Oracle® Developer Studio 12.6: OpenMP API User's Guide

Part No: E77801
June 2017

ORACLE

Oracle Developer Studio 12.6: OpenMP API User's Guide

Part No: E77801

Référence: E77801

Accès aux services de support Oracle

Les clients Oracle qui ont souscrit un contrat de support ont accès au support électronique via My Oracle Support. Pour plus d'informations, visitez le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` ou le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` si vous êtes malentendant.

Contents

Examples

Using This Documentation

- **Overview** – Describes the specifics of the OpenMP API supported by the Oracle Developer Studio 12.6 C, C++, and Fortran compilers
- **Audience** – Application developers, system developers, architects, support engineers
- **Required knowledge** – Programming experience, software development testing, aptitude to build and compile software products

Product Documentation Library

Documentation and resources for this product and related products are available at http://www.oracle.com/pls/topic/lookup?ctx=E77782-01

Feedback

Provide feedback about this documentation at http://www.oracle.com/goto/docfeedback.

♦ ♦ ♦ CHAPTER 1

Introducing the OpenMP API

The OpenMP Application Program Interface (API) is a portable, parallel programming model for writing multithreaded programs, developed in collaboration with a number of computer vendors, academics, and researchers. The OpenMP specifications are created and published by the OpenMP Architecture Review Board.

The OpenMP API is the recommended parallel programming model for all Oracle Developer Studio compilers.

1.1 Supported OpenMP Specification

This manual describes issues specific to the Oracle Developer Studio implementation of the OpenMP API specification version 4.0 (also referred to in this manual as OpenMP 4.0). The specification can be found on the official OpenMP web site at http://www.openmp.org/.

Note - For the best performance and functionality on Oracle Solaris platforms, make sure that the latest version of the OpenMP runtime library, `libmtsk.so`, is installed on the running system.

The latest information about the Oracle Developer Studio compiler releases and their implementation of the OpenMP API can be found on the Oracle Developer Studio portal at http://www.oracle.com/technetwork/server-storage/developerstudio.

Note - This release of Oracle Developer Studio fully supports the OpenMP 4.0 specification. However, the following should be noted:

- SIMD constructs are accepted. However, SIMD constructs may not result in the use of any SIMD instructions.
- Device constructs are accepted. However, all code will be executed on the host device. The only device available is the host device.

1.2 Special Conventions for This Document

The term *structured-block* refers to a block of C, C++, or Fortran 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 Developer Studio compiler, `f95`(1).

The terms “directive” and “pragma” are used interchangeably in this manual. OpenMP 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 use the OpenMP API.

2.1 Compiler Options

To enable explicit parallelization with OpenMP directives, compile your program with the `cc`, `CC`, or `f95` compiler option `-xopenmp`. The `f95` compiler accepts both `-xopenmp` and `-openmp` as synonyms.

The `-xopenmp` flag accepts the keyword sub-options listed in the following table.

`-xopenmp=parallel`	Enables recognition of OpenMP directives. The minimum optimization level for `-xopenmp=parallel` is `-xO3`. If the optimization level is lower than `-xO3`, the compiler raises the optimization level to `-xO3` and issues a warning.
`-xopenmp=noopt`	Enables recognition of OpenMP directives. The compiler does not raise the optimization level if it is lower than `-xO3`. If you explicitly set the optimization level with `-xopenmp=noopt` lower than `-xO3`, as in `-xO2 -xopenmp=noopt`, the compiler will issue an error. If you do not specify an optimization level with `-xopenmp=noopt`, the OpenMP directives are recognized and the program is parallelized accordingly, but no optimization is done.
`-xopenmp=stubs`	This option is no longer supported. *For C and C++ programs only:* An OpenMP stubs library is provided for users' convenience. To compile an OpenMP program that calls OpenMP runtime routines but ignores the OpenMP

	directives, compile the program without the `-xopenmp` option and link the object files with the `libompstubs.a` library. For example, `% cc omp_ignore.c -lompstubs` **Note -** 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 directives and does not change the optimization level.

Note the following additional points:

- If you do not specify `-xopenmp` on the command line, the compiler assumes `-xopenmp=none` (disable recognition of OpenMP directives) by default.
- If you specify `-xopenmp` but without a keyword sub-option, the compiler assumes `-xopenmp=parallel`.
- Specifying `-xopenmp=parallel` or `-xopenmp=noopt` will define the `_OPENMP` macro to have the decimal value `201307L` in C/C++ and `201307` in Fortran, where 2013 is the year and 07 is the month of the OpenMP 4.0 specification.
- When debugging OpenMP programs with `dbx`, compile with `-xopenmp=noopt -g` to enable full debugging capabilities.
- To avoid compilation warning messages, specify an appropriate optimization level explicitly rather than relying on the default value, which is subject to change.
- With Fortran, compiling with `-xopenmp`, `-xopenmp=parallel`, or `-xopenmp=noopt` implies `-stackvar`. See "2.3 Stacks and Stack Sizes" on page 24.
- When compiling and linking an OpenMP program in separate steps, include `-xopenmp` in each of the compile and the link steps.
- Use the `-xvpara` option with the `-xopenmp` option to display compiler warnings about potential OpenMP programming problems (see Chapter 7, "Scope Checking").

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 4.0 specification at http://openmp.org. Also see Chapter 9, "OpenMP Implementation-Defined Behaviors" for information about the implementation of OpenMP environment variables in Oracle Developer Studio.

Oracle Developer Studio supports additional environment variables which are not part of the OpenMP specification, are summarized in "2.2.2 Oracle Developer Studio Environment Variables" on page 19.

Note - The default number of threads for OpenMP and `autopar` programs is a multiple of the number of cores per socket (that is, cores per processor chip), which is less than or equal to MIN(total number of cores, 32).

2.2.1 OpenMP Environment Variable Behaviors and Defaults

The following table describes the behaviors of the OpenMP environment variables supported by Oracle Developer Studio and their default values. Note that the values specified for the environment variables are case insensitive and can be in uppercase or lowercase.

