set_lock(lock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_lock_kind) lock

  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
```

286 OpenMP API • Version 3.0 May 2008

```
subroutine omp_unset_lock(lock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_lock_kind) lock

  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)
  include 'omp_lib_kinds.h'
  integer(kind=omp_lock_kind) lock

  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
  include 'omp_lib_kinds.h'
  integer(kind=omp_nest_lock_kind) nlock

  nlock = -1

  return
end subroutine
```


subroutine omp_unset_lock(lock) … end subroutine (entire page repeated)
Appendix B Stubs for Runtime Library Routines 287

```
subroutine omp_destroy_nest_lock(nlock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_nest_lock_kind) nlock

  nlock = 0

  return
end subroutine

subroutine omp_set_nest_lock(nlock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_nest_lock_kind) nlock

  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)
  include 'omp_lib_kinds.h'
  integer(kind=omp_nest_lock_kind) nlock

  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
```


subroutine omp_destroy_nest_lock(nlock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_nest_lock_kind) nlock

  nlock = 0

  return
end subroutine

subroutine omp_set_nest_lock(nlock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_nest_lock_kind) nlock

  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)
  include 'omp_lib_kinds.h'
  integer(kind=omp_nest_lock_kind) nlock

  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

35 288 OpenMP API • Version 3.0 May 2008

```fortran
integer function omp_test_nest_lock(nlock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_nest_lock_kind) nlock

  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 function


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_get_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_get_wtick = one_year

  return
end function
```



```fortran
integer function omp_test_nest_lock(nlock)
  include 'omp_lib_kinds.h'
  integer(kind=omp_nest_lock_kind) nlock

  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 function


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_get_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_get_wtick = one_year

  return
end function
```

Appendix B Stubs for Runtime Library Routines 289

290 OpenMP API • Version 3.0 May 2008

APPENDIX **C**

OpenMP C and C++ Grammar

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 $term_{opt}$ indicates that the term is optional within the replacement.

The syntactic expression $term_{optseq}$ is equivalent to $term\text{-}seq_{opt}$ with the following additional rules:

term-seq :

  term

  term-seq term

  term-seq , term

APPENDIX **C**

OpenMP C and C++ Grammar

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 $term_{opt}$ indicates that the term is optional within the replacement.

The syntactic expression $term_{optseq}$ is equivalent to $term\text{-}seq_{opt}$ with the following additional rules:

term-seq :

  term

  term-seq term

  term-seq , term

291

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

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

292 OpenMP API • Version 3.0 May 2008

statement:
/* standard statements */
openmp-construct
openmp-construct:
parallel-construct
for-construct
sections-construct
single-construct
parallel-for-construct
parallel-sections-construct
task-construct
master-construct
critical-construct
atomic-construct
ordered-construct
openmp-directive:
barrier-directive
taskwait-directive
flush-directive
structured-block:
statement
parallel-construct:
parallel-directive structured-block
parallel-directive:
`# pragma omp parallel` *parallel-clause*$_{optseq}$ *new-line*

statement:
/* standard statements */
openmp-construct
openmp-construct:
parallel-construct
for-construct
sections-construct
single-construct
parallel-for-construct
parallel-sections-construct
task-construct
master-construct
critical-construct
atomic-construct
ordered-construct
openmp-directive:
barrier-directive
taskwait-directive
flush-directive
structured-block:
statement
parallel-construct:
parallel-directive structured-block
parallel-directive:
`# pragma omp parallel` *parallel-clause*$_{optseq}$ *new-line*

Appendix C OpenMP C and C++ Grammar 293

```
parallel-clause:
    unique-parallel-clause
    data-default-clause
    data-privatization-clause
    data-privatization-in-clause
    data-sharing-clause
    data-reduction-clause
unique-parallel-clause:
    if ( expression )
    num_threads ( expression )
    copyin ( variable-list )
for-construct:
    for-directive iteration-statement
for-directive:
    # pragma omp for for-clause_optseq new-line
for-clause:
    unique-for-clause
    data-privatization-clause
    data-privatization-in-clause
    data-privatization-out-clause
    data-reduction-clause
    nowait
unique-for-clause:
    ordered
    schedule ( schedule-kind )
    schedule ( schedule-kind , expression )
    collapse ( expression )
```



