OpenMP API User's Guide

Sun™ Studio 10

Sun Microsystems, Inc.
www.sun.com

Part No.819-0501-10
January 2005, Revision A

Submit comments about this document at: http://www.sun.com/hwdocs/feedback

Contents

Before You Begin ix

Typographic Conventions ix

Shell Prompts x

Supported Platforms x

Accessing Sun Studio Software and Man Pages xi

Accessing Compilers and Tools Documentation xiv

Accessing Related Solaris Documentation xvi

Resources for Developers xvi

Contacting Sun Technical Support xvii

Sending Your Comments xvii

1. OpenMP API Summary 1–1

1.1 Where to Find the OpenMP Specifications 1–1

1.2 Special Conventions Used Here 1–2

1.3 Directive Formats 1–2

1.4 Conditional Compilation 1–3

1.5 **`PARALLEL`** - Parallel Region Construct 1–4

1.6 Work-Sharing Constructs 1–5

1.6.1 **`DO`** and **`for`** Constructs 1–5

1.6.2 **`SECTIONS`** Construct 1–7

1.6.3 **SINGLE** Construct 1–7

1.6.4 Fortran **WORKSHARE** Construct 1–8

1.7 Combined Parallel Work-sharing Constructs 1–8

1.7.1 **PARALLEL DO** and **parallel for** Constructs 1–9

1.7.2 **PARALLEL SECTIONS** Construct 1–9

1.7.3 **PARALLEL WORKSHARE** Construct 1–10

1.8 Synchronization Constructs 1–10

1.8.1 **MASTER** Construct 1–11

1.8.2 **CRITICAL** Construct 1–11

1.8.3 **BARRIER** Construct 1–12

1.8.4 **ATOMIC** Construct 1–12

1.8.5 **FLUSH** Construct 1–13

1.8.6 **ORDERED** Construct 1–14

1.9 Data Environment Directives 1–15

1.9.1 **THREADPRIVATE** Directive 1–15

1.10 OpenMP Directive Clauses 1–16

1.10.1 Data Scoping Clauses 1–16

1.10.2 Scheduling Clauses 1–19

1.10.3 **NUM_THREADS** Clause 1–20

1.10.4 Placement of Clauses on Directives 1–20

1.11 OpenMP Runtime Library Routines 1–22

1.11.1 Fortran OpenMP Routines 1–22

1.11.2 C/C++ OpenMP Routines 1–22

1.11.3 Run-time Thread Management Routines 1–23

1.11.4 Routines That Manage Synchronization Locks 1–26

1.11.5 Timing Routines 1–29

2. Nested Parallelism 2–1

2.1 The Execution Model 2–1

2.2 Control of Nested Parallelism 2–2

2.2.1 **OMP_NESTED** 2–2

2.2.2 **SUNW_MP_MAX_POOL_THREADS** 2–3

2.2.3 **SUNW_MP_MAX_NESTED_LEVELS** 2–4

2.3 Using OpenMP Library Functions Within Nested Parallel Regions 2–7

2.4 Some Tips on Using Nested Parallelism 2–10

3. Automatic Scoping in Fortran 3–1

3.1 The Autoscoping Data Scope Clause 3–1

3.1.1 **__AUTO** Clause 3–1

3.1.2 **DEFAULT(__AUTO)** Clause 3–2

3.2 Scoping Rules 3–2

3.2.1 Scoping Rules For Scalar Variables 3–2

3.2.2 Scoping Rules for Arrays 3–3

3.3 General Comments About Autoscoping 3–3

3.4 Checking the Results of Autoscoping 3–4

3.5 Known Limitations of the Current Implementation 3–8

4. Implementation-Defined Behaviors 4–1

5. Compiling for OpenMP 5–1

5.1 Compiler Options To Use 5–1

5.2 Fortran 95 OpenMP Validation 5–3

5.3 OpenMP Environment Variables 5–5

5.4 Processor Binding 5–7

5.5 Stacks and Stack Sizes 5–9

6. Converting to OpenMP 6–1

6.1 Converting Legacy Fortran Directives 6–1

6.1.1 Converting Sun-Style Fortran Directives 6–1

6.1.2 Converting Cray-Style Fortran Directives 6–3

6.2 Converting Legacy C Pragmas 6–4

6.2.1 Issues Between Legacy C Pragmas and OpenMP 6–5

7. Performance Considerations 7–1

7.1 Some General Recommendations 7–1

7.2 False Sharing And How To Avoid It 7–4

7.2.1 What Is *False Sharing*? 7–4

7.2.2 Reducing False Sharing 7–5

7.3 Operating System Tuning Features 7–5

Index Index–1

Tables

TABLE 1-1 Pragmas Where Clauses Can Appear 1–21

TABLE 5-1 OpenMP Environment Variables 5–5

TABLE 5-2 Multiprocessing Environment Variables 5–6

TABLE 6-1 Converting Sun Parallelization Directives to OpenMP 6–1

TABLE 6-2 `DOALL` Qualifier Clauses and OpenMP Equivalent Clauses 6–2

TABLE 6-3 `SCHEDTYPE` Scheduling and OpenMP `schedule` Equivalents 6–2

TABLE 6-4 OpenMP Equivalents for Cray-Style `DOALL` Qualifier Clauses 6–3

TABLE 6-5 Converting Legacy C Parallelization Pragmas to OpenMP 6–4

TABLE 6-6 `taskloop` Optional Clauses and OpenMP Equivalents 6–5

TABLE 6-7 `SCHEDTYPE` Scheduling and OpenMP `schedule` Equivalents 6–5

Before You Begin

The *OpenMP API User's Guide* summarizes the OpenMP Fortran 95, C, and C++ application program interface (API) for building multiprocessing applications. Sun™ Studio compilers support the OpenMP API.

This guide is intended for scientists, engineers, and programmers who have a working knowledge of the Fortran, C, or C++ languages, and the OpenMP parallel programming model. Familiarity with the Solaris™ operating environment or UNIX® in general is also assumed.

Typographic Conventions

TABLE P-1 Typeface Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`
`AaBbCc123`	What you type, when contrasted with on-screen computer output	`% su` `Password:`
AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
AaBbCc123	Command-line placeholder text; replace with a real name or value	To delete a file, type **`rm`** *filename*.

TABLE P-2 Code Conventions

Code Symbol	Meaning	Notation	Code Example
[ ]	Brackets contain arguments that are optional.	`O[`*`n`*`]`	`O4, O`
{ }	Braces contain a set of choices for a required option.	`d{y|n}`	`dy`
|	The "pipe" or "bar" symbol separates arguments, only one of which may be chosen.	`B{dynamic|static}`	`Bstatic`
:	The colon, like the comma, is sometimes used to separate arguments.	`R`*`dir`*`[:`*`dir`*`]`	`R/local/libs:/U/a`
…	The ellipsis indicates omission in a series.	`xinline=`*`f1`*`[,…`*`fn`*`]`	`xinline=alpha,dos`

Shell Prompts

Shell	Prompt
C shell	*machine-name*`%`
C shell superuser	*machine-name*`#`
Bourne shell and Korn shell	`$`
Superuser for Bourne shell and Korn shell	`#`

Supported Platforms

This Sun Studio release supports systems that use the SPARC® and x86 families of processor architectures: UltraSPARC®, SPARC64, AMD64, Pentium, and Xeon EM64T. The supported systems for the version of the Solaris Operating System you are running are available in the hardware compatibility lists at `http://www.sun.com/bigadmin/hcl`. These documents cite any implementation differences between the platform types.

In this document, the term "x86" refers to 64-bit and 32-bit systems manufactured using processors compatible with the AMD64 or Intel Xeon/Pentium product families. For supported systems, see the hardware compatibility lists.

Accessing Sun Studio Software and Man Pages

The compilers and tools and their man pages are not installed into the standard `/usr/bin/` and `/usr/share/man` directories. To access the compilers and tools, you must have your `PATH` environment variable set correctly (see "Accessing the Compilers and Tools" on page xi). To access the man pages, you must have the your `MANPATH` environment variable set correctly (see "Accessing the Man Pages" on page xii.).

For more information about the `PATH` variable, see the `csh`(1), `sh`(1), and `ksh`(1) man pages. For more information about the `MANPATH` variable, see the `man`(1) man page. For more information about setting your `PATH` variable and `MANPATH` variables to access this release, see the installation guide or your system administrator.

Note – The information in this section assumes that your Sun Studio compilers and tools are installed in the `/opt` directory. If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Accessing the Compilers and Tools

Use the steps below to determine whether you need to change your `PATH` variable to access the compilers and tools.

▼ To Determine Whether You Need to Set Your `PATH` Environment Variable

1. **Display the current value of the `PATH` variable by typing the following at a command prompt.**

```
% echo $PATH
```

2. **Review the output to find a string of paths that contain** `/opt/SUNWspro/bin/`**.**

   If you find the path, your `PATH` variable is already set to access the compilers and tools. If you do not find the path, set your `PATH` environment variable by following the instructions in the next procedure.

▼ To Set Your `PATH` Environment Variable to Enable Access to the Compilers and Tools

1. **If you are using the C shell, edit your home** `.cshrc` **file. If you are using the Bourne shell or Korn shell, edit your home** `.profile` **file.**
2. **Add the following to your** `PATH` **environment variable. If you have Forte Developer software, Sun ONE Studio software or another release of Sun Studio software installed, add the following path before the paths to those installations.**

   ```
   /opt/SUNWspro/bin
   ```

Accessing the Man Pages

Use the following steps to determine whether you need to change your `MANPATH` variable to access the man pages.

▼ To Determine Whether You Need to Set Your `MANPATH` Environment Variable

1. **Request the** `dbx` **man page by typing the following at a command prompt.**

   ```
   % man dbx
   ```

2. **Review the output, if any.**

   If the `dbx`(1) man page cannot be found or if the man page displayed is not for the current version of the software installed, follow the instructions in the next procedure for setting your `MANPATH` environment variable.

▼ To Set Your `MANPATH` Environment Variable to Enable Access to the Man Pages

1. **If you are using the C shell, edit your home** `.cshrc` **file. If you are using the Bourne shell or Korn shell, edit your home** `.profile` **file.**
2. **Add the following to your** `MANPATH` **environment variable.**

   ```
   /opt/SUNWspro/man
   ```

Accessing the Integrated Development Environment

The Sun Studio integrated development environment (IDE) provides modules for creating, editing, building, debugging, and analyzing the performance of a C, C++, or Fortran application.

The command to start the IDE is `sunstudio`. For details on this command, see the `sunstudio(1)` man page.

The correct operation of the IDE depends on the IDE being able to find the core platform. The `sunstudio` command looks for the core platform in two locations:

- The command looks first in the default installation directory, `/opt/netbeans/3.5V`.
- If the command does not find the core platform in the default directory, it assumes that the directory that contains the IDE and the directory that contains the core platform are both installed in or mounted to the same location. For example, if the path to the directory that contains the IDE is `/foo/SUNWspro`, the command looks for the core platform in `/foo/netbeans/3.5V`.

If the core platform is not installed or mounted to either of the locations where the `sunstudio` command looks for it, then each user on a client system must set the environment variable `SPRO_NETBEANS_HOME` to the location where the core platform is installed or mounted (*/installation_directory*`/netbeans/3.5V`).

Each user of the IDE also must add */installation_directory*`/SUNWspro/bin` to their `$PATH` in front of the path to any other release of Forte Developer software, Sun ONE Studio software, or Sun Studio software.

The path */installation_directory*`/netbeans/3.5V/bin` should not be added to the user's `$PATH`.

Accessing Compilers and Tools Documentation

You can access the documentation at the following locations:

- The documentation is available from the documentation index that is installed with the software on your local system or network at `file:/opt/SUNWspro/docs/index.html`.

  If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

- Most manuals are available from the `docs.sun.com`[sm] web site. The following titles are available through your installed software only:
  - *Standard C++ Library Class Reference*
  - *Standard C++ Library User's Guide*
  - *Tools.h++ Class Library Reference*
  - *Tools.h++ User's Guide*
- The release notes are available from the `docs.sun.com` web site.
- Online help for all components of the IDE is available through the Help menu, as well as through Help buttons on many windows and dialogs, in the IDE.

The `docs.sun.com` web site (`http://docs.sun.com`) enables you to read, print, and buy Sun Microsystems manuals through the Internet. If you cannot find a manual, see the documentation index that is installed with the software on your local system or network.

Note – Sun is not responsible for the availability of third-party web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with use of or reliance on any such content, goods, or services available on or through any such sites or resources.

Documentation in Accessible Formats

The documentation is provided in accessible formats that are readable by assistive technologies for users with disabilities. You can find accessible versions of documentation as described in the following table. If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Type of Documentation	Format and Location of Accessible Version
Manuals (except third-party manuals)	HTML at `http://docs.sun.com`
Third-party manuals: • *Standard C++ Library Class Reference* • *Standard C++ Library User's Guide* • *Tools.h++ Class Library Reference* • *Tools.h++ User's Guide*	HTML in the installed software through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Readmes and man pages	HTML in the installed software through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Online help	HTML available through the Help menu in the IDE
Release notes	HTML at `http://docs.sun.com`

Related Compilers and Tools Documentation

The following table describes related documentation that is available at `file:/opt/SUNWspro/docs/index.html` and `http://docs.sun.com`. If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system

Document Title	Description
Fortran Programming Guide	Describes how to write effective Fortran code on Solaris environments; input/output, libraries, performance, debugging, and parallel processing.
Fortran Library Reference	Details the Fortran library and intrinsic routines
Fortran User's Guide	Describes the compile-time environment and command-line options for the `f95` compiler. Also includes guidelines for migrating legacy `f77` programs to `f95`.

Document Title	Description
C User's Guide	Describes the compile-time environment and command-line options for the `cc` compiler.
C++ User's Guide	Describes the compile-time environment and command-line options for the `CC` compiler.
Numerical Computation Guide	Describes issues regarding the numerical accuracy of floating-point computations.

Accessing Related Solaris Documentation

The following table describes related documentation that is available through the `docs.sun.com` web site.

