Oracle® Solaris Studio 12.3: OpenMP API User's Guide

Part No: E21996
January, 2012

120118@25097

Contents

Preface 7

1 **Introducing the OpenMP API** 11
1.1 Where to Find the OpenMP Specifications 11
1.2 Special Conventions 11

2 **Compiling and Running OpenMP Programs** 13
2.1 Compiler Options 13
2.2 OpenMP Environment Variables 15
2.2.1 OpenMP Environment Variables Defaults 15
2.2.2 Oracle Solaris Studio Environment Variables 16
2.3 Processor Binding 20
2.3.1 Virtual and Logical Processor IDs 21
2.3.2 Interpreting the Value Specified for **`SUNW_MP_PROCBIND`** 21
2.3.3 *Interaction With OS Processor Sets* 22
2.4 Stacks and Stack Sizes 22
2.5 Checking and Analyzing OpenMP Programs 23

3 **Implementation-Defined Behaviors** 25
3.1 Task Scheduling Points 25
3.2 Memory Model 25
3.3 Internal Control Variables 26
3.4 Dynamic Adjustment of Threads 26
3.5 Loop Directive 27
3.6 Constructs 27
3.6.1 **`SECTIONS`** 27
3.6.2 **`SINGLE`** 27

3.6.3 **ATOMIC** 27
3.7 Routines 27
3.7.1 **omp_set_num_threads()** 27
3.7.2 **omp_set_schedule()** 27
3.7.3 **omp_set_max_active_levels()** 28
3.7.4 **omp_get_max_active_levels()** 28
3.8 Environment Variables 28
3.9 Fortran Issues 29
3.9.1 **THREADPRIVATE** Directive 29
3.9.2 **SHARED** Clause 30
3.9.3 Runtime Library Definitions 30

4 Nested Parallelism 31
4.1 Execution Model 31
4.2 Control of Nested Parallelism 32
4.2.1 **OMP_NESTED** 32
4.2.2 **OMP_THREAD_LIMIT** 33
4.2.3 **OMP_MAX_ACTIVE_LEVELS** 33
4.3 Using OpenMP Library Routines Within Nested Parallel Regions 35
4.4 Some Tips on Using Nested Parallelism 37

5 Tasking 39
5.1 Tasking Model 39
5.2 Data Environment 41
5.3 Tasking Example 42
5.4 Programming Considerations 43
5.4.1 THREADPRIVATE and Thread-Specific Information 43
5.4.2 Locks 44
5.4.3 References to Stack Data 44

6 Automatic Scoping of Variables 49
6.1 Autoscoping Data Scope Clause 50
6.1.1 **__auto** Clause 50
6.1.2 **default(__auto)** Clause 50

6.2 Scoping Rules for a Parallel Construct 50
6.2.1 Scoping Rules for Scalar Variables 51
6.2.2 Scoping Rules for Arrays 51
6.3 Scoping Rules for a `task` Construct 51
6.3.1 Scoping Rules for Scalar Variables 51
6.3.2 Scoping Rules for Arrays 52
6.4 General Comments About Autoscoping 52
6.5 Restrictions 53
6.6 Checking the Results of Autoscoping 53
6.7 Autoscoping Examples 55

7 Scope Checking 63
7.1 Using the Scope Checking Feature 63
7.2 Restrictions 66

8 Performance Considerations 67
8.1 Some General Performance Recommendations 67
8.2 False Sharing And How To Avoid It 70
8.2.1 What Is *False Sharing*? 70
8.2.2 Reducing False Sharing 71
8.3 Oracle Solaris OS Tuning Features 71

A Placement of Clauses on Directives 73

Index 75

Preface

This guide describes the specifics of the OpenMP shared memory API supported by the Oracle Solaris Studio 12.3 C, C++, and Fortran compilers.

Supported Platforms

This Oracle Solaris Studio release supports platforms that use the SPARC family of processor architectures running the Oracle Solaris operating system, as well as platforms that use the x86 family of processor architectures running Oracle Solaris or specific Linux systems.

This document uses the following terms to cite differences between x86 platforms:

- “x86” refers to the larger family of 64-bit and 32-bit x86 compatible products.
- “x64” points out specific 64-bit x86 compatible CPUs.
- “32-bit x86” points out specific 32-bit information about x86 based systems.

Information specific to Linux systems refers only to supported Linux x86 platforms, while information specific to Oracle Solaris systems refers only to supported Oracle Solaris platforms on SPARC and x86 systems.

For a complete list of supported hardware platforms and operating system releases, see the *Oracle Solaris Studio Release Notes*.

Oracle Solaris Studio Documentation

You can find complete documentation for Oracle Solaris Studio software as follows:

- Product documentation is located at the Oracle Solaris Studio documentation web site, including release notes, reference manuals, user guides, and tutorials.
- Online help for the Code Analyzer, the Performance Analyzer, the Thread Analyzer, dbxtool, DLight, and the IDE is available through the Help menu, as well as through the F1 key and Help buttons on many windows and dialog boxes, in these tools.
- Man pages for command-line tools describe a tool's command options.

Resources for Developers

Visit the Oracle Technical Network web site to find these resources for developers using Oracle Solaris Studio:

- Articles on programming techniques and best practices
- Links to complete documentation for recent releases of the software
- Information on support levels
- User discussion forums.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–1 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`

`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`**
		`Password:`

aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX system prompt and superuser prompt for shells that are included in the Oracle Solaris OS. Note that the default system prompt that is displayed in command examples varies, depending on the Oracle Solaris release.

TABLE P–2 Shell Prompts

Shell	Prompt
Bash shell, Korn shell, and Bourne shell	`$`
Bash shell, Korn shell, and Bourne shell for superuser	`#`
C shell	`machine_name%`
C shell for superuser	`machine_name#`

CHAPTER 1

Introducing the OpenMP API

The OpenMP Application Program Interface (API) is a portable, parallel programming model for shared memory multimultithreaded architectures, developed in collaboration with a number of computer vendors. The specifications were created and are published by the OpenMP Architecture Review Board.

The OpenMP API is the recommended parallel programming model for all Oracle Solaris Studio compilers on Oracle Solaris platforms.

1.1 Where to Find the OpenMP Specifications

The material presented in this manual describes issues specific to the Oracle Solaris Studio implementation of the OpenMP 3.1 API, which can be found on the official OpenMP web site at `http://www.openmp.org`.

Additional information about OpenMP including tutorials and other resources for developers can be found on the cOMPunity web site at `http://www.compunity.org`

Latest information about the Oracle Solaris Studio compiler releases and their implementation of the OpenMP API can be found on the Oracle Solaris Studio portal at, `http://www.oracle.com/technetwork/server-storage/solarisstudio`.

1.2 Special Conventions

In the tables and examples in this document, Fortran directives and source code are shown in uppercase, 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 the Oracle Solaris Studio compiler, **f95**(1).

The terms “directive” and “pragma” are used interchangeably in this manual. OpenMP API directives are significant comments inserted by the programmer to instruct the compiler to use specialized features. As comments, they are not part of the host C, C++, or Fortran language, and may be ignored or enacted depending on compiler options.

◆ ◆ ◆ CHAPTER 2

Compiling and Running OpenMP Programs

This chapter describes compiler options and runtime settings affecting programs that utilize the OpenMP API.

Note – Starting with Oracle Solaris Studio 12.3, the default number of threads used for an OpenMP program is 2 instead of 1. This number can be changed by setting the **`OMP_NUM_THREADS`** environment variable prior to running the program, or by calling the **`omp_set_num_threads()`** routine, or by using the **`num_threads`** clause on a **`PARALLEL`** directive.

2.1 Compiler Options

To enable explicit parallelization with OpenMP directives, compile your program with the **`cc`**, **`CC`**, or **`f95`** option flag **`-xopenmp`**. (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 optimization level if it is lower than **-xO3**. If you explicitly set the optimization level lower 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 and the program is parallelized accordingly but no optimization is done.
-xopenmp=stubs	This option is no longer supported. An OpenMP stubs library is provided for users' convenience. To compile an OpenMP program that calls OpenMP library routines 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:

- For best performance and functionality on Oracle Solaris platforms, make sure that the latest OpenMP runtime library, **libmtsk.so**, is installed on the running system.
- 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**.
- Specifying **-xopenmp=parallel** or **noopt** will define the **_OPENMP** preprocessor token to be YYYYMM (specifically **201107L** for C/C++ and **201107** 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. See "2.4 Stacks and Stack Sizes" on page 22.
- When compiling and linking an OpenMP program in separate steps, include **-xopenmp** on each of the compile and the link steps.
- Use the **–xvpara** option with **–xopenmp, –xopenmp=parallel**, or **–xopenmp=noopt** to display compiler warnings about potential OpenMP programming related problems and messages related to autoscoping (see Chapter 6, "Automatic Scoping of Variables") and

scope checking (see Chapter 7, “Scope Checking”). For example, warning messages will be emitted if the compiler detects the situations in the following list:

- Loops are parallelized using OpenMP directives when there are data dependencies between different loop iterations
- OpenMP data-sharing attributes-clauses are problematic. For example, declaring a variable **shared** whose accesses in an OpenMP parallel region may cause data race, or declaring a variable **private** whose value in a parallel region is used after the parallel region.

2.2 OpenMP Environment Variables

The OpenMP specification defines several environment variables that control the execution of OpenMP programs. For details, refer to the OpenMP API Version 3.1 specification at http://openmp.org. See also “3.8 Environment Variables” on page 28 for specific information on the implementation of OpenMP environment variables by the Oracle Solaris Studio compilers.

Additional environment variables defined by the Oracle Solaris Studio compilers that are not part of the OpenMP specification are summarized in “2.2.2 Oracle Solaris Studio Environment Variables” on page 16.

2.2.1 OpenMP Environment Variables Defaults

This section describes the defaults for the OpenMP environment variables.

OMP_SCHEDULE	If not set, a default value of **STATIC** is used. Example: **setenv OMP_SCHEDULE 'GUIDED,4'**
OMP_NUM_THREADS	If not set, a default of 2 is used. Example: **setenv OMP_NUM_THREADS 16**
OMP_DYNAMIC	If not set, a default value of **TRUE** is used. . Example: **setenv OMP_DYNAMIC FALSE**
OMP_NESTED	If not set, the default is **FALSE**. Example: **setenv OMP_NESTED FALSE**
OMP_STACKSIZE	The default is 4 Megabytes for 32-bit applications, and 8 Megabytes for 64-bit applications. Example: **setenv OMP_STACKSIZE 10M**

`OMP_WAIT_POLICY`	If not set, the default is **`PASSIVE`**.
`OMP_MAX_ACTIVE_LEVELS`	If not set, the default is 4.
`OMP_THREAD_LIMIT`	If not set, the default is 1024.
`OMP_PROC_BIND`	If not set, the default is **`FALSE`**.