```
parallel-clause:
    unique-parallel-clause
    data-default-clause
    data-privatization-clause
    data-privatization-in-clause
    data-sharing-clause
    data-reduction-clause
unique-parallel-clause:
    if ( expression )
    num_threads ( expression )
    copyin ( variable-list )
for-construct:
    for-directive iteration-statement
for-directive:
    # pragma omp for for-clause_optseq new-line
for-clause:
    unique-for-clause
    data-privatization-clause
    data-privatization-in-clause
    data-privatization-out-clause
    data-reduction-clause
    nowait
unique-for-clause:
    ordered
    schedule ( schedule-kind )
    schedule ( schedule-kind , expression )
    collapse ( expression )
```

294 OpenMP API • Version 3.0 May 2008

schedule-kind:

  `static`

  `dynamic`

  `guided`

  `auto`

  `runtime`

sections-construct:

  sections-directive section-scope

sections-directive:

  `# pragma omp sections` *sections-clause*$_{optseq}$ *new-line*

sections-clause:

  data-privatization-clause

  data-privatization-in-clause

  data-privatization-out-clause

  data-reduction-clause

  `nowait`

section-scope:

  { section-sequence }

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*

schedule-kind:

  `static`

  `dynamic`

  `guided`

  `auto`

  `runtime`

sections-construct:

  sections-directive section-scope

sections-directive:

  `# pragma omp sections` *sections-clause*$_{optseq}$ *new-line*

sections-clause:

  data-privatization-clause

  data-privatization-in-clause

  data-privatization-out-clause

  data-reduction-clause

  `nowait`

section-scope:

  { section-sequence }

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*

Appendix C OpenMP C and C++ Grammar **295**

single-clause:
  unique-single-clause
  data-privatization-clause
  data-privatization-in-clause
  nowait
unique-single-clause:
  copyprivate (*variable-list* **)**
task-construct:
  task-directive structured-block
task-directive:
  # pragma omp task *task-clause*$_{optseq}$ *new-line*
task-clause:
  unique-task-clause
  data-default-clause
  data-privatization-clause
  data-privatization-in-clause
  data-sharing-clause
unique-task-clause:
  if (*scalar-expression* **)**
  untied
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-default-clause

single-clause:
  unique-single-clause
  data-privatization-clause
  data-privatization-in-clause
  nowait
unique-single-clause:
  copyprivate (*variable-list* **)**
task-construct:
  task-directive structured-block
task-directive:
  # pragma omp task *task-clause*$_{optseq}$ *new-line*
task-clause:
  unique-task-clause
  data-default-clause
  data-privatization-clause
  data-privatization-in-clause
  data-sharing-clause
unique-task-clause:
  if (*scalar-expression* **)**
  untied
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-default-clause

296 OpenMP API • Version 3.0 May 2008

data-privatization-clause

data-privatization-in-clause

data-privatization-out-clause

data-sharing-clause

data-reduction-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-default-clause

data-privatization-clause

data-privatization-in-clause

data-privatization-out-clause

data-sharing-clause

data-reduction-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*$_{opt}$ *new-line*

region-phrase:

(identifier)

data-privatization-clause

data-privatization-in-clause

data-privatization-out-clause

data-sharing-clause

data-reduction-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-default-clause

data-privatization-clause

data-privatization-in-clause

data-privatization-out-clause

data-sharing-clause

data-reduction-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*$_{opt}$ *new-line*

region-phrase:

(identifier)

Appendix C OpenMP C and C++ Grammar 297

barrier-directive:

 # pragma omp barrier *new-line*

taskwait-directive:

 # pragma omp taskwait *new-line*

atomic-construct:

atomic-directive expression-statement

atomic-directive:

 # pragma omp atomic *new-line*

flush-directive:

 # pragma omp flush *flush-vars*$_{opt}$ *new-line*

flush-vars:

(variable-list)

ordered-construct:

ordered-directive structured-block

ordered-directive:

 # pragma omp ordered *new-line*

declaration:

/* standard declarations */

threadprivate-directive

threadprivate-directive:

 # pragma omp threadprivate (*variable-list* **)** *new-line*

data-default-clause:

default (shared)

default (none)

data-privatization-clause:

private (*variable-list* **)**

data-privatization-in-clause:

firstprivate (*variable-list* **)**

barrier-directive:

 # pragma omp barrier *new-line*

taskwait-directive:

 # pragma omp taskwait *new-line*

atomic-construct:

atomic-directive expression-statement

atomic-directive:

 # pragma omp atomic *new-line*

flush-directive:

 # pragma omp flush *flush-vars*$_{opt}$ *new-line*

flush-vars:

(variable-list)

ordered-construct:

ordered-directive structured-block

ordered-directive:

 # pragma omp ordered *new-line*

declaration:

/* standard declarations */

threadprivate-directive

threadprivate-directive:

 # pragma omp threadprivate (*variable-list* **)** *new-line*

data-default-clause:

default (shared)

default (none)

data-privatization-clause:

private (*variable-list* **)**

data-privatization-in-clause:

firstprivate (*variable-list* **)**

298 OpenMP API • Version 3.0 May 2008

data-privatization-out-clause:

`lastprivate` **(** *variable-list* **)**

data-sharing-clause:

`shared` **(** *variable-list* **)**

data-reduction-clause:

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

reduction-operator:

One of: `+ * - & ^ | && ||`

/* in C */

variable-list:

identifier

variable-list **,** *identifier*

/* in C++ */

variable-list:

id-expression

variable-list **,** *id-expression*

data-privatization-out-clause:

`lastprivate` **(** *variable-list* **)**

data-sharing-clause:

`shared` **(** *variable-list* **)**

data-reduction-clause:

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

reduction-operator:

One of: `+ * - & ^ | && ||`

/* in C */

variable-list:

identifier

variable-list **,** *identifier*

/* in C++ */

variable-list:

id-expression

variable-list **,** *id-expression*

Appendix C OpenMP C and C++ Grammar **299**

300 OpenMP API • Version 3.0 May 2008

APPENDIX **D**

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.

APPENDIX **D**

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.

301

D.1 Example of the `omp.h` Header File

```
#ifndef _OMP_H_DEF
#define _OMP_H_DEF

/*
 * define the lock data types
 */
typedef void *omp_lock_t;

typedef void *omp_nest_lock_t;

/*
 * define the schedule kinds
 */
typedef enum omp_sched_t
{
    omp_sched_static = 1,
    omp_sched_dynamic = 2,
    omp_sched_guided = 3,
    omp_sched_auto = 4
/* , Add vendor specific schedule constants here */
} omp_sched_t;


/*
 * exported OpenMP functions
 */
#ifdef __cplusplus
extern          "C"
{
#endif