Document Collection	Document Title	Description
Solaris Reference Manual Collection	See the titles of man page sections.	Provides information about the Solaris operating environment.
Solaris Software Developer Collection	*Linker and Libraries Guide*	Describes the operations of the Solaris link-editor and runtime linker.
Solaris Software Developer Collection	*Multithreaded Programming Guide*	Covers the POSIX and Solaris threads APIs, programming with synchronization objects, compiling multithreaded programs, and finding tools for multithreaded programs.

Resources for Developers

Visit `http://developers.sun.com/prodtech/cc` to find these frequently updated resources:

- Articles on programming techniques and best practices
- A knowledge base of short programming tips

- Documentation of compilers and tools components, as well as corrections to the documentation that is installed with your software
- Information on support levels
- User forums
- Downloadable code samples
- New technology previews

You can find additional resources for developers at http://developers.sun.com.

Contacting Sun Technical Support

If you have technical questions about this product that are not answered in this document, go to:

http://www.sun.com/service/contacting

Sending Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. Submit your comments to Sun at this URL:

http://www.sun.com/hwdocs/feedback

Please include the part number (819-0501-10) of your document in the subject line of your email.

CHAPTER 1

OpenMP API Summary

The OpenMP™ Application Program Interface is a portable, parallel programming model for shared memory multiprocessor architectures, developed in collaboration with a number of computer vendors. The specifications were created and are published by the OpenMP Architecture Review Board. For more information on OpenMP, including tutorials and other resources, see their web site at: http://www.openmp.org/.

The OpenMP API is the recommended parallel programming model for all Sun Studio compilers on Solaris™ OS platforms. See Chapter 6 for guidelines on converting legacy Fortran and C parallelization directives to OpenMP.

This chapter summarizes the directives, run-time library routines, and environment variables comprising the OpenMP Version 2.0 Application Program Interfaces, as implemented by the Sun Studio Fortran 95, C and C++ compilers.

1.1 Where to Find the OpenMP Specifications

The material presented in this chapter *is only a summary* with many details left out intentionally for the sake of brevity. In all cases, refer to the OpenMP specification documents for complete details.

The Fortran and C/C++ OpenMP 2.0 specifications can be found on the official OpenMP website, http://www.openmp.org/.

1.2 Special Conventions Used Here

In the tables and examples that follow, Fortran directives and source code are shown in upper case, but are case-insensitive.

The term *structured-block* refers to a block of Fortran or C/C++ statements having no transfers into or out of the block.

Constructs within square brackets, [...], are optional.

Throughout this manual, "Fortran" refers to the Fortran 95 language and compiler, **`f95`**.

The terms "directive" and "pragma" are used interchangeably in this manual.

1.3 Directive Formats

Only one *directive-name* can be specified on a directive line, and applies to the succeeding program statement.

Fortran:

Fortran fixed format accepts three directive "sentinels", free format accepts only one. In the Fortran examples that follow, free format will be used.

C/C++:

C and C++ use the standard preprocessing directive starting with **`#pragma omp`**.

OpenMP 2.0 Fortran

Fixed Format:

`C$OMP` *directive-name optional_clauses...*

`!$OMP` *directive-name optional_clauses...*

`*$OMP` *directive-name optional_clauses...*

The sentinel must start in column one; continuation lines must have a non-blank or non-zero character in column 6.

Comments may appear after column 6 on the directive line, initiated by an exclamation point (**`!`**). The rest of the line after the **`!`** is ignored.

OpenMP 2.0 Fortran

Free Format:

`!$OMP` *directive-name optional_clauses...*

May appear anywhere on a line, preceded only by whitespace; an ampersand (**&**) at the end of the line identifies a continued line.

Comments may appear on the directive line, initiated by an exclamation point (!). The rest of the line is ignored.

OpenMP 2.0 C/C++

`#pragma omp` *directive-name optional_clauses...*

Each pragma must end with a new-line character, and follows the conventions of standard C and C++ for compiler pragmas.

Pragmas are case sensitive. The order in which clauses appear is not significant. White space can appear after and before the **#** and between words.

The directive applies to the succeeding statement, which must be a structured block.

1.4 Conditional Compilation

The OpenMP API defines the preprocessor symbol **_OPENMP** to be used for conditional compilation. In addition, OpenMP Fortran API accepts a conditional compilation sentinel.

OpenMP 2.0 Fortran

Fixed Format:

`!$` *fortran_95_statement*

`C$` *fortran_95_statement*

`*$` *fortran_95_statement*

`c$` *fortran_95_statement*

The sentinel must start in column 1 and have no intervening blanks. With OpenMP compilation enabled, the sentinel is replaced by two blanks. The rest of the line must conform to standard Fortran fixed format conventions. Example:

```
C23456789
!$ 10 iam = OMP_GET_THREAD_NUM() +
!$   1       index
```

OpenMP 2.0 Fortran

Free Format:

`!$` *fortran_95_statement*

This sentinel can appear in any column, preceded only by white space, and must appear as a single word. Fortran free format conventions apply to the rest of the line. Example:

```
C23456789
 !$ iam = OMP_GET_THREAD_NUM() +     &
 !$&         index
```

Fortran Preprocessor:

Compiling with OpenMP enabled defines the preprocessor symbol **_OPENMP**.

```
#ifdef _OPENMP
     iam = OMP_GET_THREAD_NUM()+index
#endif
```

OpenMP 2.0 C/C++

C/C++ Preprocessor:

Compiling with OpenMP enabled defines the macro **_OPENMP**.

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

1.5 **PARALLEL** - Parallel Region Construct

The **PARALLEL** directive defines a parallel region, which is a region of the program that is to be executed by multiple threads in parallel.

OpenMP 2.0 Fortran

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

OpenMP 2.0 C/C++

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

There are many special conditions and restrictions. Programmers are urged to refer to the appropriate OpenMP specification document for the details.

TABLE 1-1 identifies the clauses that can appear with this construct.

1.6 Work-Sharing Constructs

Work-sharing constructs divide the execution of the enclosed code region among the members of the team of threads that encounter it. Work sharing constructs must be enclosed within a parallel region for the construct to execute in parallel.

There are many special conditions and restrictions on these directives and the code they apply to. Programmers are urged to refer to the appropriate OpenMP specification document for the details.

1.6.1 **DO** and **for** Constructs

Specifies that the iterations of the **DO** or **for** loop that follows should be executed in parallel.

OpenMP 2.0 Fortran

```
!$OMP DO [clause[[,] clause]...]
  do_loop
[!$OMP END DO [NOWAIT]]
```

The **DO** directive specifies that the iterations of the **DO** loop that immediately follows should be executed in parallel. The iterations of the loop are distributed across threads already existing in the team of threads executing the parallel region that binds the loop. This directive must appear within a parallel region to be effective.

OpenMP 2.0 C/C++

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

The **`for`** pragma specifies that the iterations of the *for-loop* that immediately follows should be executed in parallel. The iterations of the loop are distributed across threads already existing in the team of threads executing the parallel region that binds the loop. This pragma must appear within a parallel region to be effective. The **`for`** pragma places restrictions on the structure of the corresponding **`for`** loop, and it must have *canonical shape*:

`for` (*initexpr*; *var logicop b*; *increxpr*)

where:

- *initexpr* is one of the following:
  var = *lb*
  integer_type var = *lb*
- *increxpr* is one of the following expression forms:
  ++*var*
  var++
  --*var*
  var--
  var += *incr*
  var -= *incr*
  var = *var* + *incr*
  var = *incr* + *var*
  var = *var* – *incr*
- *var* is a signed integer variable, made implicitly private for the range of the **`for`**. *var* must not be modified within the body of the **`for`** statement. Its value is indeterminate after the loop, unless specified **`lastprivate`**.
- *logicop* is one of the following logical operators:
  < <= > >=
- *lb*, *b*, and *incr* are loop invariant integer expressions.

There are further restrictions on the use of < or <= and > or >= as *logicalop* in the **`for`** statement. See the OpenMP C/C++ specifications for details.

TABLE 1-1 identifies the clauses that can appear with this construct.

1.6.2 **SECTIONS** Construct

The **SECTIONS** construct encloses a set of structured blocks of code to be divided among threads in the team. Each block is executed once by a thread in the team.

Each section is preceded by a **SECTION** directive, which is optional for the first section.

OpenMP 2.0 Fortran

```
!$OMP SECTIONS [clause[[,] clause]...]
[!$OMP SECTION]
  structured-block
[!$OMP SECTION
 structured-block ]
...
!$OMP END SECTIONS [NOWAIT]
```

OpenMP 2.0 C/C++

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

TABLE 1-1 identifies the clauses that can appear with this construct.

1.6.3 **SINGLE** Construct

The structured block enclosed by **SINGLE** is executed by only one thread in the team. Threads in the team that are not executing the **SINGLE** block wait at the end of the block unless **NOWAIT** is specified.

OpenMP 2.0 Fortran

```
!$OMP SINGLE [clause[[,] clause]...]
  structured-block
!$OMP END SINGLE [end-modifier]
```

OpenMP 2.0 C/C++

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

TABLE 1-1 identifies the clauses that can appear with this construct.

1.6.4 Fortran **WORKSHARE** Construct

The **WORKSHARE** construct divides the work of executing the enclosed code 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.

OpenMP 2.0 Fortran

```
!$OMP WORKSHARE
  structured-block
!$OMP END WORKSHARE [NOWAIT]
```

There is no C/C++ equivalent to the Fortran `WORKSHARE` construct.

1.7 Combined Parallel Work-sharing Constructs

The combined parallel work-sharing constructs are shortcuts for specifying a parallel region that contains one work-sharing construct.

There are many special conditions and restrictions on these directives and the code they apply to. Refer to the appropriate OpenMP specification document for the complete details. The description that follows is intended only as a summary and is not complete.

TABLE 1-1 identifies the clauses that can appear with these constructs.

1.7.1 **PARALLEL DO** and **parallel for** Constructs

Shortcut for specifying a parallel region that contains a single **DO** or **for** loop. Equivalent to a **PARALLEL** directive followed immediately by a **DO** or **for** directive. *clause* can be any of the clauses accepted by the **PARALLEL** and **DO/for** directives, except the **NOWAIT** modifier.

OpenMP 2.0 Fortran

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

OpenMP 2.0 C/C++

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

1.7.2 **PARALLEL SECTIONS** Construct

Shortcut for specifying a parallel region that contains a single **SECTIONS** directive. Equivalent to a **PARALLEL** directive followed by a **SECTIONS** directive. *clause* can be any of the clauses accepted by the **PARALLEL** and **SECTIONS** directives, except the **NOWAIT** modifier.

OpenMP 2.0 Fortran

```
!$OMP PARALLEL SECTIONS [clause[[,] clause]...]
[!$OMP SECTION]
  structured-block
[!$OMP SECTION
  structured-block ]
...
!$OMP END PARALLEL SECTIONS
```

OpenMP 2.0 C/C++

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

1.7.3 **PARALLEL WORKSHARE** Construct

The Fortran **PARALLEL WORKSHARE** construct provides a shortcut for specifying a parallel region that contains a single **WORKSHARE** directive. *clause* can be one of the clauses accepted by the **PARALLEL** directive.

OpenMP 2.0 Fortran

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

There is no C/C++ equivalent.

1.8 Synchronization Constructs

The following constructs specify thread synchronization. There are many special conditions and restrictions regarding these constructs that are too numerous to summarize here. Programmers are urged to refer to the appropriate OpenMP specification document for the complete details.

1.8.1 **MASTER** Construct

Only the master thread of the team executes the block enclosed by this directive. The other threads skip this block and continue. There is no implied barrier on entry to or exit from the master construct.

OpenMP 2.0 Fortran

```
!$OMP MASTER
  structured-block
!$OMP END MASTER
```

OpenMP 2.0 C/C++

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

1.8.2 **CRITICAL** Construct

Restrict access to the structured block to only one thread at a time. The optional *name* argument identifies the critical region. All unnamed **CRITICAL** directives map to the same name. Critical section names are global entities of the program and must be unique. For Fortran, if *name* appears on the **CRITICAL** directive, it must also appear on the **END CRITICAL** directive. For C/C++, the identifier used to name a critical region has external linkage and is in a name space which is separate from the name spaces used by labels, tags, members, and ordinary identifiers.

OpenMP 2.0 Fortran

```
!$OMP CRITICAL [(name)]
  structured-block
!$OMP END CRITICAL [(name)]
```

OpenMP 2.0 C/C++

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

1.8.3 **BARRIER** Construct

Synchronizes all the threads in a team. Each thread waits until all the others in the team have reached this point.

OpenMP 2.0 Fortran

```
!$OMP BARRIER
```

OpenMP 2.0 C/C++

```
#pragma omp barrier
```

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

Note that because the **barrier** pragma does not have a C/C++ statement as part of its syntax, there are restrictions on its placement within a program. See the C/C++ OpenMP specifications for details.

1.8.4 **ATOMIC** Construct

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

OpenMP 2.0 Fortran

```
!$OMP ATOMIC
 expression-statement
```

The directive applies only to the *expression-statement* immediately following the directive, which must be in one of these forms:

x = x operator expression

x = expression operator x

x = intrinsic(x, expr-list)

x = intrinsic(expr-list, x)

where:

- *x* is a scalar of intrinsic type
- *expression* is a scalar expression that does not reference *x*
- *expr-list* is a non-empty, comma-separated list of scalar expressions that do not reference x (*see the OpenMP 2.0 Fortran specifications for details*)
- *intrinsic* is one of **MAX, MIN, IAND, IOR,** or **IEOR**.
- *operator* is one of **+ - * / .AND. .OR. .EQV. .NEQV.**

OpenMP 2.0 C/C++

```
#pragma omp atomic
  expression-statement
```

The directive applies only to the *expression-statement* immediately following the directive, which must be in one of these forms:

x binop = expr

x++

++*x*

x--

--*x*

where:

- *x* in an lvalue expression with scalar type.
- *expr* is an expression with scalar type that does not reference *x*.
- *binop* is not an overloaded operator and one of: **+**, *****, **-**, **/**, **&**, **^**, **|**, **<<**, or **>>**.

This implementation replaces all ATOMIC directives by enclosing the expression-statement in a critical section.