2.2.2 Oracle Solaris Studio Environment Variables

The following additional environment variables affect the execution of OpenMP programs and are not part of the OpenMP specifications. Note that the values specified for the following environment variables are case insensitive.and can be in uppercase or lowercase.

2.2.2.1 PARALLEL

For compatibility with legacy programs, setting the **`PARALLEL`** environment variable has the same effect as setting **`OMP_NUM_THREADS`**. However, if both **`PARALLEL`** and **`OMP_NUM_THREADS`** are set, they must be set to the same value.

2.2.2.2 SUNW_MP_WARN

Controls warning messages issued by the OpenMP runtime library. If **`SUNW_MP_WARN`** is set to **`TRUE`**, the runtime library issues warning messages to **`stderr`**. In addition, the runtime library outputs the settings of all environment variables for informational purposes. If the environment variable is set to **`FALSE`**, the runtime library does not issue any warning messages or output any settings. The default is **`FALSE`**.

Example:

```
setenv SUNW_MP_WARN TRUE
```

The runtime library will also issue warning messages if the program registers a callback function to accept warning messages. A program can register a user callback function by calling the following function:

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

The address of the callback function is passed as argument to **`sunw_mp_register_warn()`**. This function returns 0 upon successfully registering the call-back function, or 1 upon failure.

If the program has registered a callback function, the runtime library will call the registered function and pass a pointer to the localized string containing the error message. The memory pointed to is no longer valid upon return from the callback function.

Note – The OpenMP runtime library has the ability to check for many common OpenMP violations, such as incorrect nesting and deadlocks. Runtime checking, however, adds overhead to the execution of the program. Set **`SUNW_MP_WARN`** to **`TRUE`** while testing or debugging a program to enable warning messages from the OpenMP runtime library to be displayed.

2.2.2.3 SUNW_MP_THR_IDLE

Controls the status of idle threads in an OpenMP program that are waiting at a barrier or waiting for new parallel regions to work on. You can set the value to be one of the following: **`SPIN`**, **`SLEEP`**, **`SLEEP(`** *time***`s)`**, **`SLEEP(`***time***`ms)`**, **`SLEEP(`** *time***`mc)`**, where *time* is an integer that specifies an amount of time, and **`s, ms`**, and **`mc`** specify the time unit (seconds, milliseconds, and microseconds, respectively).

`SPIN` specifies that an idle thread should spin while waiting at barrier or waiting for new parallel regions to work on. **`SLEEP`** without a time argument specifies that an idle thread should sleep immediately. **`SLEEP`** with a time argument specifies the amount of time a thread should spin-wait before going to sleep.

The default idle thread status is to sleep after possibly spin-waiting for some amount of time. **`SLEEP, SLEEP(0), SLEEP(0s), SLEEP(0ms)`**, and **`SLEEP(0mc)`** are all equivalent.

If both **`SUNW_MP_THR_IDLE`** and **`OMP_WAIT_POLICY`** are set, then **`OMP_WAIT_POLICY`** will be ignored.

Examples:

```
setenv SUNW_MP_THR_IDLE SPIN
setenv SUNW_MP_THR_IDLE SLEEP
setenv SUNW_MP_THR_IDLE SLEEP(2s)
setenv SUNW_MP_THR_IDLE SLEEP(20ms)
setenv SUNW_MP_THR_IDLE SLEEP(150mc)
```

2.2.2.4 SUNW_MP_PROCBIND

This environment variable works on both Oracle Solaris and Linux systems. The **`SUNW_MP_PROCBIND`** environment variable can be used to bind threads of an OpenMP program to virtual processors on the running system. Performance can be enhanced with processor binding, but performance degradation will occur if multiple threads are bound to the same virtual processor. If both **`SUNW_MP_PROCBIND`** and **`OMP_PROC_BIND`** are set, they must be set to the same value. See "2.3 Processor Binding" on page 20 for details.

2.2.2.5 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 to work on parallel regions. The pool does not contain the initial (main) 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. If not specified, the default is 1023. See "4.2 Control of Nested Parallelism" on page 32 for details.

Note that **SUNW_MP_MAX_POOL_THREADS** specifies the maximum number of *non-user* OpenMP threads to use for the whole program, while **OMP_THREAD_LIMIT** specifies the maximum number of *user and non-user* OpenMP threads for the whole program. If both **SUNW_MP_MAX_POOL_THREADS** and **OMP_THREAD_LIMIT** are set, they must have consistent values such that **OMP_THREAD_LIMIT** is set to one more than the value of **SUNW_MP_MAX_POOL_THREADS**.

2.2.2.6 SUNW_MP_MAX_NESTED_LEVELS

Sets the the maximum number of nested active parallel regions. A parallel region is active if it is executed by a team consisting of more than one thread. If **SUNW_MP_MAX_NESTED_LEVELS** is not specified, the default is 4. See "4.2 Control of Nested Parallelism" on page 32 for details.

Note that if both **SUNW_MP_MAX_NESTED_LEVELS** and **OMP_MAX_ACTIVE_LEVELS** are set, they must be set to the same value.

2.2.2.7 STACKSIZE

Sets the stack size for each thread. The value is in Kilobytes. The default thread stack sizes are 4 Megabytes on 32-bit SPARC V8 and x86 platforms and 8 Megabytes on 64-bit SPARC V9 and x86 platforms.

The **STACKSIZE** environment variable accepts numerical values with a suffix of either **B, K, M**, or **G** for Bytes, Kilobytes, Megabytes, or Gigabytes respectively. If no suffix is specified, the default is Kilobytes.

Examples:

```
setenv STACKSIZE 8192  // sets the thread stack size to 8 Megabytes
setenv STACKSIZE 16M  // sets the thread stack size to 16 Megabytes
```

Note that if both **STACKSIZE** and **OMP_STACKSIZE** are set, they must be set to the same value. If the values are not the same, a runtime error occurs.

2.2.2.8 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.

2.2.2.9 `SUNW_MP_WAIT_POLICY`

Allows fine-grain control of the behavior of threads in the program that are waiting for work (idle), waiting at a barrier, or waiting for tasks to complete. The behavior for each of the above types of wait has three possibilities: spin for awhile, yield the CPU for awhile, and sleep until awakened.

The syntax is (shown using csh):

`setenv SUNW_MP_WAIT_POLICY IDLE=`*val***`:BARRIER=`***val***`:TASKWAIT=`***val*

`IDLE=`*val*, **`BARRIER=`***val*, and **`TASKWAIT=`***val* are optional keywords that specify the type of wait being controlled.

Each of these keywords has a *val* setting that describes the wait behavior, **`SPIN`**, **`YIELD`**, or **`SLEEP`**.

`SPIN(`*time***`)`** specifies how long a thread should spin before yielding the CPU. *time* can be in seconds, milliseconds, or microseconds (denoted by **`s`**, **`ms`**, and **`mc`**, respectively). If no time unit is specified, then seconds is assumed. **`SPIN`** with no time parameter means that the thread should continuously spin while waiting.

`YIELD(`*number***`)`** specifies the number of times a thread should yield the CPU before sleeping. After each yield of the CPU, a thread will run again when the operating system schedules it to run. YIELD with no *number* parameter means the thread should continuously yield while waiting.

`SLEEP` specifies that a thread should immediately go to sleep.

Note that **`SPIN`**, **`SLEEP`**, and **`YIELD`** settings for a particular type of wait can be specified in any order. The settings are separated by comma. **`"SPIN(0),YIELD(0)"`** is the same as **`SLEEP`** or sleep immediately. When processing the settings for **`IDLE`**, **`BARRIER`**, and **`TASKWAIT`**, the “left-most wins” rule is used.

If both **`SUNW_MP_WAIT_POLICY`** and **`OMP_WAIT_POLICY`** are set, **`OMP_WAIT_POLICY`** will be ignored.

Examples:

```
% setenv SUNW_MP_WAIT_POLICY “BARRIER=SPIN”
```

A thread waiting at a barrier spins until all threads in the team have reached the barrier.

```
% setenv SUNW_MP_WAIT_POLICY “IDLE=SPIN(10ms),YIELD(5)”
```

A thread waiting for work (idle) spins for 10 milliseconds, then yields the CPU 5 times before going to sleep.

```
% setenv SUNW_MP_WAIT_POLICY \
“IDLE=SPIN(10ms),YIELD(2):BARRIER=SLEEP:TASKWAIT=YIELD(10)”
```

A thread waiting for work (idle) spins for 10 milliseconds, then yields the CPU 2 times before going to sleep; a thread waiting at a barrier goes to sleep immediately; a thread waiting at a taskwait yields the CPU 10 times before going to sleep.

2.3 Processor Binding

With processor binding, the programmer instructs the operating system that a thread in the program should run on the same processor throughout the execution of the program.

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

From the hardware point of view, a computer system is composed of one or more physical processors. From the operating system point of view, each of these physical processors maps to one or more virtual processors onto which threads in a program can be run. If *n* virtual processors are available, then *n* threads can be scheduled to run at the same time. Depending on the system, a virtual processor may be a processor, a core, a hardware thread, and so on.

For example, the SPARC T3 physical processor has eight cores, and each core can run eight simultaneous processing threads. From the Oracle Solaris system point of view, there are 64 virtual processors onto which threads can be scheduled to run. On Oracle Solaris platforms, the number of virtual processors can be determined by using the **`psrinfo`**(1M) command. On Linux systems, the text file `/proc/cpuinfo` provides information about available processors.

When the operating system binds threads to processors, they are in effect bound to specific *virtual* processors, not *physical* processors.

Set the **`SUNW_MP_PROCBIND`** environment variable to bind threads in an OpenMP program. The value specified for **`SUNW_MP_PROCBIND`** can be one of the following:

- The string "**`TRUE`**" or "**`FALSE`**" or "**`COMPACT`**" or "**`SCATTER`**" (or lowercase "**`true`**" or "**`false`**" or "**`compact`**" or "**`scatter`**"). For example, **`% setenv SUNW_MP_PROCBIND "TRUE"`**
- 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 "2 2 4 6"`** will bind two threads to processor 2, one will be bound to processor 4, and one will be bound to processor 6, if four threads are used.
- Two non-negative integers, *n1* and *n2*, separated by a minus sign ("-"). *n1* must be less than or equal to *n2*.
  For example, **`% setenv SUNW_MP_PROCBIND "0-6"`**

Note that the non-negative integers referred to above denote logical identifiers (IDs). Logical IDs may be different from *virtual* processor IDs. The difference will be explained in the next section.