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 int    omp_get_thread_limit(void);
extern void   omp_set_max_active_levels(int max_active_levels);
extern int    omp_get_max_active_levels(void);
extern int    omp_get_level(void);
extern int    omp_get_ancestor_thread_num(int level);
```


D.1 Example of the `omp.h` Header File

```
#ifndef _OMP_H_DEF
#define _OMP_H_DEF

/*
 * define the lock data types
 */
typedef void *omp_lock_t;

typedef void *omp_nest_lock_t;

/*
 * define the schedule kinds
 */
typedef enum omp_sched_t
{
    omp_sched_static = 1,
    omp_sched_dynamic = 2,
    omp_sched_guided = 3,
    omp_sched_auto = 4
/* , Add vendor specific schedule constants here */
} omp_sched_t;


/*
 * exported OpenMP functions
 */
#ifdef __cplusplus
extern          "C"
{
#endif

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 int    omp_get_thread_limit(void);
extern void   omp_set_max_active_levels(int max_active_levels);
extern int    omp_get_max_active_levels(void);
extern int    omp_get_level(void);
extern int    omp_get_ancestor_thread_num(int level);
```

302 OpenMP API • Version 3.0 May 2008

```
extern int     omp_get_team_size(int level);
extern int     omp_get_active_level(void);
extern void    omp_set_schedule(omp_sched_t kind, int modifier);
extern void    omp_get_schedule(omp_sched_t *kind, int *modifier);

extern void    omp_init_lock(omp_lock_t *lock);
extern void    omp_destroy_lock(omp_lock_t *lock);
extern void    omp_set_lock(omp_lock_t *lock);
extern void    omp_unset_lock(omp_lock_t *lock);
extern int     omp_test_lock(omp_lock_t *lock);

extern void    omp_init_nest_lock(omp_nest_lock_t *lock);
extern void    omp_destroy_nest_lock(omp_nest_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);

#ifdef __cplusplus
}
#endif

#endif
```


extern int   omp_get_team_size(int level);
extern int   omp_get_active_level(void);
extern void   omp_set_schedule(omp_sched_t kind, int modifier);
extern void   omp_get_schedule(omp_sched_t *kind, int *modifier);

extern void   omp_init_lock(omp_lock_t *lock);
extern void   omp_destroy_lock(omp_lock_t *lock);
extern void   omp_set_lock(omp_lock_t *lock);
extern void   omp_unset_lock(omp_lock_t *lock);
extern int   omp_test_lock(omp_lock_t *lock);

extern void   omp_init_nest_lock(omp_nest_lock_t *lock);
extern void   omp_destroy_nest_lock(omp_nest_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);

#ifdef __cplusplus
}
#endif

#endif

Appendix D Interface Declarations 303

D.2 Example of an Interface Declaration `include` File

omp_lib_kinds.h:

```
      integer      omp_lock_kind
      parameter ( omp_lock_kind = 8 )
      integer      omp_nest_lock_kind
      parameter ( omp_nest_lock_kind = 8 )

      integer      omp_sched_kind
      parameter ( omp_sched_kind = 4)

      integer ( omp_sched_kind ) omp_sched_static
      parameter ( omp_sched_static = 1 )
      integer ( omp_sched_kind ) omp_sched_dynamic
      parameter ( omp_sched_dynamic = 2 )
      integer ( omp_sched_kind ) omp_sched_guided
      parameter ( omp_sched_guided = 3 )
      integer ( omp_sched_kind ) omp_sched_auto
      parameter ( omp_sched_auto = 4 )
```

omp_lib.h:

```
C                               default integer type assumed below
C                               default logical type assumed below
C                               OpenMP Fortran API v3.0

      include 'omp_lib_kinds.h'
      integer      openmp_version
      parameter ( openmp_version = 200805 )

      external omp_set_num_threads
      external omp_get_num_threads
      integer  omp_get_num_threads
      external omp_get_max_threads
      integer  omp_get_max_threads
      external omp_get_thread_num
      integer  omp_get_thread_num
      external omp_get_num_procs
      integer  omp_get_num_procs
      external omp_in_parallel
      logical  omp_in_parallel
      external omp_set_dynamic
      external omp_get_dynamic
      logical  omp_get_dynamic
      external omp_set_nested
      external omp_get_nested
      logical  omp_get_nested
      external omp_set_schedule
      external omp_get_schedule
      external omp_get_thread_limit
```


D.2 Example of an Interface Declaration `include` File

omp_lib_kinds.h:

```
      integer      omp_lock_kind
      parameter ( omp_lock_kind = 8 )
      integer      omp_nest_lock_kind
      parameter ( omp_nest_lock_kind = 8 )

      integer      omp_sched_kind
      parameter ( omp_sched_kind = 4)

      integer ( omp_sched_kind ) omp_sched_static
      parameter ( omp_sched_static = 1 )
      integer ( omp_sched_kind ) omp_sched_dynamic
      parameter ( omp_sched_dynamic = 2 )
      integer ( omp_sched_kind ) omp_sched_guided
      parameter ( omp_sched_guided = 3 )
      integer ( omp_sched_kind ) omp_sched_auto
      parameter ( omp_sched_auto = 4 )
```

omp_lib.h:

```
C                               default integer type assumed below
C                               default logical type assumed below
C                               OpenMP Fortran API v3.0

      include 'omp_lib_kinds.h'
      integer      openmp_version
      parameter ( openmp_version = 200805 )

      external omp_set_num_threads
      external omp_get_num_threads
      integer  omp_get_num_threads
      external omp_get_max_threads
      integer  omp_get_max_threads
      external omp_get_thread_num
      integer  omp_get_thread_num
      external omp_get_num_procs
      integer  omp_get_num_procs
      external omp_in_parallel
      logical  omp_in_parallel
      external omp_set_dynamic
      external omp_get_dynamic
      logical  omp_get_dynamic
      external omp_set_nested
      external omp_get_nested
      logical  omp_get_nested
      external omp_set_schedule
      external omp_get_schedule
      external omp_get_thread_limit
```

304 OpenMP API • Version 3.0 May 2008