Environment Variable	Behavior, Default Value, and Example
`OMP_SCHEDULE`	If the schedule type specified for the `OMP_SCHEDULE` is not one of the valid types (`static`, `dynamic`, `guided`, `auto`, `sunw_mp_sched_reserved`), 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 size specified 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()`. If not set, a default value of `static` (with no chunk size) is used. Example: `%` **`setenv OMP_SCHEDULE "GUIDED,4"`**
`OMP_NUM_THREADS`	If the value specified for `OMP_NUM_THREADS` is not a positive integer, then the environment variable 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 value specified 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 execution. If not set, the default is a multiple of the number of cores per socket (that is, cores per processor chip), which is less than or equal to `MIN` (total number of cores, 32). Example: `%` **`setenv OMP_NUM_THREADS 16`**

Environment Variable	Behavior, Default Value, and Example
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(). If not set, the default is TRUE. Example: % **setenv OMP_DYNAMIC FALSE**
OMP_PROC_BIND	If the value specified for OMP_PROC_BIND is not TRUE, FALSE, or a comma separated list of master, close, or spread, then the process is exited with a nonzero status. If an initial thread cannot be bound to the first place in the OpenMP place list, then the process is exited with a nonzero status. If not set, the default is FALSE. Example: % **setenv OMP_PROC_BIND spread**
OMP_PLACES	If the value specified for OMP_PLACES is not valid or cannot be fulfilled, then the process is exited with a nonzero status. If not set, the default is cores. Example: % **setenv OMP_PLACES sockets**
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(). If not set, the default is FALSE. Example: % **setenv OMP_NESTED TRUE**
OMP_STACKSIZE	If the value specified 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) 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(). The default stack size of a helper thread is 4 Megabytes for 32-bit applications, and 8 Megabytes for 64-bit applications. Example: % **setenv OMP_STACKSIZE 10M**
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. If not set, the default is PASSIVE. Example: % **setenv OMP_WAIT_POLICY ACTIVE**
OMP_MAX_ACTIVE_LEVELS	If the value specified for OMP_MAX_ACTIVE_LEVELS is not a non-negative integer, then the value will be ignored and the default value of 4 will be used. A warning message will

Environment Variable	Behavior, Default Value, and Example
	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 not set, the default is 4. Example: % **setenv OMP_MAX_ACTIVE_LEVELS 8**
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 of 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(). If not set, the default is 1024. Example: % **setenv OMP_THREAD_LIMIT 128**
OMP_CANCELLATION	If the value specified for OMP_CANCELLATION 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(). If not set, the default is FALSE. Example: % **setenv OMP_CANCELLATION TRUE**
OMP_DISPLAY_ENV	If the value specified for OMP_DISPLAY_ENV is not TRUE, FALSE, or VERBOSE, 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(). If not set, the default is FALSE. Example: % **setenv OMP_DISPLAY_ENV VERBOSE**

2.2.2 Oracle Developer Studio Environment Variables

The following additional environment variables affect the execution of OpenMP programs but 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.

If both PARALLEL and OMP_NUM_THREADS are set, they must be set to the same value.

2.2.2.2 SUNW_MP_WARN

The OpenMP runtime library has the ability to issue warnings about many common OpenMP violations, such as incorrect nesting of regions, incorrect placement of explicit barriers, deadlocks, invalid settings of environment variables, and the like.

The environment variable `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`. If the environment variable is set to `FALSE`, the runtime library does not issue any warning messages. 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 callback function by calling the following function:

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

The address of the callback function is passed as an argument to `sunw_mp_register_warn()`. `sunw_mp_register_warn()` returns 0 upon successfully registering the callback 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 warning message. The memory pointed to is no longer valid upon return from the callback function.

Note - Set `SUNW_MP_WARN` to `TRUE` while testing or debugging a program to enable runtime checking and to display warning messages from the OpenMP runtime library. Be aware that runtime checking adds overhead to the execution of the program.

2.2.2.3 SUNW_MP_THR_IDLE

Controls the behavior of threads in an OpenMP program that are waiting for work (idle) or waiting at a barrier. 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` are optional suffixes that specify the time unit (seconds, milliseconds, and microseconds, respectively). If the time unit is not specified, then a time unit of seconds is assumed.

`SPIN` specifies that a thread should spin while waiting for work (idle) or waiting at a barrier. `SLEEP` without a time parameter specifies that a waiting thread should sleep immediately. `SLEEP`

with a time parameter specifies the amount of time a thread should spin-wait before going to sleep.

The default behavior 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
```

The following are all equivalent:

```
% setenv SUNW_MP_THR_IDLE SLEEP(5)
% setenv SUNW_MP_THR_IDLE SLEEP(5s)
% setenv SUNW_MP_THR_IDLE SLEEP(5000ms)
% setenv SUNW_MP_THR_IDLE SLEEP(5000000mc)
```

2.2.2.4 SUNW_MP_PROCBIND

The `SUNW_MP_PROCBIND` environment variable can be used to bind OpenMP threads to hardware threads on the running system. Performance can be enhanced with processor binding, but performance degradation will occur if multiple threads are bound to the same hardware thread. You cannot set both `SUNW_MP_PROCBIND` and `OMP_PROC_BIND`. If `SUNW_MP_PROCBIND` is not set, the default is `FALSE`. See Chapter 5, “Processor Binding (Thread Affinity)” for more information.

2.2.2.5 SUNW_MP_MAX_POOL_THREADS

Specifies the maximum size of the OpenMP helper thread pool. OpenMP helper threads are those threads that the OpenMP runtime library creates to work on parallel regions. The helper thread pool does not include the initial (or main) thread or any threads created explicitly by the user’s program. If this environment variable is set to zero, the OpenMP helper thread pool will be empty and all parallel regions will be executed by the initial (or main) thread. If not set, the default is 1023. See “3.2 Control of Nested Parallelism” on page 29 for more information.

Note that `SUNW_MP_MAX_POOL_THREADS` specifies the maximum number of *non-user* OpenMP threads to use for the program, while `OMP_THREAD_LIMIT` specifies the maximum number of *user and non-user* OpenMP threads to use for the program. If both `SUNW_MP_MAX_POOL_THREADS`

and OMP_THREAD_LIMIT are set, they must be set to consistent values. The value of OMP_THREAD_LIMIT must be 1 more than the value of SUNW_MP_MAX_POOL_THREADS.

2.2.2.6 SUNW_MP_MAX_NESTED_LEVELS

Sets 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 set, the default is 4. See "3.2 Control of Nested Parallelism" on page 29 for more information.

2.2.2.7 STACKSIZE

Sets the stack size for each OpenMP helper thread. The environment variable accepts numeric values with an optional suffix of B, K, M, or G for Bytes, Kilobytes, Megabytes, or Gigabytes, respectively. If no suffix is specified, the default is Kilobytes.

If not set, the default OpenMP helper thread stack size is 4 Megabytes for 32-bit applications, and 8 Megabytes for 64-bit applications.

Examples:

```
% setenv STACKSIZE 8192  <- sets the OpenMP helper thread stack size to 8 Megabytes
% setenv STACKSIZE 16M   <- sets the OpenMP helper thread stack size to 16 Megabytes
```

Note that if both STACKSIZE and OMP_STACKSIZE are set, they must be set to the same value.

2.2.2.8 SUNW_MP_GUIDED_WEIGHT

Sets the weighting factor used to determine the size of chunks in loops with the guided schedule. The value should be a positive floating-point number, and will apply to all loops with the guided schedule in the program. If not set, the default weighting factor is 2.0.

When the schedule(guided, *chunk_size*) clause is specified with the for/do directive, the loop iterations are assigned to threads in chunks as the threads request them, with the chunk sizes decreasing to *chunk_size*, except that the last chunk may have a smaller size. The thread executes a chunk of iterations and then requests another chunk until no chunks remain to be assigned. For a *chunk_size* of 1, the size of each chunk is proportional to the number of unassigned iterations divided by the number of threads, decreasing to 1. For a *chunk_size* of *k* (where *k* is greater than 1), the size of each chunk is determined in the same way with the restriction that the chunks do not contain fewer than *k* iterations except possibly for the last chunk. When no *chunk_size* is specified, the value defaults to 1.

The OpenMP runtime library, `libmtsk.so`, uses the following formula to compute the chunk sizes for a loop with the `guided` schedule:

chunk_size `=` *num-unassigned-iters* `/` (*guided-weight* `*` *num-threads*)

- *num-unassigned-iters* is the number of iterations in the loop that have not yet been assigned to any thread.
- *guided-weight* is the weighting factor specified by the `SUNW_MP_THR_GUIDED_WEIGHT` environment variable (or 2.0 if the environment variable is not set).
- *num-threads* is the number of threads used to execute the loop.

To illustrate, suppose a 100-iteration loop with the `guided` schedule. If *num-threads* = 4 and the weighting factor = 1.0, then the chunk sizes will be:

25, 18, 14, 10, 8, 6, 4, 3, 3, 2, 1,...

On the other hand, if *num-threads*= 4 and the weighting factor = 2.0, then the chunk sizes will be:

12, 11, 9, 8, 7, 6, 5, 5, 4, 4, 3,...

2.2.2.9 SUNW_MP_WAIT_POLICY

Allows fine-grained control of the behavior of OpenMP 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 these types of wait has three possibilities: spin for a while, yield the processor for a while, or sleep until awakened.

The syntax (shown using `csh`) is as follows:

```
% setenv SUNW_MP_WAIT_POLICY "IDLE=val:BARRIER=val:TASKWAIT=val"
```

`IDLE`, `BARRIER`, and `TASKWAIT` are optional keywords that specify the type of wait being controlled. `IDLE` refers to the wait for work. `BARRIER` refers to the wait at an explicit or implicit barrier. `TASKWAIT` refers to the wait at a `taskwait` region. Each of these keywords is followed by a *val* setting that describes the wait behavior using the keywords `SPIN`, `YIELD`, or `SLEEP`.

`SPIN(`*time*`)` specifies how long a waiting thread should spin before yielding the processor. *time* can be in seconds, milliseconds, or microseconds (denoted by `s`, `ms`, or `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 processor before sleeping. After each yield of the processor, 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 waiting thread should immediately go to sleep.

Note that the `SPIN`, `SLEEP`, and `YIELD` settings for a particular type of wait can be specified in any order. The settings must be separated by comma. `"SPIN(0),YIELD(0)"` is the same as `"YIELD(0),SPIN(0)"`, which is equivalent to `SLEEP` or sleep immediately. When processing the settings for `IDLE`, `BARRIER`, and `TASKWAIT`, the “left-most wins” rule is used. The "left-most wins" rule means that if different values are specified for the same type of wait, then the left-most value is the one that will apply. In the following example, two values are specified for `IDLE`. The first is `SPIN`, and the second is `SLEEP`. Because `SPIN` appears first (it is the left-most in the string), this is the value that will be applied by the OpenMP runtime library.

```
% setenv SUNW_MP_WAIT_POLICY "IDLE=SPIN:IDLE=SLEEP"
```

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

Example 1:

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

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

Example 2:

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

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

Example 3:

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

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

2.3 Stacks and Stack Sizes

Stacks are temporary memory address spaces used to hold arguments and automatic variables during an invocation of a subprogram or function. Stack overflow might occur if the size of a thread's stack is too small, causing silent data corruption or segmentation fault.

The executing program maintains a main stack for the initial (or main) thread executing the program. 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 addition, each OpenMP helper thread in the 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` variables are allocated on the thread stack. The default size of a helper thread stack is 4 Megabytes for 32-bit applications, and 8 Megabytes for 64-bit applications. Use the `OMP_STACKSIZE` environment variable to set the size of the helper thread stack.

Note that compiling Fortran programs with the `-stackvar` option forces the allocation of local variables and arrays on the stack as if they were automatic variables. `-stackvar` is implied with programs compiled with the `-xopenmp`, `-xopenmp=parallel`, or `-xopenmp=noopt` option. This could lead to stack overflow if not enough memory is allocated for the stack. Take extra care to ensure that the stacks are large enough.

Example for C shell:

```
% limit stacksize 32768    <- Sets the main thread stack size to 32 Megabytes
% setenv OMP_STACKSIZE 16384    <- Sets the helper thread stack size to 16 Megabytes
```

Example for Bourne or Korn shell:

```
$ ulimit -s 32768    <- Sets the main thread stack size to 32 Megabytes

$ OMP_STACKSIZE=16384    <- Sets the helper thread stack size to 16 Megabytes
$ export OMP_STACKSIZE
```

2.3.1 Detecting Stack Overflow

To detect stack overflow, compile your C, C++, or Fortran program with the `-xcheck=stkovf` compiler option. The syntax is as follows:

```
-xcheck=stkovf[:detect | :diagnose]
```

If `-xheck=stkovf:detect` is specified, a detected stack overflow error is handled by executing the signal handler normally associated with the error.

If `-xcheck=stkovf:diagnose` is specified, a detected stack overflow error is handled by catching the associated signal and calling `stack_violation`(3C) to diagnose the error. If a stack overflow error is diagnosed, an error message is printed to `stderr`. This is the default behavior if only `-xcheck=stkovf` is specified.

See the `cc`(1), `CC`(1), or `f95`(1) man pages for more information about the `-xcheck=stkovf` compiler option.

2.4 OpenMP Runtime Routines

The section describes the behaviors of certain OpenMP runtime routines when the program is compiled using Oracle Developer Studio compilers.

2.4.1 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()`.

2.4.2 omp_set_schedule()

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

2.4.3 omp_set_max_active_levels()

When `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()`.

2.4.4 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.

2.5 Checking and Analyzing OpenMP Programs

Oracle Developer 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. `dbx` provides several features that are tailored to OpenMP, such as single-stepping into a parallel region; printing of shared, private, and threadprivate variables in a region; printing information about parallel regions and task regions; and keeping track of synchronization events. Refer to *Oracle Developer Studio 12.6: Debugging a Program with dbx* for more information.
- Code Analyzer is a tool that provides static source-code checking as well as runtime memory access checking. Static errors detected include missing `malloc()` return value check, null pointer dereference, missing function return, and the like. Memory access errors detected include unallocated memory read/write, uninitialized memory read, freed memory read/write, and the like. Refer to *Oracle Developer Studio 12.6: Code Analyzer User's Guide* for more information.
- Thread Analyzer is a tool for detecting data races and deadlocks in multithreaded applications. It works on applications written using OpenMP, POSIX threads, Oracle Solaris threads, or a combination of these. Refer to *Oracle Developer Studio 12.6: Thread Analyzer User's Guide* and the `tha`(1) and `libtha`(3) man pages for more information.
- Performance Analyzer is a tool for analyzing application performance. The tool collects performance data based on statistical sampling of call stacks, and shows metrics of performance for functions, callers and callees, source lines, and instructions. The Performance Analyzer provides several features that are useful for understanding OpenMP performance, such as OMP work, OMP wait, and OMP overhead metrics, and user mode and machine mode views of the application. Refer to *Oracle Developer Studio 12.6: Performance Analyzer* and the `collect`(1) and `analyzer`(1) man pages for more information.

♦ ♦ ♦ CHAPTER 3

3 OpenMP Nested Parallelism

This chapter discusses the features of OpenMP nested parallelism.

3.1 OpenMP 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) helper threads. The encountering thread becomes the master of the new team. All team members execute the code in the parallel region. When a thread finishes its work within the parallel region, it waits at an implicit barrier at the end of the parallel region. When all team members have arrived at the barrier, the threads can leave the barrier. The master thread continues execution of user code in the program beyond the end of the parallel construct, while the helper 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 team executing a nested parallel region consists of one thread only (the thread that encountered the nested `parallel` construct). If nested parallelism is enabled, then the new team may consist of more than one thread.

The OpenMP runtime library maintains a pool of helper threads that can be used to work on parallel regions. When a thread encounters a parallel construct and requests a team of more than one thread, the thread will check the pool and grab idle threads from the pool, making them part of the team. The encountering thread might get fewer helper threads than it requests if the pool does not contain a sufficient number of idle threads. When the team finishes executing the parallel region, the helper threads are returned to the pool.

3.2 Control of Nested Parallelism

Nested parallelism can be controlled by setting various environment variables prior to the execution of the program, or by calling the `omp_set_nested()` runtime routine. This section discusses various environment variables that can be used to control nested parallelism.

3.2.1 OMP_NESTED

Nested parallelism can be enabled or disabled by setting the `OMP_NESTED` environment variable. By default, nested parallelism is disabled.

The following example has three levels of nested parallel constructs.

EXAMPLE 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 | 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
```

Running the program with nested parallelism disabled produces the following output:

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

3.2.2 OMP_THREAD_LIMIT

The setting of the `OMP_THREAD_LIMIT` environment variable controls the maximum number of OpenMP threads to use for the whole program. This number includes the initial (or main) thread, as well as the OpenMP helper threads that the OpenMP runtime library creates. By default, the maximum number of OpenMP threads to use for the whole program is 1024 (one initial or main thread and 1023 OpenMP helper threads).

Note that the thread pool consists of only OpenMP helper threads that the OpenMP runtime library creates. The pool does not include the initial (or main) thread or any thread created explicitly by the user's program.

If `OMP_THREAD_LIMIT` is set to 1, then the helper thread pool will be empty and all parallel regions will be executed by one thread (the initial or main thread).

The following example output shows that a parallel region might get fewer helper threads if the pool does not contain a sufficient number of helper threads. The code is the same as that in Example 1, “Nested Parallelism Example,” on page 30, except that the environment variable `OMP_THREAD_LIMIT` is set to 6. 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 helper threads. If `OMP_THREAD_LIMIT` is set to 6, then the pool contains at most 5 helper threads. Therefore, two of the four innermost parallel regions might not be able to get all the helper threads requested. The following example shows one possible result.

```
% setenv OMP_NESTED TRUE
% OMP_THREAD_LIMIT 6
% 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 = 1
Level 3: number of threads in the team = 1
```

3.2.3 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. If not set, the default is 4.

Note that setting this environment variable simply controls 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 sample code creates 4 levels of nested parallel regions.

```
#include <omp.h>
#include <stdio.h>
#define DEPTH 4
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 output shows a possible result from compiling and running the sample code when `DEPTH` is set to 4. Actual results would depend on how the operating system schedules the threads.