1.8.5 **FLUSH** Construct

Thread-visible Fortran variables or C objects are written back to memory at the point at which this directive appears. The **FLUSH** directive only provides consistency between operations within the executing thread and global memory. The optional *variable-list* consists of a comma-separated list of variables or objects that need to be flushed. A **FLUSH** directive without a *variable-list* synchronizes all thread-visible shared variables or objects.

OpenMP 2.0 Fortran

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

OpenMP 2.0 C/C++

```
#pragma omp flush [(variable-list)]
```

Note that because the **flush** pragma does not have a C/C++ statement as part of its syntax, there are restrictions on its placement within a program. See the C/C++ OpenMP specifications for details.

1.8.6 **ORDERED** Construct

The enclosed block is executed in the order that iterations would be executed in a sequential execution of the loop.

OpenMP 2.0 Fortran

```
!$OMP ORDERED
  structured-block
!$OMP END ORDERED
```

The enclosed block is executed in the order that iterations would be executed in a sequential execution of the loop. It can appear only in the dynamic extent of a **DO** or **PARALLEL DO** directive. The **ORDERED** clause must be specified on the closest **DO** directive enclosing the block.
A loop to which a **DO** directive applies must not execute the same **ordered** directive more than once per iteration, and it must not execute more than one **ordered** directive.

OpenMP 2.0 C/C++

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

The enclosed block is executed in the order that iterations would be executed in a sequential execution of the loop. It can appear only in the dynamic extent of a **for** or **parallel for** directive with the **ordered** clause specified.
A loop with a **for** construct must not execute the same **ordered** directive more than once per iteration, and it must not execute more than one **ordered** directive.

1.9 Data Environment Directives

The following directives control the data environment during execution of parallel constructs.

1.9.1 **THREADPRIVATE** Directive

Makes the *list* of objects (Fortran common blocks and named variables, C and C++ named variables) private to a thread but global within the thread.

See the OpenMP specifications for the complete details and restrictions.

OpenMP 2.0 Fortran

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

Common block names must appear between slashes. To make a common block **THREADPRIVATE**, this directive must appear after every **COMMON** declaration of that block.

OpenMP 2.0 C/C++

```
#pragma omp threadprivate (list)
```

Each variable in *list* at file, namespace, or block scope must refer to a variable declaration at file, namespace, or block scope that lexically preceeds the pragma.

1.10 OpenMP Directive Clauses

This section summarizes the data scoping and scheduling clauses that can appear on OpenMP directives.

1.10.1 Data Scoping Clauses

Several directives accept clauses that allow a user to control the scope attributes of variables within the extent of the construct. If no data scope clause is specified for a directive, the default scope for variables affected by the directive is **SHARED**.

Fortran: *list* is a comma-separated list of named variables or common blocks that are accessible in the scoping unit. Common block names must appear within slashes (for example, **/ABLOCK/**).

There are important restrictions on the use of these scoping clauses. Refer to the appropriate sections of the OpenMP specifications for complete details.

TABLE 1-1 identifies the directives on which these clauses can appear.

1.10.1.1 **PRIVATE** Clause

private(*list***)**

Declares the variables in the optional comma-separated *list* to be private to each thread in a team.

1.10.1.2 **SHARED** Clause

shared(*list***)**

All the threads in the team share the variables that appear in *list*, and access the same storage area.

1.10.1.3 **DEFAULT** Clause

Fortran

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

C/C++

default(shared | none)

Specify scoping attributes for all variables within a parallel region. **THREADPRIVATE** variables are not affected by this clause. If not specified, **DEFAULT(SHARED)** is assumed. A variable's default data-sharing attribute can be overridden by using the **private**, **firstprivate**, **lastprivate**, **reduction**, and **shared** clauses.

1.10.1.4 **FIRSTPRIVATE** Clause

firstprivate(*list***)**

The variables in list are **PRIVATE**. In addition, private copies of the variables are initialized from the original object existing before the construct.

1.10.1.5 **LASTPRIVATE** Clause

lastprivate(*list***)**

The variables in the list are **PRIVATE**. In addition, when the **LASTPRIVATE** clause appears on a **DO** or **for** directive, the thread that executes the sequentially last iteration updates the original object. On a **SECTIONS** directive, the thread that executes the lexically last **SECTION** updates the original object.

1.10.1.6 **COPYIN** Clause

Fortran

COPYIN(*list***)**

The **COPYIN** clause applies only to variables, common blocks, and variables in common blocks that are declared as **THREADPRIVATE**. In a parallel region, **COPYIN** specifies that the data in the master thread of the team be copied to the threadprivate copies at the beginning of the parallel region.

C/C++

copyin(*list***)**

The **COPYIN** clause applies only to variables that are declared as **THREADPRIVATE**. In a parallel region, **COPYIN** specifies that the data in the master thread of the team be copied to the threadprivate copies at the beginning of the parallel region.

1.10.1.7 **COPYPRIVATE** Clause

Fortran

COPYPRIVATE(*list***)**

Uses a private variable to broadcast a value, or a pointer to a shared object, from one member of a team to the other members. **COPYPRIVATE** clause can only appear on the **END SINGLE** directive. The broadcast occurs after the execution of the structured block associated with the **single** construct, and before any threads in the team have left the barrier at the end of the construct. The variables in *list* must not appear in a **PRIVATE** or **FIRSTPRIVATE** clause of the **SINGLE** construct specifying **COPYPRIVATE**.

C/C++

copyprivate(*list***)**

Uses a private variable to broadcast a value from one member of a team to the other members. The **copyprivate** clause can only appear on the **single** directive. The broadcast occurs after the execution of the structured block associated with the **single** construct, and before any threads in the team have left the barrier at the end of the construct. The variables in *list* must not appear in a **private** or **firstprivate** clause for the same **single** directive.

1.10.1.8 **REDUCTION** Clause

Fortran

REDUCTION(*operator* | *intrinsic***:***list***)**

operator is one of: **+**, *****, **-**, **.AND.**, **.OR.**, **.EQV.**, **.NEQV.**

intrinsic is one of: **MAX**, **MIN**, **IAND**, **IOR**, **IEOR**

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

C/C++

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

operator is one of: **+**, *****, **-**, **&**, **^**, **|**, **&&**, **||**

The **REDUCTION** clause is intended to be used on a region in which the reduction variable is used only in reduction statements. The variables in *list* must be **SHARED** in the enclosing context. A private copy of each variable is created for each thread as if it were **PRIVATE**. At the end of the reduction, the shared variable is updated by combining the original value with the final value of each of the private copies.

See the appropriate sections of the OpenMP specifications for complete details and restrictions on **REDUCTION** clauses and constructs.

1.10.2 Scheduling Clauses

The **SCHEDULE** clause specifies how iterations in a Fortran **DO** loop or C/C++ **for** loop are divided among the threads in a team. TABLE 1-1 shows which directives allow the **SCHEDULE** clause.

There are important restrictions on the use of these scheduling clauses. Refer to section 2.3.1 in the Fortran specification, and section 2.4.1 in the C/C++ specification for complete details.

schedule(*type* [**,***chunk*]**)**

Specifies how iterations of the **DO** or **for** loop are divided among the threads of the team. *type* can be one of **STATIC**, **DYNAMIC**, **GUIDED**, or **RUNTIME**. In the absence of a **SCHEDULE** clause, Sun Studio compilers use **STATIC** scheduling. *chunk* must be an integer expression.

1.10.2.1 **STATIC** Scheduling

schedule(static[**,***chunk*]**)**

Iterations are divided into pieces of a size specified by *chunk*. The pieces are statically assigned to threads in the team in a round-robin fashion in the order of the thread number. If not specified, *chunk* is chosen so that the iterations divide into contiguous chunks nearly equal in size with one chunk assigned to each thread.

1.10.2.2 **DYNAMIC** Scheduling

schedule(dynamic[**,***chunk*]**)**

Iterations are divided into pieces of a size specified by *chunk*, and assigned to a waiting thread. As each thread finishes its piece of the iteration space, it dynamically obtains the next set of iterations. When no *chunk* is specified, it defaults to 1.

1.10.2.3 **GUIDED** Scheduling

schedule(guided[**,***chunk*]**)**

With **GUIDED**, the chunk size is reduced in an exponentially decreasing manner with each dispatched piece of the iterations. *chunk* specifies the minimum number of iterations to dispatch each time. (The size of the chunks is determined by a formula that is implementation dependent; see "GUIDED: Determination of Chunk Sizes" on page 4-2.). When no *chunk* is specified, it defaults to 2.0.

1.10.2.4 **RUNTIME** Scheduling

schedule(runtime)

Scheduling is deferred until runtime. Schedule *type* and *chunk* size will be determined from the value of the **OMP_SCHEDULE** environment variable. (Default is **SCHEDULE(STATIC)**).

1.10.3 **NUM_THREADS** Clause

The OpenMP API provides a **NUM_THREADS** clause on the **PARALLEL**, **PARALLEL SECTIONS**, **PARALLEL DO**, **PARALLEL for,** and **PARALLEL WORKSHARE** directives.

num_threads(*scalar_integer_expression***)**

Specifies the number of threads in the team created when a thread enters a parallel region. *scalar_integer_expression* is the number of threads requested, and supersedes the number of threads defined by a prior call to the **OMP_SET_NUM_THREADS** library function, or the value of the **OMP_NUM_THREADS** environment variable. If dynamic thread management is enabled, the request is the *maximum* number of threads to use.

Note that **num_threads** does not apply to subsequent regions.

1.10.4 Placement of Clauses on Directives

TABLE 1-1 shows the clauses that can appear on these directives and pragmas:

- **PARALLEL**
- **DO**
- **for**
- **SECTIONS**
- **SINGLE**
- **PARALLEL DO**
- **parallel for**
- **PARALLEL SECTIONS**
- **PARALLEL WORKSHARE**

TABLE 1-1 Pragmas Where Clauses Can Appear

Clause/Pragma	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS	PARALLEL WORKSHARE[3]
IF	•				•	•	•
PRIVATE	•	•	•	•	•	•	•
SHARED	•				•	•	•
FIRSTPRIVATE	•	•	•	•	•	•	•
LASTPRIVATE		•	•		•	•	
DEFAULT	•				•	•	•
REDUCTION	•	•	•		•	•	•
COPYIN	•				•	•	•
COPYPRIVATE				•[1]			
ORDERED		•			•		
SCHEDULE		•			•		
NOWAIT		•[2]	•[2]	•[2]			
NUM_THREADS	•				•	•	•

1. Fortran only: **COPYPRIVATE** can appear on the **END SINGLE** directive.
2. For Fortran, a **NOWAIT** modifier can only appear on the **END DO**, **END SECTIONS**, **END SINGLE**, or **END WORKSHARE** directives.
3. Only Fortran supports **WORKSHARE** and **PARALLEL WORKSHARE**.

1.11 OpenMP Runtime Library Routines

OpenMP provides a set of callable library routines to control and query the parallel execution environment, a set of general purpose lock routines, and two portable timer routines. Full details appear in the Fortran and C/C++ OpenMP specifications.

1.11.1 Fortran OpenMP Routines

The Fortran run-time library routines are external procedures. In the following summary, *int_expr* is a scalar integer expression, and *logical_expr* is a scalar logical expression.

The **OMP_** functions returning **INTEGER(4)** and **LOGICAL(4)** are not intrinsic and must be declared properly, otherwise the compiler will assume **REAL**. Interface declarations for the OpenMP Fortran runtime library routines summarized below are provided by the Fortran include file **omp_lib.h** and a Fortran **MODULE omp_lib**, as described in the Fortran OpenMP specifications.

Supply an **INCLUDE 'omp_lib.h'** statement or **#include "omp_lib.h"** preprocessor directive, or a **USE omp_lib** statement in every program unit that references these library routines.

Compiling with **-Xlist** will report any type mismatches.

The integer parameter **omp_lock_kind** defines the **KIND** type parameters used for simple lock variables in the **OMP_*_LOCK** routines.

The integer parameter **omp_nest_lock_kind** defines the **KIND** type parameters used for the nestable lock variables in the **OMP_*_NEST_LOCK** routines.

The integer parameter **openmp_version** is defined as a preprocessor macro **_OPENMP** having the form **YYYYMM** where **YYYY** and **MM** are the year and month designations of the version of the OpenMP Fortran API.

1.11.2 C/C++ OpenMP Routines

The C/C++ run-time library functions are external functions.

The header **<omp.h>** declares two types, several functions that can be used to control and query the parallel execution environment, and lock functions that can be used to synchronize access to data.

The type **omp_lock_t** is an object type capable of representing that a lock is available, or that a thread owns a lock. These locks are referred to as simple locks.

The type **omp_nest_lock_t** is an object type capable of representing that a lock is available, or that a thread owns a lock. These locks are referred to as nestable locks.

1.11.3 Run-time Thread Management Routines

For details, refer to the appropriate OpenMP specifications.

1.11.3.1 **OMP_SET_NUM_THREADS** Routine

Sets the number of threads to use for subsequent parallel regions not specified with a **num_threads()** clause. This call affects only the subsequent parallel regions encountered by the calling thread at the same or inner nesting level.

Fortran

```
SUBROUTINE OMP_SET_NUM_THREADS(int_expr)
```

C/C++

```
#include <omp.h>
void omp_set_num_threads(int num_threads);
```

1.11.3.2 **OMP_GET_NUM_THREADS** Routine

Returns the number of threads currently in the team executing the parallel region from which it is called.

Fortran

```
INTEGER(4) FUNCTION OMP_GET_NUM_THREADS()
```

C/C++

```
#include <omp.h>
int omp_get_num_threads(void);
```

1.11.3.3 **OMP_GET_MAX_THREADS** Routine

Returns maximum number of threads that would be used to form a team if an active paralel region specified without a **num_threads()** clause were to be encountered at this point in the program.

Fortran

```
INTEGER(4) FUNCTION OMP_GET_MAX_THREADS()
```

C/C++

```
#include <omp.h>
int omp_get_max_threads(void);
```

1.11.3.4 **OMP_GET_THREAD_NUM** Routine

Returns the thread number, within its team, of the thread executing the call to this function. This number lies between 0 and `OMP_GET_NUM_THREADS()-1`, with 0 being the master thread.