```
integer omp_get_thread_limit
external omp_set_max_active_levels
external omp_get_max_active_levels
integer omp_get_max_active_levels
external omp_get_level
integer omp_get_level
external omp_get_ancestor_thread_num
integer omp_get_ancestor_thread_num
external omp_get_team_size
integer omp_get_team_size
external omp_get_active_level
integer omp_get_active_level

external omp_init_lock
external omp_destroy_lock
external omp_set_lock
external omp_unset_lock
external omp_test_lock
logical  omp_test_lock

external omp_init_nest_lock
external omp_destroy_nest_lock
external omp_set_nest_lock
external omp_unset_nest_lock
external omp_test_nest_lock
integer  omp_test_nest_lock

external omp_get_wtick
double precision  omp_get_wtick
external omp_get_wtime
double precision  omp_get_wtime
```



```
integer omp_get_thread_limit
external omp_set_max_active_levels
external omp_get_max_active_levels
integer omp_get_max_active_levels
external omp_get_level
integer omp_get_level
external omp_get_ancestor_thread_num
integer omp_get_ancestor_thread_num
external omp_get_team_size
integer omp_get_team_size
external omp_get_active_level
integer omp_get_active_level

external omp_init_lock
external omp_destroy_lock
external omp_set_lock
external omp_unset_lock
external omp_test_lock
logical  omp_test_lock

external omp_init_nest_lock
external omp_destroy_nest_lock
external omp_set_nest_lock
external omp_unset_nest_lock
external omp_test_nest_lock
integer  omp_test_nest_lock

external omp_get_wtick
double precision  omp_get_wtick
external omp_get_wtime
double precision  omp_get_wtime
```

Appendix D Interface Declarations 305

D.3 Example of a Fortran 90 Interface Declaration `module`

```
!     the "!" of this comment starts in column 1
!23456

           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
           integer, parameter :: omp_sched_kind = 4
           integer(kind=omp_sched_kind), parameter ::
     &       omp_sched_static = 1
           integer(kind=omp_sched_kind), parameter ::
     &       omp_sched_dynamic = 2
           integer(kind=omp_sched_kind), parameter ::
     &       omp_sched_guided = 3
           integer(kind=omp_sched_kind), parameter ::
     &       omp_sched_auto = 4
           end module omp_lib_kinds

         module omp_lib

            use omp_lib_kinds
!                                   OpenMP Fortran API v3.0
            integer, parameter :: openmp_version = 200805

           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

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

            function omp_get_max_threads ()
             use omp_lib_kinds
             integer (kind=omp_integer_kind) :: omp_get_max_threads
            end function omp_get_max_threads

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


D.3 Example of a Fortran 90 Interface Declaration module

!   the "!" of this comment starts in column 1
!23456

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
integer, parameter :: omp_sched_kind = 4
integer(kind=omp_sched_kind), parameter ::
&   omp_sched_static = 1
integer(kind=omp_sched_kind), parameter ::
&   omp_sched_dynamic = 2
integer(kind=omp_sched_kind), parameter ::
&   omp_sched_guided = 3
integer(kind=omp_sched_kind), parameter ::
&   omp_sched_auto = 4
end module omp_lib_kinds

module omp_lib

use omp_lib_kinds
!   OpenMP Fortran API v3.0
integer, parameter :: openmp_version = 200805

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

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

function omp_get_max_threads ()
use omp_lib_kinds
integer (kind=omp_integer_kind) :: omp_get_max_threads
end function omp_get_max_threads

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

306 OpenMP API • Version 3.0 May 2008

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

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

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

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

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

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

subroutine omp_set_schedule (kind, modifier)
 use omp_lib_kinds
 integer(kind=omp_sched_kind), intent(in) :: kind
 integer(kind=omp_integer_kind), intent(in) :: modifier
end subroutine omp_set_schedule

subroutine omp_get_schedule (kind, modifier)
 use omp_lib_kinds
 integer(kind=omp_sched_kind), intent(out) :: kind
 integer(kind=omp_integer_kind), intent(out)::modifier
end subroutine omp_get_schedule

function omp_get_thread_limit()
 use omp_lib_kinds
 integer (kind=omp_integer_kind) :: omp_get_thread_limit
end function omp_get_thread_limit

subroutine omp_set_max_active_levels(var)
 use omp_lib_kinds
 integer (kind=omp_integer_kind), intent(in) :: var
end subroutine omp_set_max_active_levels
```


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

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

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

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

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

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

subroutine omp_set_schedule (kind, modifier)
 use omp_lib_kinds
 integer(kind=omp_sched_kind), intent(in) :: kind
 integer(kind=omp_integer_kind), intent(in) :: modifier
end subroutine omp_set_schedule

subroutine omp_get_schedule (kind, modifier)
 use omp_lib_kinds
 integer(kind=omp_sched_kind), intent(out) :: kind
 integer(kind=omp_integer_kind), intent(out)::modifier
end subroutine omp_get_schedule

function omp_get_thread_limit()
 use omp_lib_kinds
 integer (kind=omp_integer_kind) :: omp_get_thread_limit
end function omp_get_thread_limit

subroutine omp_set_max_active_levels(var)
 use omp_lib_kinds
 integer (kind=omp_integer_kind), intent(in) :: var
end subroutine omp_set_max_active_levels

Appendix D Interface Declarations 307