```
% setenv OMP_NESTED TRUE
% 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
```

If `OMP_MAX_ACTIVE_LEVELS` is set to 2, then nested parallel regions at nesting depths of 3 and 4 are executed single-threaded. The following example shows a possible result.

```
% setenv OMP_NESTED TRUE
% 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
```

3.3 Calling OpenMP Runtime Routines Within Nested Parallel Regions

This section discusses calls to the following OpenMP runtime routines within nested parallel regions:

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

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 for 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 2 Calls to OpenMP Runtime Routines Within Parallel Regions

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

        /* The following statement will print out:
         *
         * 0: 2 4
```

```
         * 1: 2 6
         *
         * omp_get_num_threads() returns the number
         * of the threads in the team, so it is
         * the same for the two threads in the team.
         */
        printf("%d: %d %d\n", omp_get_thread_num(),
               omp_get_num_threads(),
               omp_get_max_threads());

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

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

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

The following example shows a possible result from running the above 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.
```

3.4 Some Tips for Using Nested Parallelism

- Nested parallel regions provide an immediate way for more threads to participate in the computation.

  For example, suppose you have a program that contains two levels of parallelism and `OMP_NUM_THREADS` is set to 2. Also, suppose your system has four hardware threads and you want to use all four hardware threads to speed up the execution of the program. Just parallelizing any one level will use only two hardware threads. You can use all four hardware threads by enabling nested parallelism.
- Nested 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.
- Nested parallel regions add overhead. If the outer level has enough parallelism 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 and the load is balanced. Suppose you have a system with four hardware threads and want to use all four hardware threads to speed up the execution of this program. In general, using all four threads for the outer parallel region would yield better performance than using two threads for the outer parallel region and using the other two threads as helper threads for the inner parallel regions because nested parallel regions will introduce additional barriers.

♦ ♦ ♦ CHAPTER 4

OpenMP Tasking

This chapter describes the OpenMP tasking model.

4.1 OpenMP Tasking Model

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

4.1.1 OpenMP Task Execution

In OpenMP, an *explicit* task is specified using the `task` construct, which 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 is 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. The thread that executes a task might be different from the thread that originally encountered the task and placed it in the pool.

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

Threads are allowed to suspend the 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.

Task scheduling points are implied at a number of locations, including the following:

- The point immediately following the generation of an explicit task
- After the point of completion of a `task` region
- In a `taskyield` region
- In a `taskwait` region
- At the end of a `taskgroup` region
- In an implicit and explicit `barrier` region

In addition to explicit tasks specified using the `task` construct, the OpenMP specification presents 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. In the latter case, 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*.

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.

4.1.2 OpenMP Task Types

The OpenMP specification defines various types of tasks that the programmer may use to reduce the overhead of tasking.

An *undeferred* is a task for which execution is not deferred with respect to the generating task; that is, the generating task region is suspended until execution of the undeferred task is completed. The undeferred task might not be executed immediately by the encountering thread. It might be placed in a pool and executed at a later time by the encountering thread or by some other thread. Once the execution of the task is completed, the generating task can resume. 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. Execution of the current task region cannot be resumed until the task with the `if` clause is completed.

Unlike an undeferred task, an *included* task is executed immediately by the encountering thread and is not placed in the pool to be executed at a later time. The task's execution is sequentially included in the generating task region. As with undeferred tasks, the generating task is suspended until the execution of the included task is completed, at which point the generating task can resume. An example of an included task is a task that is a descendant of a *final* task.

A *merged* task is a task whose data environment is the same as that of its generating task region. If a `mergeable` clause is present on a `task` directive, 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 descendent tasks to become final and included tasks. When a `final` clause is present on a `task` directive and the `final` clause expression evaluates to *true*, the generated task will be a *final* task.

4.2 OpenMP Data Environment

The `task` directive takes the following data-sharing 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 at the point when the `task` construct is encountered.

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 point when the task construct is encountered.

The OpenMP specification describes how the data-sharing attributes of variables referenced in parallel, task, or work-sharing constructs are determined.

The data-sharing attributes of variables referenced in a construct may be *predetermined*, *explicitly determined*, or *implicitly determined*. Certain variables have predetermined data-sharing attributes; for example, the loop iteration variable in a `parallel for/do` construct is private. 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.

Note - 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.

4.3 Tasking Example

The C/C++ example in this section illustrates how the OpenMP `task` and `taskwait` directives can be used to compute Fibonacci numbers recursively.

In the example, the parallel region is executed by four threads. The `single` region ensures that only one of the threads executes the `print` statement that calls `fib(n)`.

The call to `fib(n)` generates two tasks (indicated by the `task` directives). One of the tasks calls `fib(n-1)` and the other calls `fib(n-2)` The return values of these calls are added together to produce the value returned by `fib(n)`. Each of the calls to `fib(n-1)` and `fib(n-2)` in turn generates two tasks, which are 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 the execution of a final task will generate included and final tasks. Included tasks will be generated when `fib()` is called with the argument *n* = 9, 8, ..., 2. These tasks will be executed immediately by the encountering threads, thus reducing the overhead of placing tasks in the pool.

The `taskwait` directive ensures that the two tasks generated in the same 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 first call to `fib()`, all four threads will participate in the execution of the tasks generated and placed in the pool.