2.3.1 Virtual and Logical Processor IDs

Each virtual processor in a system has a unique processor ID. The Oracle Solaris **psrinfo**(1M) command displays information about the virtual processors in a system, including their virtual processor IDs. The **prtdiag**(1M) command displays system configuration and diagnostic information.

You can use **psrinfo -pv** to list all physical processors in the system and the virtual processors that are associated with each physical processor.

Virtual processor IDs could be sequential, but gaps in the IDs can also occur. For example, on a Sun Fire 4810 with 8 UltraSPARC IV processors (16 cores), the virtual processor IDs might be: 0, 1, 2, 3, 8, 9, 10, 11, 512, 513, 514, 515, 520, 521, 522, 523.

Logical processor IDs on the other hand are consecutive integers that start with 0. If the number of virtual processors available in the system is *n*, then their logical IDs are 0, 1, ..., *n*-1, in the order presented by **psrinfo**(1M).

For an interpretation of the values accepted by **SUNW_MP_PROCBIND**, see "2.3.2 Interpreting the Value Specified for **SUNW_MP_PROCBIND**" on page 21.

2.3.2 Interpreting the Value Specified for SUNW_MP_PROCBIND

If the value specified for **SUNW_MP_PROCBIND** is **FALSE**, the threads will not be bound to any processors. This is the default setting.

If the value specified for **SUNW_MP_PROCBIND** is **TRUE**, then the threads will be bound to virtual processors in a round-robin fashion. The starting processor for the binding is determined by the runtime library with the goal of achieving best performance.

If the value specified for **SUNW_MP_PROCBIND** is **COMPACT**, then the threads will be bound to virtual processors that are as close together as possible on the system. **COMPACT** allows threads to share data caches and thus improve data locality.

If the value specified for **SUNW_MP_PROCBIND** is **SCATTER**, then the threads will be bound to virtual processors that are far apart. This will allow higher memory bandwidth for each of the threads. **SCATTER** is the opposite of **COMPACT**.

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

If the value specified for **SUNW_MP_PROCBIND** is a list of two or more non-negative integers, then threads will be bound in a round-robin fashion to virtual processors with the specified logical IDs. Processors with logical IDs other than those specified will not be used.

If the value specified for **SUNW_MP_PROCBIND** is two non-negative integers separated by a minus ("-"), then threads will be bound in a round-robin fashion to virtual processors in the range that begins with the first logical ID and ends with the second logical ID. Processors with logical IDs other than those included in the range will not 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 ID is given, then an error message will be emitted and execution of the program will terminate.

If the number of threads in an OpenMP program is is greater than the number of virtual processors available, some virtual processors will have more than one thread bound to them. This may negatively impact performance.

2.3.3 Interaction With OS Processor Sets

A processor set can be specified using the **psrset** utility on Oracle Solaris platforms, or the **taskset** command on Linux platforms. **SUNW_MP_PROCBIND** does not take processor sets into account. If you use processor sets, then it is your responsibility to ensure that the setting of **SUNW_MP_PROCBIND** is consistent with the processor set used. Otherwise, the setting of **SUNW_MP_PROCBIND** will override the processor set setting on Linux systems, while on Oracle Solaris systems an error message will be issued.

2.4 Stacks and Stack Sizes

The executing program maintains a main stack for the initial (or main) 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. Stack overflow might occur if the size of a thread's stack is too small, causing silent data corruption or a segmentation fault.

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 loops with calls. (See the *Fortran User's Guide* for a discussion of the **-stackvar** flag.) However, this usage could lead to stack overflow if not enough memory is allocated for the stack.

Use the **limit** C-shell command or the **ulimit** Bourne or Korn shell command to display or set the stack size for the initial (or main) thread. In general, the default stack size for the initial thread is 8 Megabytes.

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 slave thread stack is set with the **OMP_STACKSIZE** environment variable.

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

demo$ OMP_STACKSIZE=16384    <-Set thread stack size to 16 Mb (Bourne/Korn shell)
demo$ export OMP_STACKSIZE
```

To detect stack overflows, compile your Fortran, C, or C++ programs with the **-xcheck=stkovf** compiler option to force a segmentation fault on stack overflow, thereby stopping the program before any data corruption can occur.

2.5 Checking and Analyzing OpenMP Programs

Oracle Solaris Studio provides several tools to help debug and analyze OpenMP programs.

- **dbx** is an interactive debugging tool that provides facilities to run a program in a controlled fashion and inspect the state of a stopped program. Refer to **dbx**(1). for more information
- The Thread Analyzer is a tool for detecting data races and deadlocks in a multithreaded program. Refer to the Thread Analyzer manual and the **tha**(1) and **libtha**(3) man pages for details.
- The Performance Analyzer analyzes the performance of an OpenMP program. Refer to the Performance Analyzer manual or the **collect**(1) and **analyzer**(1) man pages for details.

CHAPTER 3

Implementation-Defined Behaviors

This chapter documents how certain OpenMP features behave when the program is compiled using Oracle Solaris Studio compilers.(See Appendix E of the OpenMP 3.1 API Specification.)

3.1 Task Scheduling Points

Task scheduling points in untied task regions occur at the same points as in tied task regions. Therefore, within untied task regions, the OpenMP specification defines the following task scheduling:

- Encountered task constructs
- Encountered taskwait constructs
- Encountered taskyield constructs
- Encountered barrier directives
- Implicit barrier regions
- At the end of the untied task region

3.2 Memory Model

Memory accesses by multiple threads to the same variable without synchronization are not necessarily atomic with respect to each other. Several implementation-dependent and application-dependent factors affect whether accesses are atomic. Some variables might be larger than the largest atomic memory operation on the target platform. Some variables might be misaligned or of unknown alignment and the compiler or the runtime system might need to use multiple loads/stores to access the variable. Sometimes there are faster code sequences that use more loads/stores.

3.3 Internal Control Variables

The following internal control variables are defined by the implementation:

- *nthreads-var*: Controls the number of threads requested for encountered parallel regions. The initial value of *nthreads-var* is 1.
- *dyn-var*: Controls whether dynamic adjustment of the number of threads is enabled for encountered parallel regions. The initial value of *dyn-var* is `TRUE` (that is, dynamic adjustment is enabled).
- *run-sched-var*: Controls the schedule that the runtime schedule clause uses for loop regions. The initial value of *run-sched-var* is `static` with no chunk size.
- *def-sched-var*: Controls the implementation defined default scheduling of loop regions. The initial value of *def-sched-var* is `static` with no chunk size.
- *bind-var*: Controls the binding of threads to processors. The initial value of *bind-var* is `FALSE`.
- *stacksize-var*: Controls the stack size for threads that the OpenMP implementation creates. The initial value of *stacksize-var* is 4 Megabytes for 32-bit applications and 8 Megabytes for 64-bit applications.
- *wait-policy-var*: Controls the desired behavior of waiting threads. The initial value of *wait-policy-var* is `PASSIVE`.
- *thread-limit-var*: Controls the maximum number of threads participating in the OpenMP program. The initial value of *thread-limit-var* is 1024.
- *max-active-levels-var*: Controls the maximum number of nested active parallel regions. The initial value of *max-active-levels-var* is 4.

3.4 Dynamic Adjustment of Threads

The implementation provides the ability to dynamically adjust the number of threads. Dynamic adjustment is enabled by default. Set the **`OMP_DYNAMIC`** environment variable to **`FALSE`** or call the **`omp_set_dynamic()`** routine with the appropriate argument to disable dynamic adjustment.

When a thread encounters a parallel construct, the number of threads delivered by this implementation is determined according to Algorithm 2.1 in the OpenMP 3.1 Specification. In exceptional situations, such as a lack of system resources, the number of threads supplied will be less than described in Algorithm 2.1. In these situations, if **`SUNW_MP_WARN`** is set to **`TRUE`** or a callback function is registered through a call to **`sunw_mp_register_warn()`**, a warning message is issued.

3.5 Loop Directive

The integer type used to compute the iteration count of a collapsed loop is **`long`**.

The effect of the **`schedule(runtime)`** clause when the *run-sched-var* internal control variable is set to *auto* is `static` with no chunk size.

3.6 Constructs

3.6.1 SECTIONS

The structured blocks in the sections construct are assigned to the threads in the team in a static with no chunk size fashion, so that each thread gets an approximately equal number of consecutive structured blocks.

3.6.2 SINGLE

The first thread to encounter the **`single`** construct will execute the construct.

3.6.3 ATOMIC

The implementation replaces all **`atomic`** directives by enclosing the target statement with a special, named **`critical`** construct. This operation enforces exclusive access between all atomic regions in the program, regardless of whether these regions update the same or different storage locations.

3.7 Routines

3.7.1 omp_set_num_threads()

If the argument to **`omp_set_num_threads()`** is not a positive integer, then the call is ignored. A warning message is issued if **`SUNW_MP_WARN`** is set to **`TRUE`** or a callback function is registered by a call to **`sunw_mp_register_warn()`**.

3.7.2 omp_set_schedule()

The behavior for the Oracle Solaris Studio specific **`sunw_mp_sched_reserved`** schedule is the same as `static` with no chunk size.

3.7.3 `omp_set_max_active_levels()`

If **`omp_set_max_active_levels()`** is called from within an active parallel region, then the call is ignored. A warning message is issued if **`SUNW_MP_WARN`** is set to `TRUE` or a callback function is registered by a call to **`sunw_mp_register_warn()`**.

If the argument to **`omp_set_max_active_levels()`** is not a non-negative integer, then the call is ignored. A warning message is issued if **`SUNW_MP_WARN`** is set to `TRUE` or a callback function is registered by a call to **`sunw_mp_register_warn()`**.

3.7.4 `omp_get_max_active_levels()`

`omp_get_max_active_levels()` can be called from anywhere in the program. The call returns the value of the *max-active-levels-var* internal control variable.

3.8 Environment Variables

Variable Name	Implementation
`OMP_SCHEDULE`	If the schedule type specified for the **`OMP_SCHEDULE`** is not one of the valid types (**`static`**, **`dynamic`**, **`guided`**, or **`auto`**), then the environment variable is ignored, and the default schedule (**`static`** with no chunk size) is used. A warning message is issued if **`SUNW_MP_WARN`** is set to `TRUE` or a callback function is registered by a call to **`sunw_mp_register_warn()`**. If the schedule type specified for the **`OMP_SCHEDULE`** environment variable is **`static`**, **`dynamic`**, or **`guided`** but the chunk-specified size is a negative integer, then the chunk size used is as follows: For **`static`**, there is no chunk size . For **`dynamic`** and **`guided`**, the chunk size is 1. A warning message is issued if **`SUNW_MP_WARN`** is set to `TRUE` or a callback function is registered by a call to **`sunw_mp_register_warn()`**.
`OMP_NUM_THREADS`	If the value of the variable is not a positive integer, then the environment variable is ignored and a warning message issued if **`SUNW_MP_WARN`** is set to `TRUE` or a callback function is registered by a call to **`sunw_mp_register_warn()`**. If the value of the variable is greater than the number of threads the implementation can support, the following actions are taken: ■ If *dynamic* adjustment of the number of threads is enabled, then the number of threads will be reduced and a warning message will be issued if **`SUNW_MP_WARN`** is set to `TRUE` or a callback function is registered by a call to **`sunw_mp_register_warn()`**. ■ If dynamic adjustment of the number of threads is disabled, then an error message will be issued and the program will stop.

Variable Name	Implementation
`OMP_PROC_BIND`	If the value specified for **`OMP_PROC_BIND`** is neither **`TRUE`** nor **`FALSE`**, then an error message will be issued and the program will stop.
`OMP_DYNAMIC`	If the value specified for **`OMP_DYNAMIC`** is neither `TRUE` nor `FALSE`, then the value will be ignored, and the default value `TRUE` will be used. A warning message will be issued if **`SUNW_MP_WARN`** is set to `TRUE` or a callback function is registered by a call to **`sunw_mp_register_warn()`**.
`OMP_NESTED`	If the value specified for **`OMP_NESTED`** is neither `TRUE` nor `FALSE`, then the value will be ignored and the default value `FALSE` will be used. A warning message will be issued if **`SUNW_MP_WARN`** is set to `TRUE` or a callback function is registered by a call to **`sunw_mp_register_warn()`**.
`OMP_STACKSIZE`	If the value given for **`OMP_STACKSIZE`** does not conform to the specified format, then the value will be ignored and the default value (4 Megabytes for 32-bit applications, and 8 Megabytes for 64-bit applications) is used. A warning message is issued if **`SUNW_MP_WARN`** is set to `TRUE` or a callback function is registered by a call to **`sunw_mp_register_warn()`**.
`OMP_WAIT_POLICY`	The `ACTIVE` behavior for a thread is *spin*. The `PASSIVE` behavior for a thread is *sleep*, after possibly spinning for a while.
`OMP_MAX_ACTIVE_LEVELS`	If the value specified for **`OMP_MAX_ACTIVE_LEVELS`** is not a nonnegative integer, then the value will be ignored and the default value (4) will be used. A warning message will be issued if **`SUNW_MP_WARN`** is set to `TRUE` or a callback function is registered by a call to **`sunw_mp_register_warn()`**.
`OMP_THREAD_LIMIT`	If the value specified for **`OMP_THREAD_LIMIT`** is not a positive integer, then the value will be ignored and the default value (1024) will be used. A warning message will be issued if **`SUNW_MP_WARN`** is set to `TRUE` or a callback function is registered by a call to **`sunw_mp_register_warn()`**.

3.9 Fortran Issues

The following issues apply to Fortran only.

3.9.1 `THREADPRIVATE` Directive

If the conditions for values of data in the thread-private objects of threads (other than the initial thread) to persist between two consecutive active parallel regions do not all hold, then the allocation status of an allocatable array in the second region might be not currently allocated.

3.9.2 `SHARED` Clause

Passing a shared variable to a non-intrinsic procedure could result in the value of the shared variable being copied into temporary storage before the procedure reference, and back out of the temporary storage into the actual argument storage after the procedure reference. This copying into and out of temporary storage can occur only if the conditions found in the OpenMP 3.1 Specification **`shared`** clause section hold, namely:

- The actual argument is one of the following:
  - A shared variable
  - A subobject of a shared variable
  - An object associated with a shared variable
  - An object associated with a subobject of a shared variable
- The actual argument is also one of the following:
  - An array section
  - An array section with a vector subscript
  - An assumed-shape array
  - A pointer array
- The associated dummy argument for this actual argument is an explicit-shape array or an assumed-size array.

3.9.3 Runtime Library Definitions

Both the include file **`omp_lib.h`** and the module file **`omp_lib`** are provided in the implementation.

On Oracle Solaris platforms, the OpenMP runtime library routines that take an argument are extended with a generic interface so arguments of different Fortran **`KIND`** types can be accommodated.

CHAPTER 4

Nested Parallelism

This chapter discusses the features of OpenMP nested parallelism.

4.1 Execution Model

OpenMP uses the 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 an 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 the pool does not contain a sufficient number of idle threads. When the team finishes executing the parallel region, the slave threads return to the pool.

4.2 Control of Nested Parallelism

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

4.2.1 OMP_NESTED

Nested parallelism can be enabled or disabled by setting the **OMP_NESTED** environment variable or calling **omp_set_nested()**.

The following example has three levels of nested parallel constructs.

EXAMPLE 4–1 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 (sorted) 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
```