```
function omp_get_max_active_levels()
 use omp_lib_kinds
 integer (kind=omp_integer_kind) ::
&    omp_get_max_active_levels
end function omp_get_max_active_levels

function omp_get_level()
 use omp_lib_kinds
 integer (kind=omp_integer_kind) :: omp_get_level
end function omp_get_level

function omp_get_ancestor_thread_num(level)
 use omp_lib_kinds
 integer (kind=omp_integer_kind), intent(in) ::
&    level
 integer (kind=omp_integer_kind) ::
&    omp_get_ancestor_thread_num
end function omp_get_ancestor_thread_num

function omp_get_team_size(level)
 use omp_lib_kinds
 integer (kind=omp_integer_kind), intent(in) ::
&     level
 integer (kind=omp_integer_kind) :: omp_get_team_size
end function omp_get_team_size

function omp_get_active_level()
 use omp_lib_kinds
 integer (kind=omp_integer_kind) ::
&     omp_get_active_level
end function omp_get_active_level


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

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

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

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



```
function omp_get_max_active_levels()
 use omp_lib_kinds
 integer (kind=omp_integer_kind) ::
&    omp_get_max_active_levels
end function omp_get_max_active_levels

function omp_get_level()
 use omp_lib_kinds
 integer (kind=omp_integer_kind) :: omp_get_level
end function omp_get_level

function omp_get_ancestor_thread_num(level)
 use omp_lib_kinds
 integer (kind=omp_integer_kind), intent(in) ::
&    level
 integer (kind=omp_integer_kind) ::
&    omp_get_ancestor_thread_num
end function omp_get_ancestor_thread_num

function omp_get_team_size(level)
 use omp_lib_kinds
 integer (kind=omp_integer_kind), intent(in) ::
&     level
 integer (kind=omp_integer_kind) :: omp_get_team_size
end function omp_get_team_size

function omp_get_active_level()
 use omp_lib_kinds
 integer (kind=omp_integer_kind) ::
&     omp_get_active_level
end function omp_get_active_level


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

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

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

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

44 308 OpenMP API • Version 3.0 May 2008

```fortran
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

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

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

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

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

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

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

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

end interface

end module omp_lib
```



```fortran
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

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

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

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

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

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

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

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

end interface

end module omp_lib
```

Appendix D Interface Declarations 309

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:

```
!     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
```


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:

```
!     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
```

310 OpenMP API • Version 3.0 May 2008

APPENDIX **E**

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.

- **Task scheduling points**: it is implementation defined where task scheduling points occur in untied task regions (see Section 1.3 on page 11).
- **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 13).
- **Internal control variables**: the initial values of *nthreads-var*, *dyn-var*, *run-sched-var*, *def-sched-var*, *stacksize-var*, *wait-policy-var*, *thread-limit-var*, and *max-active-levels-var* are implementation defined (see Section 2.3.2 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 Algorithm 2-1 even if dynamic adjustment is disabled (see Section 2.4.1 on page 35).
- **Loop directive:** the integer type or kind used to compute the iteration count of a collapsed loop is implementation defined. The effect of the `schedule(runtime)` clause when the *run-sched-var* ICV is set to `auto` is implementation defined. See Section 2.5.1 on page 38.
- **`sections` construct**: the method of scheduling the structured blocks among threads in the team is implementation defined (see Section 2.5.2 on page 47).
- **`single` construct**: the method of choosing a thread to execute the structured block is implementation defined (see Section 2.5.3 on page 49).

APPENDIX **E**

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.

- **Task scheduling points**: it is implementation defined where task scheduling points occur in untied task regions (see Section 1.3 on page 11).
- **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 13).
- **Internal control variables**: the initial values of *nthreads-var*, *dyn-var*, *run-sched-var*, *def-sched-var*, *stacksize-var*, *wait-policy-var*, *thread-limit-var*, and *max-active-levels-var* are implementation defined (see Section 2.3.2 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 Algorithm 2-1 even if dynamic adjustment is disabled (see Section 2.4.1 on page 35).
- **Loop directive:** the integer type or kind used to compute the iteration count of a collapsed loop is implementation defined. The effect of the `schedule(runtime)` clause when the *run-sched-var* ICV is set to `auto` is implementation defined. See Section 2.5.1 on page 38.
- **`sections` construct**: the method of scheduling the structured blocks among threads in the team is implementation defined (see Section 2.5.2 on page 47).
- **`single` construct**: the method of choosing a thread to execute the structured block is implementation defined (see Section 2.5.3 on page 49).

311

- **`atomic` construct**: a compliant implementation may enforce exclusive access between **`atomic`** regions which update different storage locations. The circumstances under which this occurs are implementation defined (see Section 2.8.5 on page 69).
- **`omp_set_num_threads` routine**: if the argument is not a positive integer the behavior is implementation defined (see Section 3.2.1 on page 110).
- **`omp_set_schedule` routine**: the behavior for implementation defined schedule types is implementation defined (see Section 3.2.11 on page 121).
- **`omp_set_max_active_levels` routine**: when called from within any explicit parallel region the binding thread set (and binding region, if required) for the **`omp_set_max_active_levels`** region is implementation defined and the behavior is implementation defined. If the argument is not a non-negative integer then the behavior is implementation defined (see Section 3.2.14 on page 126).
- **`omp_get_max_active_levels` routine**: when called from within any explicit parallel region the binding thread set (and binding region, if required) for the **`omp_get_max_active_levels`** region is implementation defined (see Section 3.2.15 on page 127).
- **`OMP_SCHEDULE` environment variable**: if the value of the variable does not conform to the specified format then the result is implementation defined (see Section 4.1 on page 146).
- **`OMP_NUM_THREADS` environment variable**: if the value of the variable is greater than the number of threads the implementation can support or is not a positive integer then the result is implementation defined (see Section 4.2 on page 147).
- **`OMP_DYNAMIC` environment variable**: if the value is neither **true** nor **false** the behavior is implementation defined (see Section 4.3 on page 148).
- **`OMP_NESTED` environment variable**: if the value is neither **true** nor **false** the behavior is implementation defined (see Section 4.4 on page 148).
- **`OMP_STACKSIZE` environment variable**: if the value does not conform to the specified format or the implementation cannot provide a stack of the specified size then the behavior is implementation defined (see Section 4.5 on page 149).
- **`OMP_WAIT_POLICY` environment variable**: the details of the **`ACTIVE`** and **`PASSIVE`** behaviors are implementation defined (see Section 4.6 on page 150).
- **`OMP_MAX_ACTIVE_LEVELS` environment variable**: if the value is not a non-negative integer or is greater than the number of parallel levels an impementation can support then the behavior is implementation defined (see Section 4.7 on page 150).
- **`OMP_THREAD_LIMIT` environment variable**: if the requested value 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 4.8 on page 151).

- **`atomic` construct**: a compliant implementation may enforce exclusive access between **`atomic`** regions which update different storage locations. The circumstances under which this occurs are implementation defined (see Section 2.8.5 on page 69).
- **`omp_set_num_threads` routine**: if the argument is not a positive integer the behavior is implementation defined (see Section 3.2.1 on page 110).
- **`omp_set_schedule` routine**: the behavior for implementation defined schedule types is implementation defined (see Section 3.2.11 on page 121).
- **`omp_set_max_active_levels` routine**: when called from within any explicit parallel region the binding thread set (and binding region, if required) for the **`omp_set_max_active_levels`** region is implementation defined and the behavior is implementation defined. If the argument is not a non-negative integer then the behavior is implementation defined (see Section 3.2.14 on page 126).
- **`omp_get_max_active_levels` routine**: when called from within any explicit parallel region the binding thread set (and binding region, if required) for the **`omp_get_max_active_levels`** region is implementation defined (see Section 3.2.15 on page 127).
- **`OMP_SCHEDULE` environment variable**: if the value of the variable does not conform to the specified format then the result is implementation defined (see Section 4.1 on page 146).
- **`OMP_NUM_THREADS` environment variable**: if the value of the variable is greater than the number of threads the implementation can support or is not a positive integer then the result is implementation defined (see Section 4.2 on page 147).
- **`OMP_DYNAMIC` environment variable**: if the value is neither **true** nor **false** the behavior is implementation defined (see Section 4.3 on page 148).
- **`OMP_NESTED` environment variable**: if the value is neither **true** nor **false** the behavior is implementation defined (see Section 4.4 on page 148).