EXAMPLE 3 Computing Fibonacci Numbers Using Tasks

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

#define THRESHOLD 9

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
  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
```

4.4 Task Scheduling Constraints

The OpenMP specification lists several task scheduling constraints which an OpenMP task scheduler must follow.

1. An included task is executed immediately after it is generated.
2. Scheduling of new tied tasks is constrained by the set of task regions that are currently tied to the thread and that are not suspended in a barrier region. If this set is empty, any new tied task may be scheduled. Otherwise, a new tied task may be scheduled only if it is a descendant task of every task in the set.

3. A dependent task shall not be scheduled until its task dependences are fulfilled.
4. When an explicit task is generated by a construct containing an `if` clause for which the expression evaluates to *false* and the previous constraints are already met, the task is executed immediately after it is generated.

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

Constraints 1 and 4 are two cases where an OpenMP task should be executed immediately.

Constraint 2 is for preventing deadlock. In Example 4, “Illustrating Task Scheduling Constraint 2,” on page 42. Tasks A, B, and C are tied tasks. The thread that is executing Task A is about to enter the critical `taskyield` region and the thread has ownership of the lock associated with the critical region. Because `taskyield` is a task scheduling point, the thread executing Task A may choose to suspend Task A and execute another task instead. Suppose Tasks B and C are in the task pool. According to constraint 2, the thread executing Task A cannot execute Task B because Task B is not a descendant of Task A. Only Task C can be scheduled at this point, because Task C is a descendant of Task A.

If Task B were to be scheduled while Task A is suspended, then the thread to which Task A is tied cannot enter the critical region in Task B because the thread already holds the lock associated with that critical region. Therefore, a deadlock occurs. The purpose of constraint 2 is to avoid this kind of deadlock when the code is conforming.

Note that deadlock can also occur if the programmer nests a critical section inside Task C, but that would be a programming error.

EXAMPLE 4 Illustrating Task Scheduling Constraint 2

```
#pragma omp task // Task A
{
    #pragma omp critical
    {
        #pragma omp task // Task C
        {
        }
        #pragma omp taskyield
     }
}

#pragma omp task // Task B
{
    #pragma omp critical
     {
     }
}
```

4.5 Task Dependence

The OpenMP 4.0 specification introduces the `depend` clause on the `task` directive, which enforces additional constraints on the scheduling of tasks. These constraints establish dependences between sibling tasks only. Sibling tasks are OpenMP tasks that are child tasks of the same task region.

When the `in` dependence-type is specified with the `depend` clause, the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` dependence-type list. When the `out` or `inout` dependence-type is specified on the `depend` clause, the generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `in`, `out`, or `inout` dependence-type list.

The following example illustrates task dependence.

EXAMPLE 5 Illustrating the `depend` Clause Synchronizing Only Sibling Tasks

```
% cat -n task_depend_01.c
     1  #include <omp.h>
     2  #include <stdio.h>
     3  #include <unistd.h>
     4
     5  int main()
     6  {
     7      int a,b,c;
     8
     9      #pragma omp parallel
    10      {
    11          #pragma omp master
    12          {
    13              #pragma omp task depend(out:a)
    14              {
    15                  #pragma omp critical
    16                  printf ("Task 1\n");
    17              }
    18
    19              #pragma omp task depend(out:b)
    20              {
    21                  #pragma omp critical
    22                  printf ("Task 2\n");
    23              }
    24
    25              #pragma omp task depend(in:a,b) depend(out:c)
    26              {
```

```
    27                  printf ("Task 3\n");
    28              }
    29
    30              #pragma omp task depend(in:c)
    31              {
    32                  printf ("Task 4\n");
    33              }
    34          }
    35          if (omp_get_thread_num () == 1)
    36            sleep(1);
    37      }
    38      return 0;
    39  }
```

```
% cc -xopenmp -O3 task_depend_01.c
% a.out
Task 2
Task 1
Task 3
Task 4

% a.out
Task 1
Task 2
Task 3
Task 4
```

In this example, Tasks 1, 2, 3, and 4 are all child tasks of the same implicit task region, and so are sibling tasks. Task 3 is a dependent task of Tasks 1 and 2 because of the dependences on the *a* argument specified in the `depend` clauses. Therefore, Task 3 cannot be scheduled until both Tasks 1 and 2 have completed. Similarly, Task 4 is a dependent task of task 3 so Task 4 cannot be scheduled until Task 3 has completed.

Note that the `depend` clause synchronizes sibling tasks only. The following example (Example 6, “Illustrating the depend Clause Not Affecting non-Sibling Tasks,” on page 44) shows a case where the `depend` clause does not affect non-sibling tasks.

EXAMPLE 6 Illustrating the `depend` Clause Not Affecting non-Sibling Tasks

```
% cat -n task_depend_02.c
     1  #include <omp.h>
     2  #include <stdio.h>
     3  #include <unistd.h>
     4
     5  int main()
     6  {
     7      int a,b,c;
```

```
 8
 9      #pragma omp parallel
10      {
11          #pragma omp master
12          {
13              #pragma omp task depend(out:a)
14              {
15                  #pragma omp critical
16                  printf ("Task 1\n");
17              }
18
19              #pragma omp task depend(out:b)
20              {
21                  #pragma omp critical
22                  printf ("Task 2\n");
23
24                  #pragma omp task depend(out:a,b,c)
25                  {
26                    sleep(1);
27                    #pragma omp critical
28                    printf ("Task 5\n");
29                  }
30              }
31
32              #pragma omp task depend(in:a,b) depend(out:c)
33              {
34                  printf ("Task 3\n");
35              }
36
37              #pragma omp task depend(in:c)
38              {
39                  printf ("Task 4\n");
40              }
41          }
42          if (omp_get_thread_num () == 1)
43            sleep(1);
44      }
45      return 0;
46  }
```

```
% cc -xopenmp -O3 task_depend_02.c
% a.out
Task 1
Task 2
Task 3
Task 4
Task 5
```

In this above example, Task 5 is a child task of Task 2 and is not a sibling of Tasks 1, 2, 3 or 4. So, despite the depend clauses referencing the same variables (*a*, *b*, *c*), there is no dependence between Task 5 and Tasks 1, 2, 3, or 4.

4.5.1 Notes About Task Dependence

Note the following tips about task dependence:

- in, out, and inout dependence-types in the depend clause are similar to read and write operations, although the in, out and inout dependence-types are solely for establishing task dependences. They do not indicate any memory access patterns inside task regions. A task having a depend(in:*a*), depend(out:*a*), or depend(inout:*a*) clause may read or write variable *a* inside its region, or may even not access variable *a* at all.
- Having both the if clause and the depend clause on the same task directive can be expensive when the condition of the if clause evaluates to *false*. When a task has an if(*false*) clause, the encountering thread must suspend the current task region until the generated task (the task with the if(*false*) clause) is completed. At the same time, the task scheduler should not schedule the generated task until its task dependences are fulfilled. Because the point immediately following the generation of an explicit task is a task scheduling point, the task scheduler will try to schedule tasks so that the task dependences of the undeferred task are fulfilled. Finding and scheduling the right tasks in the pool may be expensive. In the worst case, it can be as expensive as having a taskwait region.
- List items used in depend clauses of the same task or sibling tasks must indicate identical storage or disjoint storage. Therefore, if array sections appear in depend clauses, make sure that the array sections indicate either identical or disjoint storage.

4.6 Task Synchronization Using taskwait and taskgroup

You can synchronize tasks by using the taskwait or taskgroup directives.

When a thread encounters a taskwait construct, the current task is suspended until all child tasks that it generated before the taskwait region complete execution.

When a thread encounters a taskgroup construct, it commences to execute the taskgroup region. At the end of the taskgroup region, the current task is suspended until all child tasks that it generated in the taskgroup region and all of their descendant tasks complete execution.

Note the difference between taskwait and taskgroup. With taskwait, the current task waits only for its child tasks. With taskgroup, the current task waits not only for the child tasks

generated in the taskgroup but also for all the descendants of those child tasks. The following two examples illustrate the difference.

EXAMPLE 7 taskwait Example