The following example runs 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
```

4.2.2 OMP_THREAD_LIMIT

The OpenMP runtime library maintains a pool of threads that can be used as slave threads in parallel regions. The setting of the **OMP_THREAD_LIMIT** environment variable controls the number of threads in the pool. By default, the number of threads in the pool is at most 1023.

The thread pool consists of only non-user threads that the runtime library creates. The pool does not include the initial thread or any thread created explicitly by the user's program.

If **OMP_THREAD_LIMIT** is set to 1 (or **SUNW_MP_MAX_POOL_THREADS** is set to 0), then 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 slave threads if the pool does not contain sufficient threads. The code is the same as Example 4–1. The number of threads needed for all the parallel regions to be active at the same time is 8. Therefore, the pool needs to contain at least 7 threads. If **OMP_THREAD_LIMIT** is set to 6 (or **SUNW_MP_MAX_POOL_THREADS** to 5), then the pool contains at most 5 slave threads. This implies that two of the four innermost parallel regions might not be able to get all the slave threads requested. The following example shows one possible result:

```
% setenv OMP_NESTED TRUE
% OMP_THREAD_LIMIT 6
% 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
```

4.2.3 OMP_MAX_ACTIVE_LEVELS

The environment variable **OMP_MAX_ACTIVE_LEVELS** controls the maximum number of nested active parallel regions. A parallel region is active if it is executed by a team consisting of more than one thread. The default maximum number of nested active parallel regions is 4.

Note that setting this environment variable is not sufficient to enable nested parallelism. This environment variable simply controls the the maximum number of nested active parallel regions; it does not enable nested parallelism. To enable nested parallelism, **`OMP_NESTED`** must be set to **`TRUE`**, or **`omp_set_nested()`** must be called with an argument that evaluates to *true*.

The following code will create 4 levels of nested parallel regions. If **`OMP_MAX_ACTIVE_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);
}
```

The following example shows the possible results from compiling and running this program with a maximum nesting level of 4. (Actual results would depend on how the OS schedules threads.)

```
% setenv OMP_MAX_ACTIVE_LEVELS 4
% a.out |sort
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
```

The following example shows a possible result running with the nesting level set at 2:

```
% setenv OMP_MAX_ACTIVE_LEVELS 2
% a.out |sort
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
```

Again, these examples only show some *possible* results. Actual results would depend on how the OS schedules threads.

4.3 Using OpenMP Library Routines 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 future parallel regions at the same or inner nesting levels encountered by the calling thread only. They do not affect parallel regions encountered by other threads.

The **`get`** calls return the values set by the calling thread. When a thread becomes the master of a team executing a parallel region, all other members of the team inherit the values of the master thread. When the master thread exits a nested parallel region and continues executing the enclosing parallel region, the values for that thread revert to their values in the enclosing parallel region just before executing the nested parallel region.

EXAMPLE 4–2 Calls to OpenMP Routines Within Parallel Regions

```
#include <stdio.h>
#include <omp.h>
int main()
```

EXAMPLE 4–2 Calls to OpenMP Routines Within Parallel Regions *(Continued)*

```
{
    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");
        }
    }
```

EXAMPLE 4–2 Calls to OpenMP Routines Within Parallel Regions *(Continued)*

```
    return(0);
}
```

The following example shows a possible result from compiling and running this program:

```
% 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.
```

4.4 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. Also, suppose 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 **`OMP_THREAD_LIMIT`** and **`OMP_MAX_ACTIVE_LEVELS`** appropriately to limit the number of threads in use and prevent runaway oversubscription.
- Creating nested parallel regions adds overhead. If the outer level has enough paralellism and the load is balanced, using all the threads at the outer level of the computation will be more efficient than creating nested parallel 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 four 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. In general, using all four threads for the outer level could yield better performance than using two threads for the outer parallel region and using the other two threads as slave threads for the inner parallel regions.

CHAPTER 5

Tasking

This chapter describes the OpenMP Tasking Model.

5.1 Tasking Model

Tasking facilitates the parallelization of applications where units of work are generated dynamically, as in recursive structures or *while* loops.

In OpenMP, an *explicit* task is specified using the **`task`** directive. The **`task`** directive defines the code associated with the task and its data environment. The task construct can be placed anywhere in the program: whenever a thread encounters a task construct, a new task is generated.

When a thread encounters a task construct, it may choose to execute the task immediately or defer its execution until a later time. If task execution is deferred, then the task in placed in a conceptual pool of tasks that is associated with the current parallel region. The threads in the current team will take tasks out of the pool and execute them until the pool is empty. A thread that executes a task might be different from the thread that originally encountered it.

The code associated with a task construct will be executed only once. A task is *tied* if the code is executed by the same thread from beginning to end. A task is *untied* if the code can be executed by more than one thread, so that different threads execute different parts of the code. By default, tasks are *tied*, and a task can be specified to be *untied* by using the **`untied`** clause with the **`task`** directive.

Threads are allowed to suspend execution of a task region at a task scheduling point in order to execute a different task. If the suspended task is tied, then the same thread later resumes execution of the suspended task. If the suspended task is untied, then any thread in the current team may resume the task execution.

The OpenMP specification defines the following task scheduling points for *tied* tasks:

- The point of encountering a task construct

- The point of encountering a taskwait construct
- The point of encountering a taskyield construct
- The point of encountering an implicit or explicit barrier
- The completion point of the task

As implemented in the Oracle Solaris Studio compilers, these scheduling points are also the task scheduling points for *untied* tasks.

In addition to explicit tasks specified using the task directive, the OpenMP specification introduces the notion of *implicit* tasks. An implicit task is a task generated by the implicit parallel region, or generated when a parallel construct is encountered during execution. The code for each implicit task is the code inside the **`parallel`** construct. Each implicit task is assigned to a different thread in the team and is tied; that is, the implicit task is always executed from beginning to end by the thread to which it is initially assigned.

All implicit tasks generated when a **`parallel`** construct is encountered are guaranteed to be complete when the master thread exits the implicit barrier at the end of the parallel region. On the other hand, all explicit tasks generated within a parallel region are guaranteed to be complete on exit from the next implicit or explicit barrier within the parallel region.