- **`OMP_STACKSIZE` environment variable**: if the value does not conform to the specified format or the implementation cannot provide a stack of the specified size then the behavior is implementation defined (see Section 4.5 on page 149).
- **`OMP_WAIT_POLICY` environment variable**: the details of the **`ACTIVE`** and **`PASSIVE`** behaviors are implementation defined (see Section 4.6 on page 150).
- **`OMP_MAX_ACTIVE_LEVELS` environment variable**: if the value is not a non-negative integer or is greater than the number of parallel levels an impementation can support then the behavior is implementation defined (see Section 4.7 on page 150).
- **`OMP_THREAD_LIMIT` environment variable**: if the requested value 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 4.8 on page 151).

312 OpenMP API • Version 3.0 May 2008

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 2.9.2 on page 81).
- **`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 2.9.3.2 on page 88).
- **Runtime library definitions**: it is implementation defined whether the include file **`omp_lib.h`** or the module **`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 108).

Fortran

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 2.9.2 on page 81).
- **`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 2.9.3.2 on page 88).
- **Runtime library definitions**: it is implementation defined whether the include file **`omp_lib.h`** or the module **`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 108).

Fortran

Appendix E Implementation Defined Behaviors in OpenMP 313

314 OpenMP API • Version 3.0 May 2008

APPENDIX **F**

Changes from Version 2.5 to Version 3.0

This appendix summarizes the major changes between the OpenMP API Version 2.5 specification and the OpenMP API Version 3.0 specification.

- The concept of tasks has been added to the OpenMP execution model (see Section 1.2.3 on page 8 and Section 1.3 on page 11).
- The **`task`** construct (see Section 2.7 on page 59) has been added, which provides a mechanism for creating tasks explicitly.
- The **`taskwait`** construct (see Section 2.8.4 on page 68) has been added, which causes a task to wait for all its child tasks to complete.
- The OpenMP memory model now covers atomicity of memory accesses (see Section 1.4.1 on page 13). The description of the behavior of **`volatile`** in terms of **`flush`** was removed.
- In Version 2.5, there was a single copy of of the *nest-var*, *dyn-var*, *nthreads-var* and *run-sched-var* internal control variables (ICVs) for the whole program. In Version 3.0, there is one copy of these ICVs per task (see Section 2.3 on page 28). As a result, the **`omp_set_num_threads`**, **`omp_set_nested`** and **`omp_set_dynamic`** runtime library routines now have specified effects when called from inside a **`parallel`** region (See Section 3.2.1 on page 110, Section 3.2.7 on page 117 and Section 3.2.9 on page 119).
- The definition of active **`parallel`** region has been changed: in Version 3.0 a **`parallel`** region is active if it is executed by a team consisting of more than one thread (see Section 1.2.2 on page 2).
- The rules for determining the number of threads used in a **`parallel`** region have been modified (see Section 2.4.1 on page 35).
- In Version 3.0, the assignment of iterations to threads in a loop construct with a **`static`** schedule kind is deterministic (see Section 2.5.1 on page 38).

APPENDIX **F**

Changes from Version 2.5 to Version 3.0

This appendix summarizes the major changes between the OpenMP API Version 2.5 specification and the OpenMP API Version 3.0 specification.

- The concept of tasks has been added to the OpenMP execution model (see Section 1.2.3 on page 8 and Section 1.3 on page 11).
- The **`task`** construct (see Section 2.7 on page 59) has been added, which provides a mechanism for creating tasks explicitly.
- The **`taskwait`** construct (see Section 2.8.4 on page 68) has been added, which causes a task to wait for all its child tasks to complete.
- The OpenMP memory model now covers atomicity of memory accesses (see Section 1.4.1 on page 13). The description of the behavior of **`volatile`** in terms of **`flush`** was removed.
- In Version 2.5, there was a single copy of of the *nest-var*, *dyn-var*, *nthreads-var* and *run-sched-var* internal control variables (ICVs) for the whole program. In Version 3.0, there is one copy of these ICVs per task (see Section 2.3 on page 28). As a result, the **`omp_set_num_threads`**, **`omp_set_nested`** and **`omp_set_dynamic`** runtime library routines now have specified effects when called from inside a **`parallel`** region (See Section 3.2.1 on page 110, Section 3.2.7 on page 117 and Section 3.2.9 on page 119).
- The definition of active **`parallel`** region has been changed: in Version 3.0 a **`parallel`** region is active if it is executed by a team consisting of more than one thread (see Section 1.2.2 on page 2).
- The rules for determining the number of threads used in a **`parallel`** region have been modified (see Section 2.4.1 on page 35).
- In Version 3.0, the assignment of iterations to threads in a loop construct with a **`static`** schedule kind is deterministic (see Section 2.5.1 on page 38).

315

- In Version 3.0, a loop construct may be associated with more than one perfectly nested loop. The number of associated loops may be controlled by the **`collapse`** clause (see Section 2.5.1 on page 38).
- Random access iterators, and variables of unsigned integer type, may now be used as loop iterators in loops associated with a loop construct (see Section 2.5.1 on page 38).
- The schedule kind **`auto`** has been added, which gives the implementation the freedom to choose any possible mapping of iterations in a loop construct to threads in the team (see Section 2.5.1 on page 38).
- Fortran assumed-size arrays now have predetermined data-sharing attributes (see Section 2.9.1.1 on page 78).
- In Fortran, **`firstprivate`** is now permitted as an argument to the **`default`** clause (see Section 2.9.3.1 on page 86).
- For list items in the **`private`** clause, implementations are no longer permitted to use the storage of the original list item to hold the new list item on the master thread. If no attempt is made to reference the original list item inside the **`parallel`** region, its value is well defined on exit from the **`parallel`** region (see Section 2.9.3.3 on page 89).
- In Version 3.0, Fortran allocatable arrays may appear in **`private`**, **`firstprivate`**, **`lastprivate`**, **`reduction`**, **`copyin`** and **`copyprivate`** clauses. (see Section 2.9.2 on page 81, Section 2.9.3.3 on page 89, Section 2.9.3.4 on page 92, Section 2.9.3.5 on page 94, Section 2.9.3.6 on page 96, Section 2.9.4.1 on page 101 and Section 2.9.4.2 on page 102).
- In Version 3.0, static class members variables may appear in a **`threadprivate`** directive (see Section 2.9.2 on page 81).
- Version 3.0 makes clear where, and with which arguments, constructors and destructors of private and threadprivate class type variables are called (see Section 2.9.2 on page 81, Section 2.9.3.3 on page 89, Section 2.9.3.4 on page 92, Section 2.9.4.1 on page 101 and Section 2.9.4.2 on page 102)