```
% cat -n taskwait.c
     1  #include <omp.h>
     2  #include <stdio.h>
     3  #include <unistd.h>
     4
     5  int main()
     6  {
     7    #pragma omp parallel
     8    #pragma omp single
     9    {
    10      #pragma omp task
    11      {
    12        #pragma omp critical
    13        printf ("Task 1\n");
    14
    15        #pragma omp task
    16        {
    17          sleep(1);
    18          #pragma omp critical
    19          printf ("Task 2\n");
    20        }
    21      }
    22
    23      #pragma omp taskwait
    24
    25      #pragma omp task
    26      {
    27        #pragma omp critical
    28        printf ("Task 3\n");
    29      }
    30    }
    31
    32    return 0;
    33  }
```

EXAMPLE 8 taskgroup Example

```
% cat -n taskgroup.c
     1  #include <omp.h>
     2  #include <stdio.h>
     3  #include <unistd.h>
```

```
 4
 5  int main()
 6  {
 7    #pragma omp parallel
 8    #pragma omp single
 9    {
10      #pragma omp taskgroup
11      {
12        #pragma omp task
13        {
14          #pragma omp critical
15          printf ("Task 1\n");
16
17          #pragma omp task
18          {
19            sleep(1);
20            #pragma omp critical
21            printf ("Task 2\n");
22          }
23        }
24      } /* end taskgroup */
25
26      #pragma omp task
27      {
28        #pragma omp critical
29        printf ("Task 3\n");
30      }
31    }
32
33    return 0;
34  }
```

Although the source codes of `taskwait.c` and `taskgroup.c` are almost the same, `taskwait.c` has a `taskwait` directive at line 23, whereas `taskgroup.c` has a `taskgroup` construct at line 10 that contains Task 1 and Task 2. In both programs, the `taskwait` and `taskgroup` directives synchronize the execution of Task 1 and Task 3; the difference is in whether they synchronize the execution of Task 2 and Task 3.

In the case of `taskwait.c`, Task 2 is not a child task of the implicit task generated by the parallel region to which the `taskwait` region is bound. So, Task 2 does not have to finish by the end of the `taskwait` region. Task 3 can be scheduled before the completion of Task 2.

In the case of `taskgroup.c`, Task 2 is a child task of Task 1, which is generated in the `taskgroup` region. So, Task 2 has to finish by the end of the `taskgroup` region, before Task 3 is encountered and scheduled.

4.7 OpenMP Programming Considerations

Tasking introduces a layer of complexity to an OpenMP program. This section discusses some task-related programming issues to consider.

4.7.1 Threadprivate and Thread-Specific Information

When a thread encounters a task scheduling point, the implementation might suspend the current task and schedule the thread to work on another task. This behavior implies that `threadprivate` variables or other thread-specific information such as the thread number in a task might 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 might 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 might be different from the thread that suspended it. Therefore, both the thread number and the value of `threadprivate` variables before and after the task scheduling point might be different.

4.7.2 OpenMP Locks

Since OpenMP 3.0, locks are owned by tasks, not by threads. Once a lock is acquired by a task, the task owns it, and the same task must release it before the task is completed. However, the `critical` construct remains a thread-based mutual exclusion mechanism.

Because locks are owned by tasks, take extra care when using locks. The following example conforms to the OpenMP 2.5 specification 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 the parallel region and the initial thread are the same. However, the example does not conform to later specifications because the task region that releases the lock *lck* is different from the task region that acquired the lock.

EXAMPLE 9 Using Locks Prior to 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);
}
```

4.7.3 References to Stack Data

A task may reference data on the stack of the routine where the task construct appears (the host routine). Because the execution of a task may be deferred until the next implicit or explicit barrier, a task could execute after the stack of the host routine has already been popped and the stack data overwritten, thereby destroying the stack data referenced by the task.

Be sure to insert the needed synchronizations so that variables are still on the stack when the task references them, as illustrated in the two examples in this section.

In Example 10, “Stack Data: Incorrect Reference,” on page 51, *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 the `work()` routine.

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

For correct results, make sure that `work()` does not return before the task has completed. This can be accomplished by inserting a `taskwait` directive after the `task` construct, as shown in Example 11, “Stack Data: Corrected Reference,” on page 51. Alternatively, *i* can be specified to be `firstprivate` in the `task` construct, instead of `shared`.

EXAMPLE 10 Stack Data: Incorrect Reference

```
#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 11 Stack Data: Corrected Reference

```
#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);
   }

   /* 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 following example, *j* in the `task` construct refers to the *j* in the `sections` construct. Therefore, the task accesses the `firstprivate` copy of *j* in the `sections` construct, which in Oracle Developer Studio is a local variable on the stack of the outlined routine for the `sections` construct.

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

For correct results, make sure that the task is executed before the `sections` region reaches its implicit barrier by inserting a `taskwait` directive after the `task` construct as shown in Example 13, “Sections Data: Corrected Reference,” on page 53. Alternatively, *j* can be specified to be `firstprivate` in the `task` construct, instead of `shared`.