The OpenMP 3.1 specifiction defines various types of tasks that the programmer may choose to reduce the overhead of tasking.

An *undeferred* task is a task for which execution is not deferred with respect to its generating task region. That is, its generating task region is suspended until execution of the undeferred task is completed. An example of an undeferred task is a task with an **`if`** clause expression that evaluates to **`false`**. In this case, an undeferred task is generated, and the encountering thread must suspend the current task region, for which execution cannot be resumed until the task with the **`if`** clause is completed.

An *included* task is a task for which execution is sequentially included in the generating task region. That is, it is undeferred and executed immediately by the encountering thread.An example of an included task is a task that is a descendent of a *final* task (described below).

The distinction between an included task and an undeferred task is subtle. In the case of an undeferred task, the generating task region is suspended until execution of the undeferred task is completed, but the undeferred task may not be executed immediately as soon as it is encountered. The undeferred task may be placed in the conceptual pool and executed at a later time by the encountering thread or by some other thread; in the meantime, the generating task is suspended. Once the execution of the undeferred task is completed, the generating task can resume.

In the case of included task, the included task is executed immediately by the encountering thread as soon as it is encountered. The task is not placed in the pool to be executed at a later time. The generating task is suspended until the execution of the included task is completed. Once the execution of the included task is completed, the generating task can resume.

A *merged* task is a task whose data environment is the same as that of its generating task region. When a **`mergeable`** clause is present on a **`task`** construct, and the generated task is an undeferred task or an included task, then the implementation may choose to generate a merged task instead. If a merged task is generated, then the behavior is as though there was no task directive at all

A *final* task is a task that forces all of its child tasks to become final and included tasks. When a **`final`** clause is present on a **`task`** construct and the **`final`** clause expression evaluates to **`TRUE`**, the generated task will be a *final* task. All descendents of that final task will be both *final* and *included* tasks.

5.2 Data Environment

The **`task`** directive takes the following data attribute clauses that define the data environment of the task:

- **`default (private`** | **`firstprivate`** | **`shared`** | **`none`**)
- **`private (`***list***`)`**
- **`firstprivate (`***list***`)`**
- **`shared (`***list***`)`**

All references within a task to a variable listed in the **`shared`** clause refer to the variable with that same name known immediately prior to the **`task`** directive.

For each **`private`** and **`firstprivate`** variable, new storage is created and all references to the original variable in the lexical extent of the **`task`** construct are replaced by references to the new storage. A **`firstprivate`** variable is initialized with the value of the original variable at the moment the task is encountered.

The OpenMP 3.0 specification version 3.0 (in section 2.9.1) describes how the data-sharing attributes of variables referenced in parallel, task, and worksharing regions are determined.

The data-sharing attributes of variables referenced in a construct may be one of the following: *predetermined*, *explicitly determined*, or *implicitly determined*. Variables with explicitly determined data-sharing attributes are those that are referenced in a given construct and are listed in a data-sharing attribute clause on the construct. Variables with implicitly determined data-sharing attributes are those that are referenced in a given construct, do not have predetermined data-sharing attributes, and are not listed in a data-sharing attribute clause on the construct.

The rules for how the data-sharing attributes of variables are implicitly determined might not always be obvious. To avoid any surprises be sure to explicitly scope all variables that are referenced in a task construct using the data sharing attribute clauses rather than relying on the OpenMP implicit scoping rules.

5.3 Tasking Example

The following C/C++ program illustrates how the OpenMP **`task`** and **`taskwait`** directives can be used to compute Fibonacci numbers recursively.

In the example, the **`parallel`** directive denotes a parallel region which will be executed by four threads. In the parallel construct, the **`single`** directive is used to indicate that only one of the threads will execute the **`print`** statement that calls `fib(n)`.

The call to `fib(n)` generates two tasks, indicated by the **`task`** directive. One of the tasks computes `fib(n-1)` and the other computes `fib(n-2)` The return values are added together to produce the value returned by `fib(n)`. Each of the calls to `fib(n-1)` and `fib(n-2)` will in turn generate two tasks. Tasks will be recursively generated until the argument passed to `fib()` is less than 2.

Note the **`final`** clause on each of the **`task`** directives. If the **`final`** clause expression (**`n <= THRESHOLD`**) evaluates to true, then the generated task will be a **`final`** task. All **`task`** constructs encountered during execution of a final task will generate included (and final) tasks. So included tasks will be generated when `fib` is called with the argument *n = 9, 8, ..., 2*. These included tasks will be executed immediately by the encountering threads, thus reducing the overhead of placing tasks in the conceptual pool.

The **`taskwait`** directive ensures that the two tasks generated in an invocation of `fib()` are completed (that is, the tasks compute `i` and `j`) before that invocation of `fib()` returns.

Note that although only one thread executes the **`single`** directive and hence the call to `fib(n)`, all four threads will participate in executing the tasks generated.

The following example is compiled using the Oracle Solaris Studio C++ compiler.

EXAMPLE 5–1 Tasking Example: Computing Fibonacci Numbers

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

#define THRESHOLD 5

int fib(int n)
{
  int i, j;

  if (n<2)
    return n;

  #pragma omp task shared(i) firstprivate(n) final(n <= THRESHOLD)
  i=fib(n-1);

  #pragma omp task shared(j) firstprivate(n) final(n <= THRESHOLD)
  j=fib(n-2);

  #pragma omp taskwait
```

EXAMPLE 5–1 Tasking Example: Computing Fibonacci Numbers *(Continued)*

```
  return i+j;
}


int main()
{
  int n = 30;
  omp_set_dynamic(0);
  omp_set_num_threads(4);

  #pragma omp parallel shared(n)
  {
     #pragma omp single
     printf ("fib(%d) = %d\n", n, fib(n));
  }
}

% CC -xopenmp -xO3 task_example.cc

% a.out
fib(30) = 832040
```

5.4 Programming Considerations

Tasking introduces a layer of complexity to an OpenMP program. The programmer needs to pay special attention to how a program with tasks works. This section discusses some programming issues to consider.

5.4.1 THREADPRIVATE and Thread-Specific Information

When a thread encounters a task scheduling point, the implementation may choose to suspend the current task and schedule the thread to work on another task. This behavior implies that the value of a **`threadprivate`** variable, or other thread-specific information such as the thread number, may change across a task scheduling point.

If the suspended task is *tied*, then the thread that resumes executing the task will be the same thread that suspended it. Therefore, the thread number will remain the same after the task is resumed. However, the value of a **`threadprivate`** variable may change because the thread might have been scheduled to work on another task that modified the **`threadprivate`** variable before resuming the suspended task.

If the suspended task is *untied*, then the thread that resumes executing the task may be different from the thread that suspended it. Therefore, both the thread number and the value of a **`threadprivate`** variable before and after the task scheduling point might be different.

5.4.2 Locks

OpenMP specifies that locks are no longer owned by threads, but by tasks. Once a lock is acquired, the current task owns it, and the same task must release it before task completion.

The **`critical`** construct, on the other hand, remains as a *thread-based mutual exclusion mechanism*.

The change in lock ownership requires extra care when using locks. The following example (which appears in Appendix A of the OpenMP 3.1 Specification) is conforming in OpenMP 2.5 because the thread that releases the lock `lck` in the parallel region is the same thread that acquired the lock in the sequential part of the program. The master thread of a parallel region and the initial thread are the same. However, it is not conforming in OpenMP 3.1 because the task region that releases the lock `lck` is different from the task region that acquires the lock.

EXAMPLE 5–2 Example Using Locks: Non-Conforming in OpenMP 3.0

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

int main()
{
  int x;
  omp_lock_t lck;

  omp_init_lock (&lck);
  omp_set_lock (&lck);
  x = 0;

  #pragma omp parallel shared (x)
  {
    #pragma omp master
    {
      x = x + 1;
      omp_unset_lock (&lck);
    }
  }
  omp_destroy_lock (&lck);
}
```

5.4.3 References to Stack Data

A task is likely to have references to data on the stack of the routine where the task construct appears. Because the execution of a task may be deferred until the next implicit or explicit barrier, a given task could execute after the stack of the routine where it appears has already been popped and the stack data overwritten, thereby destroying the stack data listed as shared by the task.

The programmer is responsible for inserting the needed synchronizations to ensure that variables are still on the stack when the task references them, as illustrated in the following two examples.

In the first example, `i` is specified to be **`shared`** in the **`task`** construct, and the task accesses the copy of `i` that is allocated on the stack of `work()`.

Task execution may be deferred, so tasks are executed at the implicit barrier at the end of the parallel region in `main()` after the `work()` routine has already returned. So when a task references `i`, it accesses some undetermined value that happens to be on the stack at that time.

For correct results, the programmer needs to make sure that `work()` does not exit before the tasks have completed. This is done by inserting a **`taskwait`** directive after the **`task`** construct. Alternatively, `i` can be specified to be **`firstprivate`** in the **`task`** construct, instead of **`shared`**.

EXAMPLE 5–3 Stack Data: First Example– Incorrect Version

```
#include <stdio.h>
#include <omp.h>
void work()
 {
   int i;

   i = 10;
   #pragma omp task shared(i)
   {
     #pragma omp critical
     printf("In Task, i = %d\n",i);
   }
 }

int main(int argc, char** argv)
 {
    omp_set_num_threads(8);
    omp_set_dynamic(0);

    #pragma omp parallel
    {
      work();
    }
 }
```

EXAMPLE 5–4 Stack Data: First Example — Corrected Version

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

void work()
 {
   int i;

   i = 10;
   #pragma omp task shared(i)
   {
     #pragma omp critical
```

EXAMPLE 5–4 Stack Data: First Example — Corrected Version *(Continued)*

```
     printf("In Task, i = %d\n",i);
   }

   /* Use TASKWAIT for synchronization. */
   #pragma omp taskwait
 }

int main(int argc, char** argv)
 {
    omp_set_num_threads(8);
    omp_set_dynamic(0);

    #pragma omp parallel
    {
      work();
    }
 }
```

In the second example, `j` in the **`task`** construct references the `j` in the **`sections`** construct. Therefore, the task accesses the **`firstprivate`** copy of `j` in the **`sections`** construct, which (in some implementations, including the Oracle Solaris Studio compilers) is a local variable on the stack of the outlined routine for the **`sections`** construct.

Task execution may deferred so the task is executed at the implicit barrier at the end of the **`sections`** region, after the outlined routine for the **`sections`** construct has exited. So when the task references `j`, it accesses some undetermined value on the stack.

For correct results, the programmer needs to make sure that the task is executed before the **`sections`** region reaches its implicit barrier by inserting a **`taskwait`** directive after the **`task`** construct. Alternatively, `j` can be specified to be **`firstprivate`** in the **`task`** construct, instead of **`shared`**.

EXAMPLE 5–5 Second Example — Incorrect Version

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

int main(int argc, char** argv)
 {
    omp_set_num_threads(2);
    omp_set_dynamic(0);
    int j=100;

    #pragma omp parallel shared(j)
    {
       #pragma omp sections firstprivate(j)
       {
          #pragma omp section
          {
             #pragma omp task shared(j)
             {
```