Fortran

```
INTEGER(4) FUNCTION OMP_GET_THREAD_NUM()
```

C/C++

```
#include <omp.h>
int omp_get_thread_num(void);
```

1.11.3.5 **OMP_GET_NUM_PROCS** Routine

Return the number of processors available to the program.

Fortran

```
INTEGER(4) FUNCTION OMP_GET_NUM_PROCS()
```

C/C++

```
#include <omp.h>
int omp_get_num_procs(void);
```

1.11.3.6 **OMP_IN_PARALLEL** Routine

Determine whether or not thread is executing within the dynamic extent of a parallel region.

Fortran

```
LOGICAL(4) FUNCTION OMP_IN_PARALLEL()
```

Returns `.TRUE.` if called within the dynamic extent of an active parallel region, `.FALSE.` otherwise.

C/C++

```
#include <omp.h>
int omp_in_parallel(void);
```

Returns nonzero if called within the dynamic extent of an active parallel region, zero otherwise.

An active parallel region is a parallel region where the **IF** clause evaluates to **TRUE**.

1.11.3.7 **OMP_SET_DYNAMIC** Routine

Enables or disables dynamic adjustment of the number of available threads. (Dynamic adjustment is enabled by default.) This call affects only the subsequent parallel regions encountered by the calling thread at the same or inner nesting level.

Fortran

```
SUBROUTINE OMP_SET_DYNAMIC(logical_expr)
```

Dynamic adjustment is enabled when *logical_expr* evaluates to `.TRUE.`, and is disabled otherwise.

C/C++

```
#include <omp.h>
void omp_set_dynamic(int dynamic);
```

If *dynamic* evaluates to nonzero, dynamic adjustment is enabled; otherwise it is disabled.

1.11.3.8 **OMP_GET_DYNAMIC** Routine

Determine whether or not dynamic thread adjustment is enabled at this point in the program.

Fortran

```
LOGICAL(4) FUNCTION OMP_GET_DYNAMIC()
```

Returns `.TRUE.` if dynamic thread adjustment is enabled, `.FALSE.` otherwise.

C/C++

```
#include <omp.h>
int omp_get_dynamic(void);
```

Returns nonzero if dynamic thread adjustment is enabled, zero otherwise.

1.11.3.9 **OMP_SET_NESTED** Routine

Enables or disables nested parallelism.This call affects only the subsequent parallel regions encountered by the calling thread at the same or inner nesting level.

Fortran

```
SUBROUTINE OMP_SET_NESTED(logical_expr)
```

Nested parallelism is enabled if *logical_expr* evaluates to **.TRUE.**, and is disabled otherwise.

C/C++

```
#include <omp.h>
void omp_set_nested(int nested);
```

Nested parallelism is enabled if *nested* evaluates to non-zero, and is disabled otherwise.

Nested parallelism is disabled by default. See Chapter 2 for information on nested parallelism.

1.11.3.10 **OMP_GET_NESTED** Routine

Determine whether or not nested parallelism is enabled at this point in the program.

Fortran

```
LOGICAL(4) FUNCTION OMP_GET_NESTED()
```

Returns **.TRUE.** if nested parallelism is enabled, **.FALSE.** otherwise.

C/C++

```
#include <omp.h>
int omp_get_nested(void);
```

Returns nonzero if nested parallelism is enabled, zero otherwise.

See Chapter 2 for information on nested parallelism.

1.11.4 Routines That Manage Synchronization Locks

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

Fortran:

The lock variable *var* must be accessed only through these routines. Use the parameters **OMP_LOCK_KIND** and **OMP_NEST_LOCK_KIND** (defined in **omp_lib.h INCLUDE** file and the **omp_lib MODULE**) for this purpose. For example,

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

C/C++:

Simple lock variables must have type **omp_lock_t** and must be accessed only through these functions. All simple lock functions require an argument that points to **omp_lock_t** type.

Nested lock variables must have type **omp_nest_lock_t**, and similarly all nested lock functions require an argument that points to **omp_nest_lock_t** type.

1.11.4.1 OMP_INIT_LOCK and OMP_INIT_NEST_LOCK Routines

Initialize a lock variable for subsequent calls.

Fortran

```
SUBROUTINE OMP_INIT_LOCK(var)
SUBROUTINE OMP_INIT_NEST_LOCK(nvar)
```

C/C++

```
#include <omp.h>
void omp_init_lock(omp_lock_t *lock);
void omp_init_nest_lock(omp_nest_lock_t *lock);
```

1.11.4.2 OMP_DESTROY_LOCK and OMP_DESTROY_NEST_LOCK Routines

Removes a lock variable.

Fortran

```
SUBROUTINE OMP_DESTROY_LOCK(var)
SUBROUTINE OMP_DESTROY_NEST_LOCK(nvar)
```

C/C++

```
#include <omp.h>
void omp_destroy_lock(omp_lock_t *lock);
void omp_destroy_nest_lock(omp_nest_lock_t *lock);
```

1.11.4.3 **OMP_SET_LOCK** and **OMP_SET_NEST_LOCK** Routines

Forces the executing thread to wait until the specified lock is available. The thread is granted ownership of the lock when it is available.

Fortran

```
SUBROUTINE OMP_SET_LOCK(var)
SUBROUTINE OMP_SET_NEST_LOCK(nvar)
```

C/C++

```
#include <omp.h>
void omp_set_lock(omp_lock_t *lock);
void omp_set_nest_lock(omp_nest_lock_t *lock);
```

1.11.4.4 **OMP_UNSET_LOCK** and **OMP_UNSET_NEST_LOCK** Routines

Releases the executing thread from ownership of the lock. Behavior is undefined if the thread does not own that lock.

Fortran

```
SUBROUTINE OMP_UNSET_LOCK(var)
SUBROUTINE OMP_UNSET_NEST_LOCK(nvar)
```

C/C++

```
#include <omp.h>
void omp_unset_lock(omp_lock_t *lock);
void omp_unset_nest_lock(omp_nest_lock_t *lock);
```

1.11.4.5 **OMP_TEST_LOCK** and **OMP_TEST_NEST_LOCK** Routines

OMP_TEST_LOCK attempts to set the lock associated with lock variable. Call does not block execution of the thread.

OMP_TEST_NEST_LOCK returns the new nesting count if the lock was set successfully, otherwise it returns 0. Call does not block execution of the thread.

Fortran

```
LOGICAL(4) FUNCTION OMP_TEST_LOCK(var)
```

Returns **.TRUE.** if the lock was set, **.FALSE.** otherwise.

```
INTEGER(4) FUNCTION OMP_TEST_NEST_LOCK(nvar)
```

Returns nesting count if lock was set successfully, zero otherwise.

C/C++

```
#include <omp.h>
int omp_test_lock(omp_lock_t *lock);
```

Returns a nonzero value if lock was set successfully, zero otherwise.

```
int omp_test_nest_lock(omp_nest_lock_t *lock);
```

Returns lock nest count if lock was set successfully, zero otherwise.

1.11.5 Timing Routines

Two functions support a portable wall clock timer.

1.11.5.1 **OMP_GET_WTIME** Routine

Returns the elapsed wall clock time in seconds "since some arbitrary time in the past".

Fortran

```
REAL(8) FUNCTION OMP_GET_WTIME()
```

C/C++

```
#include <omp.h>
double omp_get_wtime(void);
```

1.11.5.2 **OMP_GET_WTICK** Routine

Returns the number of seconds between successive clock ticks.

Fortran

```
REAL(8) FUNCTION OMP_GET_WTICK()
```

C/C++

```
#include <omp.h>
double omp_get_wtick(void);
```

CHAPTER 2

Nested Parallelism

This chapter discusses the features of OpenMP nested parallelism.

2.1 The Execution Model

OpenMP uses a fork-join model of parallel execution. When a thread encounters a parallel construct, the thread creates a team composed of itself and some additional (possibly zero) number of threads. The encountering thread becomes the master of the new team. The other threads of the team are called slave threads of the team. All team members execute the code inside the parallel construct. When a thread finishes its work within the parallel construct, it waits at the implicit barrier at the end of the parallel construct. When all team members have arrived at the barrier, the threads can leave the barrier. The master thread continues execution of user code beyond the end of the parallel construct, while the slave threads wait to be summoned to join other teams.

OpenMP parallel regions can be nested inside each other. If nested parallelism is disabled, then the new team created by a thread encountering a parallel construct inside a parallel region consists only of the encountering thread. If nested parallelism is enabled, then the new team may consist of more than one thread.

The OpenMP runtime library maintains a pool of threads that can be used as slave threads in parallel regions. When a thread encounters a parallel construct and needs to create a team of more than one thread, the thread will check the pool and grab idle threads from the pool, making them slave threads of the team. The master thread might get fewer slave threads than it needs if there is not a sufficient number of the idle threads in the pool. When the team finishes executing the parallel region, the slave threads return to the pool.

2.2 Control of Nested Parallelism

Nested parallelism can be controlled at runtime by setting various environment variables prior to execution of the program.

2.2.1 OMP_NESTED

Nested parallelism can be enabled or disabled by setting the **OMP_NESTED** environment variable or calling `omp_set_nested()` function (Section 1.11.3.9, "OMP_SET_NESTED Routine" on page 1-26).

The following example shows a team of more than one thread executing a nested parallel region when nested parallelism is enabled.

CODE EXAMPLE 2-1 Nested Parallelism Example

```
#include <omp.h>
#include <stdio.h>
void report_num_threads(int level)
{
    #pragma omp single
    {
        printf("Level %d: number of threads in the team - %d\n",
                  level, omp_get_num_threads());
    }
}

int main()
{
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(2)
    {
        report_num_threads(1);
        #pragma omp parallel num_threads(2)
        {
            report_num_threads(2);
            #pragma omp parallel num_threads(2)
            {
                report_num_threads(3);
            }
        }
    }
    return(0);
}
```

Compiling and running this program with nested parallelism enabled produces the following output:

```
% setenv OMP_NESTED TRUE
% a.out
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
```

Compare with running the same program but with nested parallelism disabled:

```
% setenv OMP_NESTED FALSE
% a.out
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 1
Level 3: number of threads in the team - 1
Level 2: number of threads in the team - 1
Level 3: number of threads in the team - 1
```

2.2.2 SUNW_MP_MAX_POOL_THREADS

The OpenMP runtime library maintains a pool of threads that can be used as slave threads in parallel regions. Setting the **SUNW_MP_MAX_POOL_THREADS** environment variable controls the number of threads in the pool. The default value is 1023.

The thread pool consists of only non-user threads that the runtime library creates. It does not include the master thread or any thread created explicitly by the user's program. If this environment variable is set to zero, the thread pool will be empty and all parallel regions will be executed by one thread.

The following example shows that a parallel region can get fewer threads if there are not sufficient threads in the pool.The code is the same as above. The number of threads needed for all the parallel regions to be active at the same time is 8. The pool needs to contain at least 7 idle threads. If we set **SUNW_MP_MAX_POOL_THREADS** to 5,

two of the four inner-most parallel regions may not be able to get all the slave threads they ask for. One possible result is shown below.

```
% setenv OMP_NESTED TRUE
% setenv SUNW_MP_MAX_POOL_THREADS 5
% a.out
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 1
Level 3: number of threads in the team - 1
```

2.2.3 SUNW_MP_MAX_NESTED_LEVELS

Environment variable **SUNW_MP_MAX_NESTED_LEVELS** controls the maximum depth of nested parallel regions that require more than one thread.

Any parallel region that has an active nested depth greater than the value of this environment variable will be executed by only one thread. A parallel region is considered not active if it is an OpenMP parallel region that has a false IF clause.

The following code will create 4 levels of nested parallel regions. If **SUNW_MP_MAX_NESTED_LEVELS** is set to 2, then nested parallel regions at nested depth of 3 and 4 are executed single-threaded.

```
#include <omp.h>
#include <stdio.h>
#define DEPTH 5
void report_num_threads(int level)
{
    #pragma omp single
    {
        printf("Level %d: number of threads in the team - %d\n",
               level, omp_get_num_threads());
    }
}
void nested(int depth)
{
    if (depth == DEPTH)
        return;

    #pragma omp parallel num_threads(2)
    {
        report_num_threads(depth);
        nested(depth+1);
    }
}
int main()
{
    omp_set_dynamic(0);
    omp_set_nested(1);
    nested(1);
    return(0);
}
```

Compiling and running this program with a maximum nesting level of 4 gives the following possible output. (Actual results will depend on how the OS schedules threads).

```
% setenv SUNW_MP_MAX_NESTED_LEVELS 4
% a.out |sort +2n
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
Level 4: number of threads in the team - 2
```

Running with the nesting level set at 2 gives the following as a possible result:

```
% setenv SUNW_MP_MAX_NESTED_LEVELS 2
% a.out |sort +2n
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 1
Level 3: number of threads in the team - 1
Level 3: number of threads in the team - 1
Level 3: number of threads in the team - 1
Level 4: number of threads in the team - 1
Level 4: number of threads in the team - 1
Level 4: number of threads in the team - 1
Level 4: number of threads in the team - 1
```

2.3 Using OpenMP Library Functions Within Nested Parallel Regions

Calls to the following OpenMP routines within nested parallel regions deserve some discussion.

- `omp_set_num_threads()`
- `omp_get_max_threads()`
- `omp_set_dynamic()`
- `omp_get_dynamic()`
- `omp_set_nested()`
- `omp_get_nested()`

The 'set' calls affect only the parallel regions at the same or inner nesting levels encountered by the calling thread. They do not affect parallel regions encountered by other threads, and they do not affect parallel regions the calling thread will later encounter in any outer levels.

The 'get' calls will return the values set by the calling thread. When a team is created, the slave threads will inherit the values from the master thread.