- The runtime library routines **`omp_set_schedule`** and **`omp_get_schedule`** have been added; these routines respectively set and retrieve the value of the *run_sched_var* ICV (see Section 3.2.11 on page 121 and Section 3.2.12 on page 123).
- The *thread-limit-var* ICV has been added, which controls the maximium number of threads participating in the OpenMP program. The value of this ICV can be set with the **`OMP_THREAD_LIMIT`** environment variable and retrieved with the **`omp_get_thread_limit`** runtime library routine (see Section 2.3.1 on page 28, Section 3.2.13 on page 125 and Section 4.8 on page 151).
- The *max-active-levels-var* ICV has been added, which controls the number of nested active **`parallel`** regions. The value of this ICV can be set with the **`OMP_MAX_ACTIVE_LEVELS`** environment variable and the **`omp_set_max_active_levels`** runtime library routine, and it can be retrieved

- In Version 3.0, a loop construct may be associated with more than one perfectly nested loop. The number of associated loops may be controlled by the **`collapse`** clause (see Section 2.5.1 on page 38).
- Random access iterators, and variables of unsigned integer type, may now be used as loop iterators in loops associated with a loop construct (see Section 2.5.1 on page 38).
- The schedule kind **`auto`** has been added, which gives the implementation the freedom to choose any possible mapping of iterations in a loop construct to threads in the team (see Section 2.5.1 on page 38).
- Fortran assumed-size arrays now have predetermined data-sharing attributes (see Section 2.9.1.1 on page 78).
- In Fortran, **`firstprivate`** is now permitted as an argument to the **`default`** clause (see Section 2.9.3.1 on page 86).
- For list items in the **`private`** clause, implementations are no longer permitted to use the storage of the original list item to hold the new list item on the master thread. If no attempt is made to reference the original list item inside the **`parallel`** region, its value is well defined on exit from the **`parallel`** region (see Section 2.9.3.3 on page 89).
- In Version 3.0, Fortran allocatable arrays may appear in **`private`**, **`firstprivate`**, **`lastprivate`**, **`reduction`**, **`copyin`** and **`copyprivate`** clauses. (see Section 2.9.2 on page 81, Section 2.9.3.3 on page 89, Section 2.9.3.4 on page 92, Section 2.9.3.5 on page 94, Section 2.9.3.6 on page 96, Section 2.9.4.1 on page 101 and Section 2.9.4.2 on page 102).
- In Version 3.0, static class members variables may appear in a **`threadprivate`** directive (see Section 2.9.2 on page 81).
- Version 3.0 makes clear where, and with which arguments, constructors and destructors of private and threadprivate class type variables are called (see Section 2.9.2 on page 81, Section 2.9.3.3 on page 89, Section 2.9.3.4 on page 92, Section 2.9.4.1 on page 101 and Section 2.9.4.2 on page 102)
- The runtime library routines **`omp_set_schedule`** and **`omp_get_schedule`** have been added; these routines respectively set and retrieve the value of the *run_sched_var* ICV (see Section 3.2.11 on page 121 and Section 3.2.12 on page 123).
- The *thread-limit-var* ICV has been added, which controls the maximium number of threads participating in the OpenMP program. The value of this ICV can be set with the **`OMP_THREAD_LIMIT`** environment variable and retrieved with the **`omp_get_thread_limit`** runtime library routine (see Section 2.3.1 on page 28, Section 3.2.13 on page 125 and Section 4.8 on page 151).
- The *max-active-levels-var* ICV has been added, which controls the number of nested active **`parallel`** regions. The value of this ICV can be set with the **`OMP_MAX_ACTIVE_LEVELS`** environment variable and the **`omp_set_max_active_levels`** runtime library routine, and it can be retrieved

41 **316** OpenMP API • Version 3.0 May 2008

with the **`omp_get_max_active_levels`** runtime library routine (see Section 2.3.1 on page 28, Section 3.2.14 on page 126, Section 3.2.15 on page 127 and Section 4.7 on page 150).

- The *stacksize-var* ICV has been added, which controls the stack size for threads that the OpenMP implementation creates. The value of this ICV can be set with the **`OMP_STACKSIZE`** environment variable (see Section 2.3.1 on page 28 and Section 4.5 on page 149).
- The *wait-policy-var* ICV has been added, which controls the desired behavior of waiting threads. The value of this ICV can be set with the **`OMP_WAIT_POLICY`** environment variable (see Section 2.3.1 on page 28 and Section 4.6 on page 150).
- The **`omp_get_level`** runtime library routine has been added, which returns the number of nested **`parallel`** regions enclosing the task that contains the call (see Section 3.2.16 on page 129).
- The **`omp_get_ancestor_thread_num`** runtime library routine has been added, which returns, for a given nested level of the current thread, the thread number of the ancestor (see Section 3.2.17 on page 130).
- The **`omp_get_team_size`** runtime library routine has been added, which returns, for a given nested level of the current thread, the size of the thread team to which the ancestor belongs (see Section 3.2.18 on page 131).
- The **`omp_get_active_level`** runtime library routine has been added, which returns the number of nested, active **`parallel`** regions enclosing the task that contains the call (see Section 3.2.19 on page 133).
- In Version 3.0, locks are owned by tasks, not by threads (see Section 3.3 on page 134).

with the **`omp_get_max_active_levels`** runtime library routine (see Section 2.3.1 on page 28, Section 3.2.14 on page 126, Section 3.2.15 on page 127 and Section 4.7 on page 150).

- The *stacksize-var* ICV has been added, which controls the stack size for threads that the OpenMP implementation creates. The value of this ICV can be set with the **`OMP_STACKSIZE`** environment variable (see Section 2.3.1 on page 28 and Section 4.5 on page 149).
- The *wait-policy-var* ICV has been added, which controls the desired behavior of waiting threads. The value of this ICV can be set with the **`OMP_WAIT_POLICY`** environment variable (see Section 2.3.1 on page 28 and Section 4.6 on page 150).
- The **`omp_get_level`** runtime library routine has been added, which returns the number of nested **`parallel`** regions enclosing the task that contains the call (see Section 3.2.16 on page 129).
- The **`omp_get_ancestor_thread_num`** runtime library routine has been added, which returns, for a given nested level of the current thread, the thread number of the ancestor (see Section 3.2.17 on page 130).
- The **`omp_get_team_size`** runtime library routine has been added, which returns, for a given nested level of the current thread, the size of the thread team to which the ancestor belongs (see Section 3.2.18 on page 131).
- The **`omp_get_active_level`** runtime library routine has been added, which returns the number of nested, active **`parallel`** regions enclosing the task that contains the call (see Section 3.2.19 on page 133).
- In Version 3.0, locks are owned by tasks, not by threads (see Section 3.3 on page 134).

Appendix F Changes from Version 2.5 to Version 3.0 **317**

318 OpenMP API • Version 3.0 May 2008