EXAMPLE 5–5 Second Example — Incorrect Version *(Continued)*

```
                  #pragma omp critical
                  printf("In Task, j = %d\n",j);
                }
             }
          }
       }

       printf("After parallel, j = %d\n",j);
    }
```

EXAMPLE 5–6 Second Example — Corrected Version

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

int main(int argc, char** argv)
 {
    omp_set_num_threads(2);
    omp_set_dynamic(0);
    int j=100;

    #pragma omp parallel shared(j)
    {
       #pragma omp sections firstprivate(j)
       {
          #pragma omp section
          {
             #pragma omp task shared(j)
             {
               #pragma omp critical
               printf("In Task, j = %d\n",j);
             }

             /* Use TASKWAIT for synchronization. */
             #pragma omp taskwait
          }
       }
    }

    printf("After parallel, j = %d\n",j);
 }
```

CHAPTER 6

Automatic Scoping of Variables

Declaring the data sharing attributes of variables referenced in an OpenMP construct is called *scoping*. A description of each of the data sharing attributes can be found in Data-Sharing Attribute Clauses section (Chapter 2) of the OpenMP specification.

In an OpenMP program, every variable referenced in an OpenMP construct is scoped. Generally, a variable referenced in a construct may be scoped in one of two ways. Either the programmer explicitly declares the scope of a variable with a *data sharing attribute clause*, or the implementation of the OpenMP API in the compiler automatically applies rules for predetermined or implicitly determined scopes according to the Data Environment: Data Scoping Rules section (Chapter 2) of the OpenMP specification.

Most users find scoping to be the hardest part of using the OpenMP paradigm. Explicitly scoping variables can be tedious and error-prone, especially with large and complicated programs. Moreover, the rules for implicitly determined and predetermined scopes of variables specified in the OpenMP 3.0 specification can yield some unexpected results. The **`task`** directive, which was introduced in OpenMP Specification 3.0, added to the complexity and difficulty of scoping.

The automatic scoping feature (called *autoscoping*)supported by the Oracle Solaris Studio compilers can be a very helpful tool, as it relieves the programmer from having to explicitly determine the scopes of variables. With autoscoping, the compiler determines the scopes of variables by using some smart rules in a very simple user model.

Earlier compiler releases limited autoscoping to variables in a **`parallel`** construct. Current Oracle Solaris Studio compilers extend the autoscoping feature to variables referenced in a **`task`** construct as well.

6.1 Autoscoping Data Scope Clause

Autoscoping is invoked either by specifying the variables to be scoped automatically on a **`__auto`** data scope clause, or by using a **`default(__auto)`** clause. Both are extensions to the OpenMP specification provided by the Oracle Solaris Studio compilers.

6.1.1 `__auto` Clause

Syntax: **`__auto(`***list-of-variables***`)`**

For Fortran, **`__AUTO(`***list-of-variables***`)`** is also accepted.

The **`__auto`** clause on a parallel or task construct directs the compiler to automatically determine the scope of the named variables in the construct. (Note the two underscores before **`auto`**.)

The **`__auto`** clause can appear on a **`PARALLEL`**, **`PARALLEL DO/for`**, **`PARALLEL SECTIONS`**, Fortran 95 **`PARALLEL WORKSHARE`**, or **`TASK`** directive.

If a variable is specified on the **`__auto`** clause, then it cannot be specified in any other data sharing attribute clause.

6.1.2 `default(__auto)` Clause

Syntax: **`default(__auto)`**

For Fortran, **`DEFAULT(__AUTO)`** is also accepted.

The **`default(__auto)`** clause on a parallel or task construct directs the compiler to automatically determine the scope of all variables referenced in the construct that are not explicitly scoped in any data scope clause.

The **`default(__auto)`** clause can appear on a **`PARALLEL`**, **`PARALLEL DO/for`**, **`PARALLEL SECTIONS`**, Fortran 95 **`PARALLEL WORKSHARE`**, or **`TASK`** directive.

6.2 Scoping Rules for a Parallel Construct

Under automatic scoping, the compiler applies the rules described in this section to determine the scope of a variable in a parallel construct.

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

6.2.1 Scoping Rules for Scalar Variables

When autoscoping a scalar variable that is referenced in a parallel construct and that does not have predetermined or implicitly determined scope, the compiler checks the use of the variable against the following rules **PS1**-**PS3** in the given order.

- **PS1**: If the use of the variable in the parallel region is free of *data race* conditions for the threads in the team executing the region, then the variable is scoped **`SHARED`**.
- **PS2**: 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`**.
- **PS3**: 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.

6.2.2 Scoping Rules for Arrays

- **PA1**: 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`**.

6.3 Scoping Rules for a `task` Construct

Under automatic scoping, the compiler applies the rules described in this section to determine the scope of a variable in a **`task`** construct.

These rules do not apply to variables scoped implicitly by the OpenMP specification, such as loop index variables of **`PARALLEL DO`**/**`for`** loops.

6.3.1 Scoping Rules for Scalar Variables

When autoscoping a scalar variable that is referenced in a task construct and that does not have predetermined or implicitly determined scope, the compiler checks the use of the variable against the following rules **TS1**-**TS5** in the given order.

- **TS1:** If the use of the variable is read-only in the **`task`** construct and read-only in the parallel construct in which the task construct is enclosed, then the variable is autoscoped as **`FIRSTPRIVATE`**.
- **TS2:** If the use of the variable is free of data race and the variable will be accessible while the task is executing, then the variable is autoscoped as **`SHARED`**.
- **TS3:** If the use of the variable is free of data race, is read-only in the task construct, and the variable may not be accessible while the task is executing, then the variable is autoscoped as **`FIRSTPRIVATE`**.

- **TS4:** If the use of the variable is not free of data race, and in each thread executing the task region the variable is always written before being read by the same thread, and the value assigned to the variable in the task is not used outside the task region, then the variable is autoscoped as **`PRIVATE`**.
- **TS5:** If the use of the variable is not free of data race, and the variable is not read-only in the task region, and some read in the task region might get the value defined outside the task, and the value assigned to the variable inside the task is not used outside the task region, then the variable is autoscoped as **`FIRSTPRIVATE`**.

6.3.2 Scoping Rules for Arrays

Autoscoping for tasks does not handle arrays.

6.4 General Comments About Autoscoping

Note that task autoscoping rules and autoscoping results could change in future releases. Also, the order that implicitly determined scoping rules and autoscoping rules are applied could change in future releases as well.

The programmer explicitly requests autoscoping with the **`_auto(`***list-of-variables***`)`** clause or the **`default(_auto)`** clause. Specifying the **`default(_auto)`** or **`_auto(`***list-of-variables***`)`** clause for a **`parallel`** construct doesn't imply that same clause applies to **`task`** constructs that are lexically or dynamically enclosed in the **`parallel`** construct.

When autoscoping a variable that does not have predetermined implicit scope, the compiler checks the use of the variable against the above rules in the given order. If a rule matches, the compiler will scope the variable according to the matching rule. If no rule matches, or if autoscoping cannot handle the variable (due to certain restrictions, described in the next section), the compiler will scope the variable as **`SHARED`** and treat the **`parallel`** or **`task`** construct as if an **`IF (.FALSE.)`** or **`if(0)`** clause were specified.

Autoscoping failures generally occur for one of two reasons. One reason is that the use of the variable does not match any of the rules. The other reason 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 synchronizations are some typical causes.

6.5 Restrictions

- To enable autoscoping, the program must be compiled with **`-xopenmp`** at an optimization level of **`-xO3`** or higher. Autoscoping is not enabled if the program is compiled with just **`-xopenmp=noopt`**.
- Parallel and task autosoping in C and C++ can handle only basic data types: integer, floating point, and pointer.
- Task autoscoping cannot handle arrays.
- Task autoscoping in C and C++ cannot handle global variables.
- Task autoscoping cannot handle untied tasks.
- Task autoscoping cannot handle tasks that are lexically enclosed in some other tasks. For example:

  ```
  #pragma omp task /* task1 */
  {
    ...
    #pragma omp task /* task 2 */
    {
      ...
    }
   ...
  }
  ```

  In the example, the compiler does not attempt autoscoping for `task2` because it is lexically nested in `task1`. The compiler will scope all variables referenced in `task2` as **`SHARED`** and treat `task2` as if an **`IF(.FALSE.)`** or **`if(0)`** clause is specified on the task.
- Only OpenMP directives are recognized and used in the analysis. Calls to OpenMP runtime routines 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 by using OpenMP synchronization directives, such as **`BARRIER`** and **`MASTER`**, are recognized and used in data race analysis. User-implemented synchronizations such as busy-waiting are not recognized.

6.6 Checking the Results of Autoscoping

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

The compiler produces an inline commentary when compiled with the **`-g`** debug option that can be viewed with the **`er_src`** command, as shown in the following example. The **`er_src`** command is provided as part of the Oracle Solaris Studio software. For more information, see the **`er_src`**(1) man page or the *Oracle Solaris Studio Performance Analyzer* manual.

A good place to start is to compile with the **-xvpara** option. Compiling with **–xvpara** will give you a general idea about whether autoscoping for a particular construct was successful.

EXAMPLE 6–1 Autoscoping With **-vpara**

```
%cat source1.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 -vpara -c -g source1.f
"source1.f", line 2: Autoscoping for OpenMP construct succeeded.
Check er_src for details
```

If autoscoping fails for a particular construct, a warning message is issued (with **-xvpara**) as shown in the following example.

EXAMPLE 6–2 Autoscoping Failure With **-vpara**

```
%cat source2.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 -g source2.f
"source2.f", line 2: Warning: Autoscoping for OpenMP construct failed.
 Check er-src for details. Parallel region will be executed by
 a single thread.
```

More detailed information appears in the compiler commentary displayed by **er_src**:

EXAMPLE 6–3 Using **er_src**

```
% er_src source2.o
Source file: source2.f
Object file: source2.o
Load Object: source2.o

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

   Source OpenMP region below has tag R1
   Variables autoscoped as SHARED in R1: y
   Variables autoscoped as PRIVATE in R1: t, i
   Variables treated as shared because they cannot be autoscoped in R1: x
   R1 will be executed by a single thread because
```

EXAMPLE 6–3 Using **`er_src`** *(Continued)*

```
  autoscoping for some variable s was not successful
Private variables in R1: i, t
Shared variables in R1: y, x
  2. C$OMP PARALLEL DO DEFAULT(__AUTO)

Source loop below has tag L1
L1 parallelized by explicit user directive
L1 autoparallelized
L1 parallel loop-body code placed in function _$d1A2.MAIN_
   along with 0 inne r loops
L1 could not be pipelined because it contains calls
  3.         DO I=1, 100
  4.                 T = Y(I)
  5.                 CALL FOO(X)
  6.                 X(I) = T*T
  7.         END DO
  8. C$OMP END PARALLEL DO
  9.         END
 10.
```

6.7 Autoscoping Examples

This section provides some examples to illustrate how the autoscoping rules work.

EXAMPLE 6–4 A More Complicated Example

```
     REAL FUNCTION FOO (N, X, Y)
     INTEGER       N, I
     REAL          X(*), Y(*)
     REAL          W, MM, M

     W = 0.0

C$OMP PARALLEL DEFAULT(__AUTO)

C$OMP SINGLE
      M = 0.0
C$OMP END SINGLE

      MM = 0.0

C$OMP DO
      DO I = 1, N
         T = X(I)
         Y(I) = T
         IF (MM .GT. T) THEN
            W = W + T
            MM = T
         END IF
      END DO
C$OMP END DO

C$OMP CRITICAL
```

EXAMPLE 6–4 A More Complicated Example *(Continued)*

```
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. Besides the OpenMP parallel constructs, the code in the parallel region does the following:

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`**.