CODE EXAMPLE 2-2 OpenMP Function Calls Within Parallel Regions

```
#include <stdio.h>
#include <omp.h>
int main()
{
    omp_set_nested(1);
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(2)
    {
        if (omp_get_thread_num() == 0)
            omp_set_num_threads(4);       /* line A */
        else
            omp_set_num_threads(6);       /* line B */

        /* The following statement will print out
         *
         * 0: 2 4
         * 1: 2 6
         *
         * omp_get_num_threads() returns the number
         * of the threads in the team, so it is
         * the same for the two threads in the team.
         */
        printf("%d: %d %d\n", omp_get_thread_num(),
               omp_get_num_threads(),
               omp_get_max_threads());

        /* Two inner parallel regions will be created
         * one with a team of 4 threads, and the other
         * with a team of 6 threads.
         */
        #pragma omp parallel
        {
            #pragma omp master
            {
                /* The following statement will print out
                 *
                 * Inner: 4
                 * Inner: 6
                 */
                printf("Inner: %d\n", omp_get_num_threads());
            }
            omp_set_num_threads(7);      /* line C */
        }
```

```
        /* Again two inner parallel regions will be created,
         * one with a team of 4 threads, and the other
         * with a team of 6 threads.
         *
         * The omp_set_num_threads(7) call at line C
         * has no effect here, since it affects only
         * parallel regions at the same or inner nesting
         * level as line C.
         */

        #pragma omp parallel
        {
            printf("count me.\n");
        }
    }
    return(0);
}
```

Compiling and running this program gives the following as one possible result:

```
% a.out
0: 2 4
Inner: 4
1: 2 6
Inner: 6
count me.
count me.
count me.
count me.
count me.
count me.
count me.
count me.
count me.
count me.
```

2.4 Some Tips on Using Nested Parallelism

- Nesting parallel regions provides an immediate way to allow more threads to participate in the computation.

  For example, suppose you have a program that contains two levels of parallelism and the degree of parallelism at each level is 2. And, your system has four cpus and you want use all four CPUs to speed up the execution of this program. Just parallelizing any one level will use only two CPUs. You want to parallelize both levels.

- Nesting parallel regions can easily create too many threads and oversubscribe the system. Set **`SUNW_MP_MAX_POOL_THREADS`** and **`SUNW_MP_MAX_NESTED_LEVELS`** appropriately to limit the number of threads in use and prevent runaway oversubscription.

- Creating nested parallel regions adds overhead. If there is enough parallelism at the outer level and the load is balanced, generally it will be more efficient to use all the threads at the outer level of the computation than to create nested regions at the inner levels.

  For example, suppose you have a program that contains two levels of parallelism. The degree of parallelism at the outer level is 4 and the load is balanced. You have a system with four CPUs and want to use all four CPUs to speed up the execution of this program. Then, in general, using all 4 threads for the outer level could yield better performance than using 2 threads for the outer parallel region, and using the other 2 threads as slave threads for the inner parallel regions.

CHAPTER 3

Automatic Scoping in Fortran

Declaring the scope attributes of variables in an OpenMP parallel region is called *scoping*. In general, if a variable is scoped as **SHARED**, all threads share a single copy of the variable. If a variable is scoped as **PRIVATE**, each thread has its own copy of the variable. OpenMP has a rich data environment. In addition to **SHARED** and **PRIVATE**, the scope of a variable can also be declared **FIRSTPRIVATE**, **LASTPRIVATE**, **REDUCTION**, or **THREADPRIVATE**.

OpenMP requires the user to declare the scope of each variable used in a parallel region. This is a tedious and error-prone process and many find this the hardest part of using OpenMP to parallelize programs.

The Sun Studio 9 release of the Fortran 95 compiler, **f95**, provides an automatic scoping feature. The compiler analyzes the execution and synchronization pattern of a parallel region and determines what the scope of a variable should be, based on a set of scoping rules.

3.1 The Autoscoping Data Scope Clause

The autoscoping data scope clause is a Sun extension to the Fortran OpenMP specification. A user can specify a variable to be autoscoped by using one of the following two clauses.

3.1.1 __AUTO Clause

__AUTO(*list-of-variables***)**

The compiler will determine the scope of the variables listed within a parallel region. (Note the *two* underscores before **AUTO**).

The **__AUTO** clause can appear on a **PARALLEL**, **PARALLEL DO**, **PARALLEL SECTIONS**, or **PARALLEL WORKSHARE** directive.

If a variable is listed in the **__AUTO** clause, then it cannot be specified in any other data scope clause.

3.1.2 **DEFAULT(__AUTO)** Clause

Set the default scoping in this parallel region to be **__AUTO**.

The **DEFAULT(__AUTO)** clause can appear on a **PARALLEL**, **PARALLEL DO**, **PARALLEL SECTIONS**, or **PARALLEL WORKSHARE** directive.

3.2 Scoping Rules

Under automatic scoping, the compiler applies the following rules to determine the scope of a variable in a parallel region.

These rules do not apply to variables scoped implicitly by the OpenMP Specification, such as loop index variables of worksharing **DO** loops.

3.2.1 Scoping Rules For Scalar Variables

- **S1**: If the use of the variable in the parallel region is free of *data race*[1] conditions for the threads in the team executing the region, then the variable is scoped **SHARED**.
- **S2**: If in each thread executing the parallel region, the variable is always written before being read by the same thread, then the variable is scoped **PRIVATE**. The variable is scoped as **LASTPRIVATE** if it can be scoped **PRIVATE** and is read before it is written after the parallel region, and the construct is either a **PARALLEL DO** or a **PARALLEL SECTIONS**.
- **S3**: If the variable is used in a reduction operation that can be recognized by the compiler, then the variable is scoped **REDUCTION** with that particular operation type.

1. A *data race* exists when two threads can access the same shared variable at the same time with at least one thread modifying the variable. To remove a data race condition, put the accesses in a critical section or synchronize the threads.

3.2.2 Scoping Rules for Arrays

- **A1**: If the use of the array in the parallel region is free of data race conditions for the threads in the team executing the region, then the array is scoped as **SHARED**.

3.3 General Comments About Autoscoping

If a user specifies the following variables to be autoscoped by **__AUTO(***list-of-variables***)** or **DEFAULT(__AUTO)**, the compiler will scope the variable according to the implicit scoping rules in the OpenMP Specification.

- A **THREADPRIVATE** variable
- A Cray *pointee*.
- A loop iteration variable used only in sequential loops in the lexical extent of the region or worksharing DO loops that bind to the region.
- Implied **DO** or **FORALL** indices.
- Variables used only in work-sharing constructs that bind to the region, and specified in a data scope attribute clause for each such construct.

When autoscoping a variable that does not have implicit scope, the compiler checks the use of the variable against the rules above, in the order shown. If a rule matches, the compiler will scope the variable according to the matching rule. If a rule does not match, the compiler tries the next rule. If the compiler is unable to find a match, autoscoping fails for that variable.

When autoscoping of a variable fails, the variable is scoped as **SHARED**, and the binding parallel region will be serialized as if an **IF (.FALSE.)** clause were specified.

There are two reasons why autoscoping fails. One is that the use of the variable does not match any of the rules. The other is that the source code is too complex for the compiler to do a sufficient analysis. Function calls, complicated array subscripts, memory aliasing, and user-implemented synchronization are some typical causes. (See Section 3.5, "Known Limitations of the Current Implementation" on page 3-8.)

3.4 Checking the Results of Autoscoping

Use *compiler commentary* to check autoscoping results and to see if any parallel regions are serialized because autoscoping failed.

The compiler will produce an inline commentary when compiled with the **-g** debug option. This generated commentary can be viewed with the **er_src** command, as shown in CODE EXAMPLE 3-2. (The **er_src** command is provided as part of the Sun Studio software; for more information, see the `er_src`(1) man page or the *Sun Studio Performance Analyzer* manual.)

A good place to start is to compile with the **-vpara** option. A warning message will be printed out if autoscoping fails, as shown in CODE EXAMPLE 3-1.

CODE EXAMPLE 3-1 Compiling With **-vpara**

```
>cat t.f
      INTEGER X(100), Y(100), I, T
C$OMP PARALLEL DO DEFAULT(__AUTO)
      DO I=1, 100
         T = Y(I)
         CALL FOO(X)
         X(I) = T*T
      END DO
C$OMP END PARALLEL DO
      END
>f95 -xopenmp -xO3 -vpara -c t.f
"t.f", line 3: Warning: parallel region is serialized
       because the autoscoping of following variables failed
       - x
```

CODE EXAMPLE 3-2 Using Compiler Commentary

```
>cat t.f
      INTEGER X(100), Y(100), I, T
C$OMP PARALLEL DO DEFAULT(__AUTO)
      DO I=1, 100
         T = Y(I)
         X(I) = T*T
      END DO
C$OMP END PARALLEL DO
      END

>f95 -xopenmp -xO3 -g -c t.f
>er_src t.o
Source file: ./t.f
Object file: ./t.o
Load Object: ./t.o

     1.       INTEGER X(100), Y(100), I, T
     2.

Private variables in OpenMP construct below: t,i
Shared variables in OpenMP construct below: y,x
Variables autoscoped as PRIVATE in OpenMP construct below:
       i, t
Variables autoscoped as SHARED in OpenMP construct below:
       y, x
     3. C$OMP PARALLEL DO DEFAULT(__AUTO)

Loop below parallelized by explicit user directive
     4.       DO I=1, 100

Loop below scheduled with steady-state cycle count = 3
Loop below unrolled 2 times
Loop below has 1 loads, 1 stores, 0 prefetches, 0 FPadds, 0 FPmuls, and
0 FPdivs per iteration
     5.          T = Y(I)
     6.          X(I) = T*T
     7.       END DO
     8. C$OMP END PARALLEL DO
     9.
    10.       END
```

Next, a more complicated example to illustrate how the autoscoping rules work.

CODE EXAMPLE 3-3 A More Complicated Example

```
 1.       REAL FUNCTION FOO (N, X, Y)
 2.       INTEGER       N, I
 3.       REAL          X(*), Y(*)
 4.       REAL          W, MM, M
 5.
 6.       W = 0.0
 7.
 8. C$OMP PARALLEL DEFAULT(__AUTO)
 9.
10. C$OMP SINGLE
11.        M = 0.0
12. C$OMP END SINGLE
13.
14.        MM = 0.0
15.
16. C$OMP DO
17.        DO I = 1, N
18.           T = X(I)
19.           Y(I) = T
20.           IF (MM .GT. T) THEN
21.              W = W + T
22.              MM = T
23.           END IF
24.        END DO
25. C$OMP END DO
26.
27. C$OMP CRITICAL
28.        IF ( MM .GT. M ) THEN
29.           M = MM
30.        END IF
31. C$OMP END CRITICAL
32.
33. C$OMP END PARALLEL
34.
35.       FOO = W - M
36.
37.       RETURN
38.       END
```

The function **FOO()** contains a parallel region, which contains a **SINGLE** construct, a work-sharing **DO** construct and a **CRITICAL** construct. If we ignore all the OpenMP parallel constructs, what the code in the parallel region does is:

1. Copy the value in array **X** to array **Y**
2. Find the maximum positive value in **X**, and store it in **M**
3. Accumulate the value of some elements of **X** into variable **W**.

Let's see how the compiler uses the above rules to find the appropriate scopes for the variables in the parallel region.

The following variables are used in the parallel region, **I**, **N**, **MM**, **T**, **W**, **M**, **X**, and **Y**. The compiler will determine the following.

- Scalar **I** is the loop index of the work-sharing **DO** loop. The OpenMP specification mandates that **I** be scoped **PRIVATE**.
- Scalar **N** is only read in the parallel region and therefore will not cause data race, so it is scoped as **SHARED** following rule **S1**.
- Any thread executing the parallel region will execute statement 14, which sets the value of scalar **MM** to 0.0. This write will cause data race, so rule **S1** does not apply. The write happens before any read of **MM** in the same thread, so **MM** is scoped as **PRIVATE** according to rule **S2**.
- Similarly, scalar **T** is scoped as **PRIVATE**.
- Scalar **W** is read and then written at statement 21, so rules **S1** and **S2** do not apply. The addition operation is both associative and communicative, therefore, **W** is scoped as **REDUCTION(+)** according to rule **S3**.
- Scalar **M** is written in statement 11 which is inside a **SINGLE** construct. The implicit barrier at the end of the **SINGLE** construct ensures that the write in statement 11 will not happen concurrently with either the read in statement 28 or the write in statement 29, while the latter two will not happen at the same time because both are inside the same **CRITICAL** construct. No two threads can access **M** at the same time. Therefore, the writes and reads of **M** in the parallel region do not cause a data race, and, following rule **S1**, **M** is scoped **SHARED**.
- Array **X()** is only read and not written in the region, so it is scoped as **SHARED** by rule **A1**.
- The writes to array **Y()** is distributed among the threads, and no two threads will write to the same elements of **Y()**. As there is no data race, **Y()** is scoped **SHARED** according to rule **A1**.

3.5 Known Limitations of the Current Implementation

Here are the known limitations to autoscoping in the Sun Studio 9 Fortran 95 compiler.

- Only OpenMP directives are recognized and used in the analysis. OpenMP API function calls are not recognized. For example, if a program uses **`OMP_SET_LOCK()`** and **`OMP_UNSET_LOCK()`** to implement a critical section, the compiler is not able to detect the existence of the critical section. Use **`CRITICAL`** and **`END CRITICAL`** directives if possible.
- Only synchronizations specified in OpenMP synchronization directives, such as **`BARRIER`** and **`MASTER`**, are recognized and used in the analysis. User-implemented synchronizations, such as busy-waiting, are not recognized.
- Autoscoping is not supported when compiling with **`-xopenmp=noopt`**.

CHAPTER 4

Implementation-Defined Behaviors

This chapter notes specific issues in the OpenMP 2.0 Fortran and C/C++ specifications that are implementation dependent. For last-minute information regarding the latest compiler releases, see the C, C++, and Fortran readme files.

- **Scheduling**

  The default, in the absence of an explicit **OMP_SCHEDULE** environment variable, or an explicit **SCHEDULE** clause, is **static** scheduling.

- **Number of Threads**

  Without an explicit **num_threads()** clause, call to the **omp_set_num_threads**() function, or an explicit definition of the **OMP_NUM_THREADS** environment variable, the default number of threads in a team is 1.