How does the compiler use the these rules to correctly scope 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 information:

- 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, and 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**. Because no data race occurs, **Y** is scoped **SHARED** according to rule **A1**.

EXAMPLE 6–5 Example With QuickSort

```
static void par_quick_sort (int p, int r, float *data)
{
   if (p < r)
   {
      int q = partition (p, r, data);

      #pragma omp task default(__auto) if ((r-p)>=low_limit)
      par_quick_sort (p, q-1, data);

      #pragma omp task default(__auto) if ((r-p)>=low_limit)
      par_quick_sort (q+1, r, data);
   }
}

int main ()
{
  ...
  #pragma omp parallel
  {
     #pragma omp single nowait
     par_quick_sort (0, N-1, &Data[0]);
  }
  ...
}

er_src result:

      Source OpenMP region below has tag R1
      Variables autoscoped as FIRSTPRIVATE in R1: p, q, data
      Firstprivate variables in R1: data, p, q
        47. #pragma omp task default(__auto) if ((r-p)>=low_limit)
        48. par_quick_sort (p, q-1, data);

      Source OpenMP region below has tag R2
      Variables autoscoped as FIRSTPRIVATE in R2: q, r, data
      Firstprivate variables in R2: data, q, r
        49. #pragma omp task default(__auto) if ((r-p)>=low_limit)
        50. par_quick_sort (q+1, r, data);
```

The scalar variables p and q, and the pointer variable data, are read-only in the task construct, and read-only in the parallel region. Therefore, they are autoscoped as **FIRSTPRIVATE** according to **TS1.**

EXAMPLE 6–6 Another Fibbonacci Example

```
int fib (int n)
{
   int x, y;
   if (n < 2) return n;

   #pragma omp task default(__auto)
   x = fib(n - 1);
```

EXAMPLE 6–6 Another Fibbonacci Example *(Continued)*

```
   #pragma omp task default(__auto)
   y = fib(n - 2);

   #pragma omp taskwait
   return x + y;
}

er_src result:

   Source OpenMP region below has tag R1
   Variables autoscoped as SHARED in R1: x
   Variables autoscoped as FIRSTPRIVATE in R1: n
   Shared variables in R1: x
   Firstprivate variables in R1: n
    24.         #pragma omp task default(__auto) /* shared(x) firstprivate(n) */
    25.         x = fib(n - 1);

   Source OpenMP region below has tag R2
   Variables autoscoped as SHARED in R2: y
   Variables autoscoped as FIRSTPRIVATE in R2: n
   Shared variables in R2: y
   Firstprivate variables in R2: n
    26.         #pragma omp task default(__auto) /* shared(y) firstprivate(n) */
    27.         y = fib(n - 2);
    28.
    29.         #pragma omp taskwait
    30.         return x + y;
    31. }
```

Scalar `n` is read-only in the task constructs and read-only in the parallel construct. Therefore, `n` is autoscoped as **`FIRSTPRIVATE`**, according to TS1.

Scalar variables `x` and `y` are local variables of function `fib()`. Accesses to `x` and `y` in both tasks are free of data race. Because there is a **`taskwait`**, the two tasks will complete execution before the thread executing `fib()`, which encountered the tasks, exits `fib()`. This implies that `x` and `y` will be active while the two tasks are executing. Therefore, `x` and `y` are autoscoped as **`SHARED`**, according to TS2.

EXAMPLE 6–7 Another Autoscoping Example

```
int main(void)
{
  int yy = 0;

  #pragma omp parallel default(__auto) shared(yy)
  {
    int xx = 0;

    #pragma omp single
    {
       #pragma omp task default(__auto) // task1
       {
          xx = 20;
```

EXAMPLE 6–7 Another Autoscoping Example *(Continued)*

```
        }
     }

     #pragma omp task default(__auto) // task2
     {
        yy = xx;
     }
   }

   return 0;
}

er_src result:

   Source OpenMP region below has tag R1
   Variables autoscoped as PRIVATE in R1: xx
   Private variables in R1: xx
   Shared variables in R1: yy
     7.   #pragma omp parallel default(__auto) shared(yy)
     8.   {
     9.     int xx = 0;
    10.

   Source OpenMP region below has tag R2
    11.     #pragma omp single
    12.     {

   Source OpenMP region below has tag R3
   Variables autoscoped as SHARED in R3: xx
   Shared variables in R3: xx
    13.        #pragma omp task default(__auto) // task1
    14.        {
    15.           xx = 20;
    16.        }
    17.     }
    18.

   Source OpenMP region below has tag R4
   Variables autoscoped as PRIVATE in R4: yy
   Variables autoscoped as FIRSTPRIVATE in R4: xx
   Private variables in R4: yy
   Firstprivate variables in R4: xx
    19.     #pragma omp task default(__auto) // task2
    20.     {
    21.        yy = xx;
    22.     }
    23.   }
```

In this example, `xx` is a private variable in the parallel region. One of the threads in the team modifies its initial value of `xx` by executing `task1`. Then all of the threads encounter `task2`, which uses `xx` to do some computation.

In `task1`, the use of `xx` is free of data race. Because an implicit barrier is at the end of the single construct and `task1` should complete before exiting this barrier, `xx` will be active while `task1` is executing. Therefore, according to TS2, `xx` is autoscoped as **`SHARED`** on `task1`.

EXAMPLE 6–7 Another Autoscoping Example *(Continued)*

In task2, the use of xx is read-only. However, the use of xx is not read-only in the enclosing parallel construct. Because xx is predetermined as **PRIVATE** for the parallel construct, whether xx will be active while task2 is executing is not certain. Therefore, according to TS3, xx is autoscoped **FIRSTPRIVATE** on task2.

In task2, the use of yy is not free of data race, and in each thread executing task2, the variable yy is always written before being read by the same thread. So, according to TS4, yy is autoscoped **PRIVATE** on task2.

EXAMPLE 6–8 A Final Example

```
int foo(void)
{
  int xx = 1, yy = 0;

  #pragma omp parallel shared(xx,yy)
  {
    #pragma omp task default(__auto)
    {
       xx += 1;

       #pragma omp atomic
       yy += xx;
    }

    #pragma omp taskwait
  }
  return 0;
}
```

```
er_src result:

   Source OpenMP region below has tag R1
   Shared variables in R1: yy, xx
     5.   #pragma omp parallel shared(xx,yy)
     6.   {

   Source OpenMP region below has tag R2
   Variables autoscoped as SHARED in R2: yy
   Variables autoscoped as FIRSTPRIVATE in R2: xx
   Shared variables in R2: yy
   Firstprivate variables in R2: xx
     7.     #pragma omp task default(__auto)
     8.     {
     9.        xx += 1;
    10.
    11.        #pragma omp atomic
    12.        yy += xx;
    13.     }
    14.
    15.     #pragma omp taskwait
    16.   }
```

EXAMPLE 6–8 A Final Example *(Continued)*

The use of xx in the **task** construct is not read-only and is not free of data race. However the read of x in the task region gets the value of x defined outside the task. (In this example, because xx is **SHARED** for the parallel region, the definition of x is actually outside the parallel region.) Therefore, according to TS5, xx is autoscoped as **FIRSTPRIVATE**.

The use of yy in the **task** construct is not read-only but is free of data race. yy will be accessible while the task is executing because there is a **taskwait**. Therefore, according to TS2, yy is autoscoped as **SHARED**.

CHAPTER 7

Scope Checking

Autoscoping can help the programmer decide how to scope variables. However, for some complicated programs, autoscoping might not be successful or the result of autoscoping might not be what the programmer expects. Incorrect scoping can cause many inconspicuous yet serious problems. For example, incorrectly scoping some variable as **`SHARED`** may cause a data race; incorrectly privatizing a variable may result in an undefined value for the variable outside the construct.

Oracle Solaris Studio C, C++, and Fortran compilers provide a compile-time scope-checking feature where the compiler determines whether variables in an OpenMP program are correctly scoped.

Based on the compiler's capabilities, scope checking can discover potential problems including data races, inappropriate privatization or reduction of variables, and other scoping issues. During scope checking, the data-sharing attributes specified by the programmer, the implicit data-sharing attributes determined by the compiler, and autoscoping results are all checked by the compiler.

7.1 Using the Scope Checking Feature

To enable scope checking, compile the OpenMP program with the **`-xvpara`** and **`-xopenmp`** options, and at optimization level **`-xO3`** or higher. Scope checking does not work if the program is compiled with just **`-xopenmp=noopt`**. If the optimization level is less than **`-xO3`**, the compiler will issue a warning message and will not do any scope checking.

During scope checking, the compiler will check all OpenMP constructs. If the scoping of some variables causes problems, the compiler will issue warning messages, and, in some cases, suggestions for the correct data sharing attribute clause to use.