- **Dynamic Adjustment of Threads**

  If dynamic adjustment is enabled, the number of threads in the team is adjusted to be the minimum of:

  the number of threads the user requested

  1 + the number of available threads in the pool

  the number of available processors

  If dynamic adjustment is disabled, then the number of threads in the team will be the minimum of:

  the number of threads the user requested

  1 + the number of available threads in the pool

  If the number of threads supplied is less than the number the user requested and **SUNW_MP_WARN** is set to **TRUE** or a callback function is registered through a call to `sunw_mp_register_warn()`, a warning message will be issued.

In exceptional situations, such as when there is lack of system resources, the number of threads supplied will be less than described above. In these situations, if dynamic adjustment is disabled and **SUNW_MP_WARN** is set to **TRUE** or a callback function is registered via a call to `sunw_mp_register_warn()`, a warning message will be issued.

Refer to Chapter 2 for more information about the pool of threads and the nested parallelism execution model.

- **Nested Parallelism**

  Nested parallelism is supported. Nested parallel regions can be executed by multiple threads. Nested parallelism is disabled by default. Set the `OMP_NESTED` environment variable, or call the `omp_set_nested()` function to enable it. See Chapter 2.

- **ATOMIC Directive**

  This implementation replaces all **ATOMIC** directives and pragmas by enclosing the target statement in a critical region.

- **GUIDED: Determination of Chunk Sizes**

  The default chunk size for **SCHEDULE(GUIDED)** when *chunksize* is not specified is 1. The OpenMP runtime library uses the following formula for computing the chunk sizes for a loop with **GUIDED** scheduling:

  chunksize = *unassigned_iterations* / (*weight* * *num_threads*)

  where:

  unassigned_iterations is the number of iterations in the loop that have not yet been assigned to any thread;

  weight is a floating-point constant that can be set by the user at runtime with the **SUNW_MP_GUIDED_WEIGHT** environment variable (Section 5.3, "OpenMP Environment Variables" on page 5-5). The current default, if not specified, assumes *weight* is 2.0;

  num_threads is the number of threads used to execute the loop.

  Choice of the weighting value affects the size of the initial and subsequent chunks of iterations assigned to threads in loops, and has a direct affect on load balancing. Experimental results show that the default weight of 2.0 works well generally. However some applications could benefit from a different weight value.

- **Explicitly Threaded Programs**

  Programs using POSIX or Solaris threads can contain OpenMP directives or call routines that contain OpenMP directives.

- **Runtime Warnings**
  - Setting the **SUNW_MP_WARN** environment variable (Section 5.3, "OpenMP Environment Variables" on page 5-5) enables runtime validity checking by the OpenMP multitasking library.

    For example, the following code will fall into an endless loop as threads wait at different barriers, and must be terminated with a control-C from the terminal:

```
% cat bad1.c

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

int
main(void)
{
   omp_set_dynamic(0);
   omp_set_num_threads(4);

   #pragma omp parallel
   {
        int i = omp_get_thread_num();

        if (i % 2) {
            printf("At barrier 1.\n");
            #pragma omp barrier
        }
   }
   return 0;
}
% cc -xopenmp -xO3 bad1.c
% ./a.out              run the program
At barrier 1.
At barrier 1.
                       program hung in endless loop
Control-C   to terminate execution
```

But if we set **SUNW_MP_WARN** before execution, the runtime library will detect the problem:

```
% setenv SUNW_MP_WARN TRUE
% ./a.out
At barrier 1.
At barrier 1.
WARNING (libmtsk): Threads at barrier from different directives.
    Thread at barrier from bad1.c:11.
    Thread at barrier from bad1.c:17.
    Possible Reasons:
    Worksharing constructs not encountered by all threads in the team in the
       same order.
    Incorrect placement of barrier directives.
```

- The C compiler also provides a function that can be used to register a callback function when errors are detected. The registered callback function is called and passed a pointer to an error message string as an argument whenever an error is detected

  ```
  int sunw_mp_register_warn(void (*func) (void *) )
  ```

  Access to the prototype for this function requires adding
  `#include <sunw_mp_misc.h>`

For example:

```
% cat bad2.c
#include <omp.h>
#include <sunw_mp_misc.h>
#include <stdio.h>

void handle_warn(void *msg)
{
    printf("handle_warn: %s\n", (char *)msg);
}

void set(int i)
{
    static int k;
#pragma omp critical
    {
        k++;
    }
#pragma omp barrier
}

int main(void)
{
  int i, rc;
  omp_set_dynamic(0);
  omp_set_num_threads(4);
  if (sunw_mp_register_warn(handle_warn) != 0) {
      printf ("Installing callback failed\n");
  }
#pragma omp parallel for
  for (i = 0; i < 20; i++) {
      set(i);
  }
  return 0;
}

% cc -xopenmp -xO3 bad2.c
% a.out
handle_warn: WARNING (libmtsk): at bad2.c:21 Barrier is not permitted
in dynamic extent of for / DO.
```

`handle_warn()` is installed as the callback handler function when an error is detected by the OpenMP runtime library. The handler in this example merely prints the error message passed to it from the library, but could be used to trap certain errors.

CHAPTER 5

Compiling for OpenMP

This chapter describes how to compile programs that utilize the OpenMP API.

To run a parallelized program in a multithreaded environment, you must set the **OMP_NUM_THREADS** environment variable prior to program execution. This tells the runtime system the maximum number of threads the program can create. The default is 1. In general, set **OMP_NUM_THREADS** to a value no larger than the available number of processors on the target platform. Set **OMP_DYNAMIC** to **FALSE** to use the number of threads specified by **OMP_NUM_THREADS**.

The compiler readme files contain information about limitations and known deficiencies regarding their OpenMP implementation. Readme files are viewable directly by invoking the compiler with the `-xhelp=readme` flag, or by pointing an HTML browser to the documentation index for the installed software at

```
file:/opt/SUNWspro/docs/index.html
```

5.1 Compiler Options To Use

To enable explicit parallelization with OpenMP directives, compile your program with the **cc**, **CC**, or **f95** option flag **-xopenmp**. This flag can take an optional keyword argument. (The **f95** compiler accepts both **-xopenmp** and **-openmp** as synonyms.)

The **-xopenmp** flag accepts the following keyword sub-options.

-xopenmp=parallel	Enables recognition of OpenMP pragmas. The minimum optimization level for -xopenmp=parallel is -xO3. The compiler changes the optimization from a lower level to -xO3 if necessary, and issues a warning.
-xopenmp=noopt	Enables recognition of OpenMP pragmas. The compiler does not raise the level if it is lower than -xO3. If you explicitly set the optimization levellower than -xO3, as in -xO2 -openmp=noopt the compiler will issue an error. If you do not specify an optimization level with -openmp=noopt, the OpenMP pragmas are recognized, the program is parallelized accordingly, but no optimization is done. (This sub-option applies to cc and f95 only; CC issues a warning if specified, and no OpenMP parallelization is done.)
-xopenmp=stubs	This option is no longer supported. An OpenMP stubs library is provided for users' convenvience. To compile an OpenMP program that calls OpenMP library functions but ignores the OpenMP pragmas, compile the program without an -xopenmp option, and link the object files with the libompstubs.a library. For example, % **cc omp_ignore.c -lompstubs** Linking with both libompstubs.a and the OpenMP runtime library libmtsk.so is unsupported and may result in unexpected behavior.
-xopenmp=none	Disables recognition of OpenMP pragmas and does not change the optimization level.

Additional Notes:

- If you do not specify **-xopenmp** on the command line, the compiler assumes **-xopenmp=none** (disabling recognition of OpenMP pragmas).
- If you specify **-xopenmp** but without a keyword sub-option, the compiler assumes **-xopenmp=parallel**.
- Do not specify **-xopenmp** together with **-xparallel** or **-xexplicitpar** on the command line.
- Specifying **-xopenmp=parallel** or **noopt** will define the **_OPENMP** preprocessor token to be YYYYMM (specifically **200203L** for C/C++ and **200011** for Fortran 95).
- When debugging OpenMP programs with dbx, compile with **-xopenmp=noopt -g**

- The default optimization level for **-xopenmp** might change in future releases. Compilation warning messages can be avoided by specifying an appropriate optimization level explicitly.
- With Fortran 95, **-xopenmp** , **-xopenmp=parallel**, **-xopenmp=noopt** will add **-stackvar** automatically.
- If you compile with **-xopenmp** when building a dynamic (**.so**) library, you must also specify **-xopenmp** when linking the executable, and the compiler used to create the executable must be at least as new as the compiler that built the dynamic library with **-xopenmp**. Using different compiler versions with **-xopenmp** to create the executable and the library, can result in unexpected behavior.

5.2 Fortran 95 OpenMP Validation

You can obtain a static, interprocedural validation of a Fortran 95 program's OpenMP directives by using the **f95** compiler's global program checking feature. Enable OpenMP checking by compiling with the **-XlistMP** flag. (Diagnostic messages from **-XlistMP** appear in a separate file created with the name of the source file and a **.lst** extension). The compiler will diagnose the following violations and parallelization inhibitors:

- Violations in the specifications of parallel directives, including improper nesting.
- Parallelization inhibitors due to data usage, detected by interprocedural dependence analysis.
- Parallelization inhibitors detected by interprocedural pointer analysis.

For example, compiling a source file `ord.f` with `-XlistMP` produces a diagnostic file `ord.lst`:

```
FILE  "ord.f"
     1  !$OMP PARALLEL
     2  !$OMP DO ORDERED
     3                  do i=1,100
     4                          call work(i)
     5                  end do
     6  !$OMP END DO
     7  !$OMP END PARALLEL
     8
     9  !$OMP PARALLEL
    10  !$OMP DO
    11                  do i=1,100
    12                          call work(i)
    13                  end do
    14  !$OMP END DO
    15  !$OMP END PARALLEL
    16                  end
    17                  subroutine work(k)
    18  !$OMP ORDERED
         ^
**** ERR-OMP:  It is illegal for an ORDERED directive to bind to a
directive (ord.f, line 10, column 2) that does not have the
ORDERED clause specified.
    19                  write(*,*) k
    20  !$OMP END ORDERED
    21                  return
    22                  end
```

In this example, the **ORDERED** directive in subroutine **WORK** receives a diagnostic that refers to the second **DO** directive because it lacks an **ORDERED** clause.

5.3 OpenMP Environment Variables

The OpenMP specifications define four environment variables that control the execution of OpenMP programs. These are summarized in the following table.

TABLE 5-1 OpenMP Environment Variables

Environment Variable	Function
OMP_SCHEDULE	Sets schedule type for **DO**, **PARALLEL DO**, **parallel for, for**, directives/pragmas with schedule type **RUNTIME** specified. If not defined, a default value of **STATIC** is used. *value* is *"type[,chunk]"* Example: `setenv OMP_SCHEDULE "GUIDED,4"`
OMP_NUM_THREADS *or* **PARALLEL**	Sets the number of threads to use during execution of a parallel region. This number can be overriden by a **NUM_THREADS** clause, or a call to **OMP_SET_NUM_THREADS()**. If not set, a default of 1 is used. *value* is a positive integer. For compatibility with legacy programs, setting the **PARALLEL** environment variable has the same effect as setting **OMP_NUM_THREADS**. However, if they are both set to different values, the runtime library will issue an error message. Example: `setenv OMP_NUM_THREADS 16`
OMP_DYNAMIC	Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. If not set, a default value of **TRUE** is used. *value* is either **TRUE** or **FALSE**. Example: `setenv OMP_DYNAMIC FALSE`
OMP_NESTED	Enables or disables nested parallelism. *value* is either **TRUE** or **FALSE**. The default is **FALSE**. Example: `setenv OMP_NESTED FALSE`

Additional multiprocessing environment variables affect execution of OpenMP programs and are not part of the OpenMP specifications. These are summarized in the following table.

TABLE 5-2 Multiprocessing Environment Variables

Environment Variable	Function
SUNW_MP_WARN	Controls warning messages issued by the OpenMP runtime library. If set to **TRUE** the runtime library issues warning messages to `stderr`; **FALSE** disables warning messages. The default is **FALSE**. The OpenMP runtime library has the ability to check for many common OpenMP violations, such as incorrect nesting and deadlocks. Runtime checking does add overhead to the execution of the program. See "Runtime Warnings" on page 4-3. Example: **setenv SUNW_MP_WARN TRUE**
SUNW_MP_THR_IDLE	Controls the end-of-task status of each helper thread executing the parallel part of a program. You can set the value to **SPIN**, **SLEEP** *n***s**, or **SLEEP** *n***ms**. The default is **SLEEP** — the thread sleeps after completing a parallel task until a new parallel task arrives. Choosing **SLEEP** *time* specifies the amount of time a helper thread should spin-wait after completing a parallel task. If, while a thread is spinning, a new task arrives for the thread, the thread executes the new task immediately. Otherwise, the thread goes to sleep and is awakened when a new task arrives. *time* may be specified in seconds, (*n***s**) or just (*n*), or milliseconds, (*n***ms**). **SLEEP** with no argument puts the thread to sleep immediately after completing a parallel task. **SLEEP**, **SLEEP (0)**, **SLEEP (0s)**, and **SLEEP (0ms)** are all equivalent. Example: **setenv SUNW_MP_THR_IDLE SLEEP(50ms)**
SUNW_MP_PROCBIND	The **SUNW_MP_PROCBIND** environment variable can be used to bind LWPs (lightweight processes) of an OpenMP program to processors. Performance can be enhanced with processor binding, but performance degradation will occur if multiple LWPs are bound to the same processor. See Section 5.4, "Processor Binding" on page 5-7 for details.

TABLE 5-2 Multiprocessing Environment Variables *(Continued)*

Environment Variable	Function
SUNW_MP_MAX_POOL_THREADS	Specifies the maximum size of the thread pool. The thread pool contains only non-user threads that the OpenMP runtime library creates. It does not contain the master thread or any threads created explicitly by the user's program. If this environment variable is set to zero, the thread pool will be empty and all parallel regions will be executed by one thread. The default, if not specified, is 1023. See Section 2.2, "Control of Nested Parallelism" on page 2-2 for details.
SUNW_MP_MAX_NESTED_LEVELS	Specifies the maximum depth of active nested parallel regions. Any parallel region that has an active nested depth greater than the value of this environment variable will be executed by only one thread. A parallel region is considered not active if it is an OpenMP parallel region that has a false **IF** clause. The default, if not specified, is 4. See Section 2.2, "Control of Nested Parallelism" on page 2-2 for details.
STACKSIZE	Sets the stack size for each thread. The value is in kilobytes. The default thread stack sizes are 4 Mb on 32-bit SPARC V8 and x86 platforms, and 8 Mb on 64-bit SPARC V9 and x86 platforms. Example: **setenv STACKSIZE 8192** *sets the thread stack size to 8 Mb*
SUNW_MP_GUIDED_WEIGHT	Sets the weighting factor used to determine the size of chunks assigned to threads in loops with **GUIDED** scheduling. The value should be a positive floating-point number, and will apply to all loops with **GUIDED** scheduling in the program. If not set, the default value assumed is 2.0.

5.4 Processor Binding

Processor binding, when used along with static scheduling, benefits applications that exhibit a certain data reuse pattern where data accessed by a thread in a parallel region will be in the local cache from a previous invocation of a parallel region.

By default, lightweight processes, LWPs, are not bound to processors. It is left up to the Solaris OS to schedule LWPs onto processors. The multitasking routines in the OpenMP runtime library, `libmtsk`, always use a one-to-one threading model; that is, each thread corresponds to a single LWP.

The value specified by the **SUNW_MP_PROCBIND** environment variable denotes the "logical" processor identifiers (IDs) to which the LWPs are to be bound. Logical processor IDs are consecutive integers that start with 0, and may or may not be identical to the actual processor IDs. If *n* processors are available online, then their virtual processor IDs are 0, 1, ..., *n*-1, in the order presented by `psrinfo`(1M).

The mapping between logical processor IDs and real processor IDs is dependent on the system. On most systems, real processor IDs are sequential; however, removing system boards may cause holes in the range. On some systems, IDs are in groups of 4 with gaps of 32 between the beginning of each group; thus processors would be numbered 0, 1, 2, 3, 32, 33, 34, 35 and so forth.

The number of threads created by `libmtsk` is determined by environment variables and/or API calls in the user's program. **SUNW_MP_PROCBIND** specifies a set of logical processors as described below. LWPs are bound to that set of logical processors in a cyclic fashion. If the number of LWPs is less than the number of processors, then some processors do not have LWPs bound to them. If the number of LWPs is greater than the number of processors, them some processors will have more than one LWP bound to them.

The value specified for **SUNW_MP_PROCBIND** can be one of the following:

- The string **TRUE** or **FALSE** (or lower case). For example,
  `% setenv SUNW_MP_PROCBIND false`
- A non-negative integer. For example,
  `% setenv SUNW_MP_PROCBIND 2`
- A list of two or more non-negative integers separated by one or more spaces. For example,
  `% setenv SUNW_MP_PROCBIND "0 2 4 6"`
- Two non-negative integers, *n1* and *n2*, separated by a minus ("**-**"); *n1* must be less than or equal to *n2*. For example,
  `% setenv SUNW_MP_PROCBIND "0-6"`

If the value specified for **SUNW_MP_PROCBIND** is **FALSE**, then no processor binding is performed. This is the default behavior.

If the value specified for **SUNW_MP_PROCBIND** is **TRUE**, then it is as if it were the integer 0.

If the value specified for **SUNW_MP_PROCBIND** is a non-negative integer, then that integer specifies the starting logical processor ID to which LWPs should be bound. LWPs will be bound to processors in a round-robin fashion, starting with the specified logical processor ID, and wrapping around to logical processor ID 0 after logical processor ID *n*-1.

If the value specified for **SUNW_MP_PROCBIND** is a list of two or more non-negative integers, then LWPs will be bound in a round-robin fashion to the specified logical processor IDs. No IDs other then those in the list will be used.

If the value specified for **SUNW_MP_PROCBIND** is two non-negative integers separated by a minus ("**-**"), then LWPs will be bound in a round-robin fashion to processors in the range that begins with the first logical processor ID and ends with the second logical processor ID. No IDs other than those mentioned in the range will be used.

If the value specified for **SUNW_MP_PROCBIND** does not conform to one of the forms described above, or if an invalid logical processor ID is given, then the environment variable **SUNW_MP_PROCBIND** will be ignored and LWPs will not be bound to processors. If warnings are enabled, a warning message will be issued in this case.

5.5 Stacks and Stack Sizes

The executing program maintains a main memory stack for the initial thread executing the program, as well as distinct stacks for each slave thread. Stacks are temporary memory address spaces used to hold arguments and automatic variables during invocation of a subprogram or function reference.

In general, the default main stack size is 8 megabytes. Compiling Fortran programs with the `f95` **-stackvar** option forces the allocation of local variables and arrays on the stack as if they were automatic variables. Use of **-stackvar** with OpenMP programs is implied with explicitly parallelized programs because it improves the optimizer's ability to parallelize calls in loops. (See the *Fortran User's Guide* for a discussion of the **-stackvar** flag.) However, this may lead to stack overflow if not enough memory is allocated for the stack.

Use the **limit** C-shell command, or the `ulimit` ksh/sh command, to display or set the size of the main stack.

Each slave thread of an OpenMP program has its own thread stack. This stack mimics the initial (or main) thread stack but is unique to the thread. The thread's **PRIVATE** arrays and variables (local to the thread) are allocated on the thread stack. The default size is 4 megabytes on 32-bit SPARC V8 and x86 platforms, and 8 megabytes on 64-bit SPARC V9 and x86 platforms. The size of the helper thread stack is set with the **STACKSIZE** environment variable.

```
demo% setenv STACKSIZE 16384   <-Set thread stack size to 16 Mb (C shell)

demo% STACKSIZE=16384          <-Same, using Bourne/Korn shell
demo% export STACKSIZE
```

Finding the best stack size might have to be determined by trial and error. If the stack size is too small for a thread to run it may cause silent data corruption in neighboring threads, or segmentation faults. If you are unsure about stack

overflows, compile your Fortran, C, or C++ programs with the **-xcheck=stkovf** flag to force a segmentation fault on stack overflow. This stops the program before any data corruption can occur.

CHAPTER 6

Converting to OpenMP

This chapter gives guidelines for converting legacy programs using Sun or Cray directives and pragmas to OpenMP.

6.1 Converting Legacy Fortran Directives

Legacy Fortran programs use either Sun or Cray style parallelization directives. A description of these directives can be found in the chapter *Parallelization* in the *Fortran Programming Guide*.

6.1.1 Converting Sun-Style Fortran Directives

The following tables give OpenMP near equivalents to Sun parallelization directives and their subclauses. These are only suggestions.

TABLE 6-1 Converting Sun Parallelization Directives to OpenMP

Sun Directive	Equivalent OpenMP Directive
`C$PAR DOALL` *[qualifiers]*	`!$omp parallel do` *[qualifiers]*

TABLE 6-1 Converting Sun Parallelization Directives to OpenMP *(Continued)*

Sun Directive	Equivalent OpenMP Directive

`C$PAR DOSERIAL`	No exact equivalent. You can use:
	`!$omp master`
	loop
	`!$omp end master`

`C$PAR DOSERIAL*`	No exact equivalent. You can use:
	`!$omp master`
	loopnest
	`!$omp end master`

`C$PAR TASKCOMMON` *block[,...]*	`!$omp threadprivate (/block/[,...])`

The `DOALL` directive can take the following optional qualifier clauses.

TABLE 6-2 `DOALL` Qualifier Clauses and OpenMP Equivalent Clauses

Sun DOALL Clause	OpenMP PARALLEL DO Equivalent Clauses
`PRIVATE(`*v1,v2,...*`)`	`private(`*v1,v2,...*`)`
`SHARED(`*v1,v2,...*`)`	`shared(`*v1,v2,...*`)`
`MAXCPUS(`*n*`)`	`num_threads(`*n*`)`. No exact equivalent.
`READONLY(`*v1,v2,...*`)`	No exact equivalent. You can achieve the same effect by using `firstprivate(`*v1,v2,...*`)`.
`STOREBACK(`*v1,v2,...*`)`	`lastprivate(`*v1,v2,...*`)`.
`SAVELAST`	No exact equivalent. You can achieve the same effect by using `lastprivate(`*v1,v2,...*`)`.
`REDUCTION(`*v1,v2,...*`)`	`reduction(operator:`*v1,v2,...)* Must supply the reduction operator as well as the list of variables.
`SCHEDTYPE(`*spec*`)`	`schedule(`*spec*`)` (See TABLE 6-3)

The `SCHEDTYPE(`*spec*`)` clause accepts the following scheduling specifications.

TABLE 6-3 `SCHEDTYPE` Scheduling and OpenMP `schedule` Equivalents

SCHEDTYPE(spec)	OpenMP schedule(*spec*) Clause Equivalent
`SCHEDTYPE(STATIC)`	`schedule(static)`

TABLE 6-3 SCHEDTYPE Scheduling and OpenMP schedule Equivalents *(Continued)*

SCHEDTYPE(spec)	OpenMP schedule(*spec*) Clause Equivalent
SCHEDTYPE(SELF(*chunksize*))	schedule(dynamic,*chunksize*) Default *chunksize* is 1.
SCHEDTYPE(FACTORING(*m*))	No exact equivalent.
SCHEDTYPE(GSS(*m*))	schedule(guided, *m*) Default *m* is 1.

6.1.1.1 Issues Between Sun-Style Fortran Directives and OpenMP

- Scoping of private variables must be declared explicitly with OpenMP. With Sun directives, the compiler uses its own default scoping rules for variables not explicitly scoped in a **PRIVATE** or **SHARED** clause: all scalars are treated as **PRIVATE**, and all array references are **SHARED**. With OpenMP, the default data scope is **SHARED** unless a **DEFAULT(PRIVATE)** clause appears on the **PARALLEL DO** directive. A **DEFAULT(NONE)** clause causes the compiler to flag variables not scoped explicitly. However, see Chapter 3 for information on autoscoping in Fortran.
- Since there is no **DOSERIAL** directive, mixing automatic and explicit OpenMP parallelization may have different effects: some loops may be automatically parallelized that would not have been with Sun directives.
- OpenMP provides a richer parallelism model by providing parallel regions and parallel sections. It could be possible to get better performance by redesigning the parallelism strategies of a program that uses Sun directives to take advantage of these features of OpenMP.

6.1.2 Converting Cray-Style Fortran Directives

Cray-style Fortran parallelization directives are identical to Sun-style except that the sentinel that identifies these directives is **!MIC$**. Also, the set of qualifier clauses on the **!MIC$ DOALL** is different.

TABLE 6-4 OpenMP Equivalents for Cray-Style DOALL Qualifier Clauses

Cray DOALL Clause	OpenMP PARALLEL DO Equivalent Clauses
SHARED(*v1,v2,...*)	SHARED(*v1,v2,...*)
PRIVATE(*v1,v2,...*)	PRIVATE(*v1,v2,...*)
AUTOSCOPE	No equivalent. Scoping must be explicit, or with the **DEFAULT** clause, or with the **__AUTO** clause

TABLE 6-4 OpenMP Equivalents for Cray-Style DOALL Qualifier Clauses *(Continued)*

Cray DOALL Clause	OpenMP PARALLEL DO Equivalent Clauses
SAVELAST	No exact equivalent. You can achieve the same effect by using lastprivate.
MAXCPUS(n)	num_threads(*n*) . No exact equivalent.
GUIDED	schedule(guided, *m*) Default *m* is 1.
SINGLE	schedule(dynamic,1)
CHUNKSIZE(n)	schedule(dynamic,*n*)
NUMCHUNKS(m)	schedule(dynamic,*n/m*) where *n* is the number of iterations

6.1.2.1 Issues Between Cray-Style Fortran Directives and OpenMP Directives

The differences are the same as for Sun-style directives, except that there is no equivalent for the Cray AUTOSCOPE.

6.2 Converting Legacy C Pragmas

The C compiler accepts legacy pragmas for explicit parallelization. These are described in the *C User's Guide*. As with the Fortran directives, these are only suggestions.

The legacy parallelization pragmas are:

TABLE 6-5 Converting Legacy C Parallelization Pragmas to OpenMP

Legacy C Pragma	Equivalent OpenMP Pragma
#pragma MP taskloop *[clauses]*	#pragma omp parallel for *[clauses]*
#pragma MP serial_loop	No exact equivalent. You can use #pragma omp master *loop*
#pragma MP serial_loop_nested	No exact equivalent. You can use #pragma omp master *loopnest*

The `taskloop` pragma can take on one or more of the following optional clauses.

TABLE 6-6 `taskloop` Optional Clauses and OpenMP Equivalents

`taskloop` **Clause**	**OpenMP** `parallel for` **Equivalent Clause**
`maxcpus(`*n*`)`	No exact equivalent. Use `num_threads(`*n*`)`
`private(`*v1,v2,...*`)`	`private(`*v1,v2,...*`)`
`shared(`*v1,v2,...*`)`	`shared(`*v1,v2,...*`)`
`readonly(`*v1,v2,...*`)`	No exact equivalent. You can achieve the same effect by using `firstprivate(`*v1,v2,...*`)`.
`storeback(`*v1,v2,...*`)`	You can achieve the same effect by using `lastprivate(`*v1,v2,...*`)`.
`savelast`	No exact equivalent. You can achieve the same effect by using `lastprivate(`*v1,v2,...*`)`.
`reduction(`*v1,v2,...*`)`	`reduction(operator:`*v1,v2,...*`)`. Must supply the reduction operator as well as the list of variables.
`schedtype(`*spec*`)`	`schedule(`*spec*`)` (See TABLE 6-7)

The `schedtype(`*spec*`)` clause accepts the following scheduling specifications.