For example:

EXAMPLE 7–1 Scope Checking

```
% cat t.c

#include <omp.h>
#include <string.h>

int main()
{
  int g[100], b, i;

  memset(g, 0, sizeof(int)*100);

  #pragma omp parallel for shared(b)
  for (i = 0; i < 100; i++)
  {
    b += g[i];
  }

  return 0;
}

%  cc -xopenmp -xO3 -xvpara source1.c
"source1.c", line 10: Warning: inappropriate scoping
         variable ’b’ may be scoped inappropriately as ’shared’
         . write at line 13 and write at line 13 may cause data race

"source1.c", line 10: Warning: inappropriate scoping
         variable ’b’ may be scoped inappropriately as ’shared’
         . write at line 13 and read at line 13 may cause data race
```

The compiler will not do scope checking if the optimization level is less than **-xO3**.

```
%  cc -xopenmp=noopt  -xvpara source1.c
 "source1.c", line 10: Warning: Scope checking under vpara compiler
option is supported with optimization level -xO3 or higher.
 Compile with a higher optimization level to enable this feature
```

The following example shows potential scoping errors:

EXAMPLE 7–2 Scoping Error Example

```
% cat source2.c

#include <omp.h>

int main()
{
  int g[100];
  int r=0, a=1, b, i;

  #pragma omp parallel for private(a) lastprivate(i) reduction(+:r)
  for (i = 0; i < 100; i++)
  {
    g[i] = a;
    b = b + g[i];
    r = r * g[i];
```

EXAMPLE 7–2 Scoping Error Example *(Continued)*

```
  }

  a = b;
  return 0;
}

% cc -xopenmp -xO3 -xvpara source2.c
"source2.c", line 8: Warning: inappropriate scoping
        variable 'r' may be scoped inappropriately as 'reduction'
        . reference at line 13 may not be a reduction of the specified type

"source2.c", line 8: Warning: inappropriate scoping
        variable 'a' may be scoped inappropriately as 'private'
        . read at line 11 may be undefined
        . consider 'firstprivate'

"source2.c", line 8: Warning: inappropriate scoping
        variable 'i' may be scoped inappropriately as 'lastprivate'
        . value defined inside the parallel construct is not used outside
        . consider 'private'

"source2.c", line 8: Warning: inappropriate scoping
        variable 'b' may be scoped inappropriately as 'shared'
        . write at line 12 and write at line 12 may cause data race

"source2.c", line 8: Warning: inappropriate scoping
        variable 'b' may be scoped inappropriately as 'shared'
        . write at line 12 and read at line 12 may cause data race
```

This artifical example shows some typical errors of scoping that scope checking can detect.

1. `r` is specified as a reduction variable whose operation is **`+`**, but actually the operation should be **`*`**.
2. `a` is explicitly scoped as **`PRIVATE`**. Because **`PRIVATE`** variables do not have an initial value, the reference on line 11 to `a` could read garbage values. The compiler points out this problem, and suggests that the programmer consider scoping `a` as **`FIRSTPRIVATE`**.
3. Variable `i` is the loop index variable. In some cases, the programmer may wish to specify it to be **`LASTPRIVATE`** if the value of the loop index is used after the loop. However, in the example, `i` is not referenced at all after the loop. The compiler issues a warning and suggests that the programmer scope `i` as **`PRIVATE`**. Using **`PRIVATE`** instead of **`LASTPRIVATE`** can lead to better performance.
4. The programmer does not explicitly specify a data-sharing attribute for variable `b`. According to page 79, lines 27-28 of the OpenMP Specification 3.0, `b` will be implicitly scoped as **`SHARED`**. However, scoping `b` as **`SHARED`** will cause a data race. The correct data-sharing attribute of `b` should be **`REDUCTION`**.

7.2 Restrictions

- As mentioned, scope checking works only with optimization level **`-xO3`** or higher. Scope checking does not work if the program is compiled with just **`-xopenmp=noopt`**.
- Only synchronizations specified by using OpenMP synchronization directives, such as **`BARRIER`** and **`MASTER`**, are recognized and used in the data race analysis. User-implemented synchronizations, such as *busy-waiting*, are not recognized.

CHAPTER 8

Performance Considerations

Once you have a correct, working OpenMP program, consider its overall performance. This chapter describes some general techniques that you can utilize to improve the efficiency and scalability of an OpenMP application.

The Oracle Solaris Studio portal posts occasional articles and case studies regarding performance analysis and optimization of OpenMP applications at `http://www.oracle.com/technetwork/server-storage/solarisstudio`.

8.1 Some General Performance Recommendations

Some general techniques for improving performance of OpenMP applications are:

- 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 might require reloads from memory, all of which decrease efficiency.

  By default, idle threads will be put to sleep after a certain timeout period. If a thread does not find work by the end of the timeout period, it will go to sleep, thus avoiding wasting processor cycles at the expense of other threads. The default timeout period might not be sufficient for your application, causing the threads to go to sleep too soon or too late. In general, if an application has dedicated processors to run on, **`SPIN`** would give best application performance. If an application shares processors with other applications at the same time, then the default setting, or a setting that does not let the threads spin for too long, would be best for system throughput .Use the **SUNW_MP_THR_IDLE** environment variable to override the default timeout period, even up to the point where the idle threads will never be put to sleep and remain active all the time.

- 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
```

  A more efficient construct:

```
!$OMP PARALLEL
  ....
  !$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
```

  while this construct is more efficient:

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

- On Oracle Solaris systems, use **SUNW_MP_PROCBIND** to bind threads 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 "2.3 Processor Binding" on page 20.
- 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, which 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** could lead to load imbalance.
  - **DYNAMIC,GUIDED** incurs a synchronization overhead to keep track of which chunks have been assigned. 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**.

  Small data cases could cause OpenMP parallel loops to underperform. Use the **if** clause on **PARALLEL** constructs to indicate that a loop should run parallel only in those cases where some performance gain can be expected.
- When possible, merge loops. For example, merge two loops:

```
!$omp parallel do
  do i = ...
..........statements_1...
  end do
```

```
!$omp parallel do
  do i = ...
..........statements_2...
  end do
```

Merged into a single loop:

```
!$omp parallel do
  do i = ...
..........statements_1...
..........statements_2...
  end do
```

- Try nested parallelism if your application lacks scalability beyond a certain level. See "1.2 Special Conventions" on page 11 for more information about nested parallelism in OpenMP.

8.2 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.

8.2.1 What Is *False Sharing*?

Most high performance processors, such as UltraSPARC processors, 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.

However, simultaneous updates of individual elements in the same cache line coming from different processors invalidate 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 more recent copy of the line from memory or elsewhere, even though the element accessed has not been modified. This occurs because cache coherency is maintained on a cache-line basis and not for individual elements. As a result, interconnect traffic and overhead increases. Also, while the cache-line update is in progress, access to the elements in the line is inhibited.

This situation is called *false sharing*. If it occurs frequently, performance and scalability of an OpenMP application suffers significantly.

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.
- Data updating occurs very frequently (as in a tight loop).

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

8.2.2 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 the following techniques:

- Making use of private data as much as possible
- Using the compiler's optimization features to eliminate memory loads and stores

In specific cases, the impact of false sharing might be less visible when dealing with larger problem sizes, as there might be less sharing.

Techniques for tackling false sharing are very much dependent on the particular application. In some cases, a change in the way the data is allocated can reduce false sharing. In other cases, changing the mapping of iterations to threads by giving each thread more work per chunk (by changing the *chunksize* value) can also lead to a reduction in false sharing.

8.3 Oracle Solaris OS Tuning Features

Oracle Solaris supports features that improve the performance of OpenMP programs without hardware upgrades, such as Memory Placement Optimizations (MPO) and Multiple Page Size Support (MPSS), among others.

MPO emables the OS to allocate pages close to the processors that access those pages. SunFire E20K and SunFire E25K systems have different memory latencies within the same UniBoard as opposed to 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.

The MPSS feature enables a program to use different page sizes for different regions of virtual memory. The default Oracle Solaris page size is relatively small (8 KB on UltraSPARC processors and 4 KB on AMD64 Opteron processors). Applications that suffer from too many TLB misses could experience a performance boost by using a larger page size.

TLB misses can be measured using the Oracle Solaris Studio Performance Analyzer.

The default page size on a specific platform can be obtained with the Oracle Solaris OS command: **`/usr/bin/pagesize`** . The **`-a`** option on this command lists all the supported page sizes. (See the **`pagesize`**(1) man page for details.)

The three ways to change the default page size for an application are:

- Use the Oracle Solaris OS command **`ppgsz`**(1).
- Compile the application with the **`-xpagesize`**, **`-xpagesize_heap`**, and **`-xpagesize_stack`** options. See the compiler man pages for details.
- Use MPSS specific environment variables. See the **`mpss.so.1`**(1) man page for details.

◆ ◆ ◆ APPENDIX A

Placement of Clauses on Directives

TABLE A–1 Pragmas Where Clauses Can Appear

Clause/Pragma	parallel	do/for	sections	single	parallel do/for	parallel sections	parallel workshare	task
if	+				+	+	+	+
private	+	+	+	+	+	+	+	+
shared	+				+	+	+	+
firstprivate	+	+	+	+	+	+	+	+
lastprivate		+	+		+	+		
default	+				+	+	+	+
reduction	+	+	+		+	+	+	
copyin	+				+	+	+	
copyprivate				+ (1)				
ordered		+			+			
schedule		+			+			
nowait		+ (2)	+ (2)	+ (2)				
num_threads	+				+	+	+	
untied								+
final								+
mergeable								+
__auto	+				+	+	+	+

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`**.

Index

A
__auto, 50
automatic scoping, 49–61

C
cache line, 70
compiling for OpenMP, 13–23

D
default(_auto), 50
directive, 12
documentation, accessing, 7
documentation index, 7
dynamic thread adjustment, 15

E
environment variables, 15–20

F
false sharing, 70

G
guided scheduling, 18

I
idle threads, 17
implementation, 25

M
memory placement optimization (MPO), 71

N
nested parallelism, 15, 31, 32
number of threads, **OMP_NUM_THREADS**, 15

O
OMP_DYNAMIC, 15
OMP_MAX_ACTIVE_LEVELS, 16, 33
OMP_NESTED, 15, 32
OMP_NUM_THREADS, 15
OMP_PROC_BIND, 16
OMP_SCHEDULE, 15
OMP_STACKSIZE, 15
OMP_THREAD_LIMIT, 16
OMP_WAIT_POLICY, 16
OpenMP API specification, 11
Oracle Solaris OS tuning, 71

P

PARALLELenvironment variable, 16
parallelism, nested, 31
performance, 67
pragma, 12

S

scalability, 70
scheduling, **OMP_SCHEDULE**, 15
scoping of variables
 automatic, 49–61
 compiler commentary, 53–55
 rules, 50–51
SLEEP, 17
SPIN, 17
stack size, 18, 22
stacks, 22
STACKSIZE, 18
-stackvar, 22
SUNW_MP_MAX_POOL_THREADS, 33
SUNW_MP_THR_IDLE, 17
SUNW_MP_WAIT_POLICY, 19
SUNW_MP_WARN, 16

T

task construct, automatic scoping rules, 51–52
thread stack size, 18

W

warning messages, 16
weighting factor, 18

X

-xopenmp, 13