TABLE 6-7 `SCHEDTYPE` Scheduling and OpenMP `schedule` Equivalents

schedtype(spec)	**OpenMP schedule(*spec*) Clause Equivalent**
`SCHEDTYPE(STATIC)`	`schedule(static)`
`SCHEDTYPE(SELF(`*chunksize*`))`	`schedule(dynamic,`*chunksize*`)` Note: Default *chunksize* is 1.
`SCHEDTYPE(FACTORING(`*m*`))`	No exact equivalent.
`SCHEDTYPE(GSS(`*m*`))`	`schedule(guided,` *m*`)` Default *m* is 1.

6.2.1 Issues Between Legacy C Pragmas and OpenMP

- OpenMP scopes variables declared within a parallel construct as **`private`**. A **`default(none)`** clause on a **`#pragma omp parallel`** `for` directive causes the compiler to flag variables not scoped explicitly.
- Since there is no **`serial_loop`** directive, mixing automatic and explicit OpenMP parallelization may have different effects: some loops may be automatically parallelized that would not have been with legacy C directives.

- Because OpenMP provides a richer parallelism model, it is often possible to get better performance by redesigning the parallelism strategies of a program that uses legacy C directives to take advantage of these features.

CHAPTER 7

Performance Considerations

Once you have a correct, working OpenMP program, it is worth considering its overall performance. There are some general techniques that you can utilize to improve the efficiency and scalability of an OpenMP application, as well as techniques specific to the Sun platforms. These are discussed briefly here.

For additional information, see *Techniques for Optimizing Applications: High Performance Computing*, by Rajat Garg and Ilya Sharapov, which is available from http://www.sun.com/books/catalog/garg.xml

Also, visit the Sun Developer portal for occasional articles and case studies regarding performance analysis and optimization of OpenMP applications, at http://developers.sun.com/prodtech/cc/.

7.1 Some General Recommendations

The following are some general techniques for improving performance of OpenMP applications.

- Minimize synchronization.
  - Avoid or minimize the use of **BARRIER**, **CRITICAL** sections, **ORDERED** regions, and locks.
  - Use the **NOWAIT** clause where possible to eliminate redundant or unnecessary barriers. For example, there is always an implied barrier at the end of a parallel region. Adding **NOWAIT** to a final **DO** in the region eliminates one redundant barrier.
  - Use named **CRITICAL** sections for fine-grained locking.
  - Use explicit **FLUSH** with care. Flushes can cause data cache restores to memory, and subsequent data accesses may require reloads from memory, all of which decrease efficiency.

- If a **SHARED** variable in a parallel region is read by the threads executing the region, but not written to by any of the threads, then specify that variable to be **FIRSTPRIVATE** instead of **SHARED**. This avoids accessing the variable by dereferencing a pointer, and avoids cache conflicts.
- Avoid wasting resources by requesting more threads than you plan to use. Experiment with **SUNW_MP_THR_IDLE**, to put spinning worker threads to sleep when not needed. See Section 5.3, "OpenMP Environment Variables" on page 5-5.
- Parallelize at the highest level possible, such as outer **DO/FOR** loops. Enclose multiple loops in one parallel region. In general, make parallel regions as large as possible to reduce parallelization overhead. For example:

This construct is less efficient:

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

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

than this one:

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

  !$OMP DO
    ....
  !$OMP END DO

!$OMP END PARALLEL
```

- Use **PARALLEL DO/FOR** instead of worksharing **DO/FOR** directives in parallel regions. The **PARALLEL DO/FOR** is implemented more efficiently than a general parallel region containing possibly several loops. For example:

This construct is less efficient:

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

than this one:

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

- Use **SUNW_MP_PROCBIND** to bind lightweight processes (LWPs) to processors. Processor binding, when used along with static scheduling, benefits applications that exhibit a certain data reuse pattern where data accessed by a thread in a parallel region will be in the local cache from a previous invocation of a parallel region. See Section 5.4, "Processor Binding" on page 5-7.
- Use **MASTER** instead of **SINGLE** wherever possible.
  - The **MASTER** directive is implemented as an **IF**-statement with no implicit **BARRIER** :

    ```
    IF(omp_get_thread_num() == 0) {...}
    ```

  - The **SINGLE** directive is implemented similar to other worksharing constructs. Keeping track of which thread reached **SINGLE** first adds additional runtime overhead. There is an implicit **BARRIER** if **NOWAIT** is not specified. It is less efficient.
- Choose the appropriate loop scheduling.
  - **STATIC** causes no synchronization overhead and can maintain data locality when data fits in cache. However, **STATIC** may lead to load imbalance.
  - **DYNAMIC,GUIDED** incurs a synchronization overhead to keep track of which chunks have been assigned. And, while these schedules could lead to poor data locality, they can improve load balancing. Experiment with different chunk sizes.
- Use **LASTPRIVATE** with care, as it has the potential of high overhead.
  - Data needs to be copied from private to shared storage upon return from the parallel construct.

- The compiled code checks which thread executes the logically last iteration. This imposes extra work at the end of each chunk in a parallel **DO**/**FOR**. The overhead adds up if there are many chunks.

- Use efficient thread-safe memory management.
  - Applications could be using **malloc()** and **free()** explicitly, or implicitly in the compiler-generated code for dynamic/allocatable arrays, vectorized intrinsics, and so on.
  - The thread-safe **malloc()** and **free()** in **libc** have a high synchronization overhead caused by internal locking. Faster versions can be found in the **libmtmalloc** library. Link with **-lmtmalloc** to use **libmtmalloc**.

7.2 False Sharing And How To Avoid It

Careless use of shared memory structures with OpenMP applications can result in poor performance and limited scalability. Multiple processors updating adjacent shared data in memory can result in excessive traffic on the multiprocessor interconnect and, in effect, cause serialization of computations.

7.2.1 What Is *False Sharing*?

Most high performance processors, such as UltraSPARC III, insert a cache buffer between slow memory and the high speed registers of the CPU. Accessing a memory location causes a slice of actual memory (a *cache line*) containing the memory location requested to be copied into the cache. Subsequent references to the same memory location or those around it can probably be satisfied out of the cache until the system determines it is necessary to maintain the coherency between cache and memory and restore the cache line back to memory.

However, simultaneous updates of individual elements in the same cache line coming from different processors invalidates entire cache lines, even though these updates are logically independent of each other. Each update of an individual element of a cache line marks the line as *invalid*. Other processors accessing a different element in the same line see the line marked as *invalid.* They are forced to fetch a fresh copy of the line from memory, even though the element accessed has not been modified. This is because cache coherency is maintained on a cache-line basis, and not for individual elements. As a result there will be an increase in interconnect traffic and overhead. Also, while the cache-line update is in progress, access to the elements in the line is inhibited.

This situation is called *false sharing*, and could be a significant cause of poor performance and scalability in OpenMP applications.

False sharing degrades performance when all of the following conditions occur.

- Shared data is modified by multiple processors.
- Multiple processors update data within the same cache line.
- This updating occurs very frequently (for example, in a tight loop).

Note that shared data that is read-only in a loop does not lead to false sharing.

7.2.2 Reducing False Sharing

Careful analysis of those parallel loops that play a major part in the execution of an application can reveal performance scalability problems caused by false sharing. In general, false sharing can be reduced by

- changing the structure and use of shared data into private data,
- increasing the problem size (iteration length),
- changing the mapping of iterations to processors to give each processor more work per iteration (*chunk size*),
- utilizing the compiler's optimization features to eliminate memory loads and stores.

7.3 Operating System Tuning Features

Starting with the Solaris 9 release, the operating system provides scalability and high performance for the SunFire™ systems. New features introduced with Solaris 9 OS that improve the performance of OpenMP programs without hardware upgrades are Memory Placement Optimizations (MPO) and Multiple Page Size Support (MPSS), among others.

MPO allows the OS to allocate pages close to the processors that access those pages. SunFire 6800, SunFire 15K, and SunFire E25K systems have different memory latencies within the same UniBoard™ versus between different UniBoards. The default MPO policy, called *first-touch*, allocates memory on the UniBoard containing the processor that first touches the memory. The first-touch policy can greatly improve the performance of applications where data accesses are made mostly to the memory local to each processor with first-touch placement. Compared to a random memory placement policy where the memory is evenly distributed throughout the system, the memory latencies for applications can be lowered and the bandwidth increased, leading to higher performance.

MPSS allows a program to use different page sizes for different regions of virtual memory. The default page size for Solaris 9 OS is 8KB. With the 8KB page size, applications that use large memory can have a lot of TLB misses, since the number of TLB entries on UltraSPARC™ III Cu and UltraSPARC IV allow accesses to only a few megabytes of memory. UltraSPARC III Cu and UltraSPARC IV support four different page sizes: 8 KB, 64 KB, 512 KB, and 4MB. With MPSS, user processes can request one of these four page sizes. Thus MPSS can significantly reduce the number of TLB misses and lead to improved performance for applications that use a large amount of memory.

Index

Symbols

__AUTO, 3-1

A

accessible documentation, -xv
automatic scoping, 3-1

B

barrier, 1-10
binding processors, 5-7

C

cache line, 7-4
common blocks
  in data scoping clauses, 1-16
compilers, accessing, -xi
compiling for OpenMP, 5-1
conditional compilation, 1-3
converting to OpenMP
  Cray-style Fortran directives, 6-3
  legacy C pragmas, 6-4
  Sun-style Fortran directives, 6-1
critical region, 1-10

D

data scoping clauses
  COPYIN, 1-17
  COPYPRIVATE, 1-18
  DEFAULT, 1-17
  FIRSTPRIVATE, 1-17
  LASTPRIVATE, 1-17
  PRIVATE, 1-16
  REDUCTION, 1-18
  SHARED, 1-16
directive
  formats, 1-2
  See pragma
directive clauses
  data scoping, 1-16
  scheduling, 1-19
directives
  ATOMIC, 1-12, 4-2
  BARRIER, 1-11
  CRITICAL, 1-11
  DO, 1-5
  FLUSH, 1-13
  for, 1-6
  MASTER, 1-11
  ORDERED, 1-14
  PARALLEL, 1-3, 1-4
  PARALLEL DO, 1-9
  parallel for, 1-9
  PARALLEL SECTIONS, 1-9
  PARALLEL WORKSHARE, 1-10
  SECTION, 1-7
  SECTIONS, 1-7
  SINGLE, 1-7
  THREADPRIVATE, 1-15
  validation (Fortran 95), 5-3
  WORKSHARE, 1-8
documentation index, -xiv
documentation, accessing, -xiv to -xv
dynamic thread adjustment, 5-5
dynamic threads, 4-1

E

environment variables, 5-5
explicitly threaded programs, 4-2

F

false sharing, 7-4

G

guided scheduling, 5-7
guided weight, 4-2

H

header files
  omp.h, 1-22
  omp_lib.h, 1-22

I

idle threads, 5-6
implementation, 4-1

M

man pages, accessing, -xi
MANPATH environment variable, setting, -xii
master thread, 1-10
memory placement optimization (MPO), 7-5
multiple page size support in Solaris, 7-6

N

nested parallelism, 2-1, 2-2, 4-2, 5-5
NUM_THREADS, 1-20
number of threads, 1-20, 4-1
  OMP_NUM_THREADS, 5-5

O

omp.h, 1-22
OMP_DESTROY_LOCK(), 1-27
OMP_DESTROY_NEST_LOCK(), 1-27
OMP_DYNAMIC, 5-5
OMP_GET_DYNAMIC(), 1-25
OMP_GET_MAX_THREADS(), 1-23
OMP_GET_NESTED(), 1-26
OMP_GET_NUM_PROCS(), 1-24
OMP_GET_NUM_THREADS(), 1-23
OMP_GET_THREAD_NUM(), 1-24
OMP_GET_WTICK(), 1-29
OMP_GET_WTIME(), 1-29
OMP_IN_PARALLEL(), 1-24
OMP_INIT_LOCK(), 1-27
OMP_INIT_NEST_LOCK(), 1-27
omp_lib.h, 1-22
OMP_NESTED, 2-2, 5-5
OMP_NUM_THREADS, 5-5
OMP_SCHEDULE, 5-5
OMP_SET_DYNAMIC(), 1-25
OMP_SET_LOCK(), 1-28
OMP_SET_NEST_LOCK(), 1-28
OMP_SET_NESTED(), 1-26
OMP_SET_NUM_THREADS(), 1-23
OMP_TEST_LOCK(), 1-28
OMP_TEST_NEST_LOCK(), 1-28
OMP_UNSET_LOCK(), 1-28
OMP_UNSET_NEST_LOCK(), 1-28
OpenMP 2.0 specifications, 1-1
ordered region, 1-14

P

parallel region, 1-3, 1-4
parallelism, nested, 2-1
PATH environment variable, setting, -xii
performance, 7-1
platforms, supported, -x
pragma
  See directive
processor binding, 5-7

R

run-time
  C/C++, 1-22
  Fortran, 1-22
runtime checking, 4-3

S

scalability, 7-4
scheduling, 4-1, 4-2
  OMP_SCHEDULE, 5-5
scheduling clauses
  SCHEDULE, 1-19, 4-1, 4-2
scoping of variables, 3-3

automatic, 3-1
autoscoping limitations, 3-8
compiler commentary, 3-4
rules, 3-2
shell prompts, -x
SLEEP, 5-6
Solaris OS tuning, 7-5
SPIN, 5-6
stack size, 5-7, 5-9
stacks, 5-9
STACKSIZE, 5-7
-stackvar, 5-9
SUNW_MP_GUIDED_WEIGHT, 4-2, 5-7
SUNW_MP_MAX_NESTED_LEVELS, 2-4, 5-7
SUNW_MP_MAX_POOL_THREADS, 2-3, 5-7
sunw_mp_misc.h, 4-4
SUNW_MP_PROCBIND, 5-6, 5-8
sunw_mp_register_warn(), 4-4
SUNW_MP_THR_IDLE, 5-6
SUNW_MP_WARN, 4-3, 5-6
supported platforms, -x
synchronization, 1-10
synchronization locks, 1-26

T

thread stack size, 5-7
timing routines, 1-29
typographic conventions, -ix

V

validation of directives (Fortran 95), 5-3

W

warning messages, 5-6
weighting factor, 4-2, 5-7
work-sharing, 1-5
combined directives, 1-8

X

-XlistMP, 5-3
-xopenmp, 5-1