EXAMPLE 12 Sections Data: Incorrect Reference

```
#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);
          }
        }
     } /* Implicit barrier for sections */
  } /* Implicit barrier for parallel */

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

EXAMPLE 13 Sections Data: Corrected Reference

```
#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
         }
      } /* Implicit barrier for sections */
   }/* Implicit barrier for parallel */

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

♦ ♦ ♦ CHAPTER 5

Processor Binding (Thread Affinity)

This chapter describes processor binding.

5.1 Processor Binding Overview

With processor binding (also called *thread affinity*), the program instructs the operating system that a thread in the program should run on the same place on the machine throughout its execution, and should not be moved to other places. A *place* in this context refers to some grouping of sockets, cores, or hardware threads.

Processor binding can improve the performance of 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.

A computer system can be viewed as a hierarchy of sockets, cores, and hardware threads. Each socket contains one or more cores, and each core contains one or more hardware threads.

On Oracle Solaris platforms, the `psrinfo`(1M) command can be used to list available hardware threads. On Linux platforms, the text file `/proc/cpuinfo` provides information about available hardware threads.

When the operating system binds a thread to a processor, the thread will in effect be bound to a specific hardware thread or to a group of hardware threads.

To control the binding of OpenMP threads to processors, you can use the OpenMP 4.0 environment variables, `OMP_PLACES` and `OMP_PROC_BIND`. Alternatively, you can use the Oracle-specific environment variable `SUNW_MP_PROCBIND`. These two sets of environment variables should not be mixed. The environment variables are described in "5.2 `OMP_PLACES` and `OMP_PROC_BIND`" on page 56.

Note - The OpenMP environment variables described in this chapter control the binding of OpenMP threads only (that is, any user threads recorded in the OpenMP runtime library, as well as helper threads that the library created). The environment variables do not control the binding of other user threads. The library records a user thread if the user thread encounters an OpenMP construct or calls an OpenMP runtime routine.

5.2 OMP_PLACES and OMP_PROC_BIND

OpenMP 4.0 provides the OMP_PLACES and OMP_PROC_BIND environment variables to specify how the OpenMP threads in a program are bound to processors. These two environment variable are often used in conjunction with each other. OMP_PLACES is used to specify the *places* on the machine to which the threads are bound. OMP_PROC_BIND is used to specify the *binding policy* (*thread affinity policy*) which prescribes how the threads are assigned to places. Setting OMP_PLACES alone does not enable binding. You also need to set OMP_PROC_BIND.

According to the OpenMP specification, the value of OMP_PLACES can be one of two types of values: either an abstract name describing a set of places (threads, cores, or sockets), or an explicit list of places described by non-negative numbers. Intervals can also be used to define places using the <lowerbound> : <length> : <stride> notation to represent the following list of numbers: "<lower-bound>, <lower-bound> + <stride>, …, <lower-bound> + (<length>-1) *<stride>". When <stride> is omitted, a unit stride is assumed. If OMP_PLACES is not set, then the default value is cores.

EXAMPLE 14 One hardware thread in each place

```
% OMP_PLACES="{0:1}:8:32"

{0:1} defines a place which has one hardware thread only, namely place {0}. The interval
 {0:1}:8:32 is therefore equivalent to
{0}:8:32, which defines 8 places starting with place {0}, and the stride is 32. So the
 list of places is as follows:

Place 0: {0}
Place 1: {32}
Place 2: {64}
Place 3: {96}
Place 4: {128}
Place 5: {160}
Place 6: {192}
Place 7: {224}
```

EXAMPLE 15 Two hardware threads in each place

```
% OMP_PLACES="{0:2}:32:8"

{0:2} defines a place which has two hardware threads, namely place {0,1}. The interval
 {0:2}:24:8 is therefore equivalent to
{0,1}:24:8 which defines 24 places starting with place {0,1}, and the stride is 8. So
 the list of places is as follows:

Place 0: {0,1}
Place 1: {8,9}
Place 2: {16,17}
Place 3: {24,25}
Place 4: {32,33}
Place 5: {40,41}
Place 6: {48,49}
Place 7: {56,57}
Place 8: {64,65}
Place 9: {72,73}
Place 10: {80,81}
Place 11: {88,89}
Place 12: {96,97}
Place 13: {104,105}
Place 14: {112,113}
Place 15: {120,121}
Place 16: {128,129}
Place 17: {136,137}
Place 18: {144,145}
Place 19: {152,153}
Place 20: {160,161}
Place 21: {168,169}
Place 22: {176,177}
Place 23: {184,185}
```

In addition to the two environment variables, `OMP_PLACES` and `OMP_PROC_BIND`, OpenMP 4.0 provides the `proc_bind` clause, which can appear on a `parallel` directive. The `proc_bind` clause is used to specify how the team of threads executing the parallel region are bound to processors.

For details about the `OMP_PLACES` and `OMP_PROC_BIND` environment variables and the `proc_bind` clause, refer to the OpenMP 4.0 specification.

5.2.1 Controlling Thread Affinity in OpenMP 4.0

This section provides details about Section 2.5.2, "Controlling OpenMP Thread Affinity", in the OpenMP 4.0 specification.

When a thread encounters a parallel construct that includes a `proc_bind` clause, the `OMP_PROC_BIND` environment variable is used to determine the policy for binding threads to places. If the parallel construct includes a `proc_bind` clause, then the binding policy specified by the `proc_bind` clause overrides the policy specified by `OMP_PROC_BIND`. Once a thread in the team is assigned to a place, the implementation does not move it to another place.

The *master* thread affinity policy instructs the execution environment to assign every thread in the team to the same place as the master thread. The place partition is not changed by this policy, and each implicit task inherits the *place-partition-var* Internal Control Variable (ICV) of the parent implicit task.

The *close* thread affinity policy instructs the execution environment to assign the threads in the team to places close to the place of the parent thread. The place partition is not changed by this policy, and each implicit task inherits the *place-partition-var* ICV of the parent implicit task. If T is the number of threads in the team, and P is the number of places in the parent's place partition, then the assignment of threads in the team to places is as follows:

- T <= P. The master thread executes on the place of the parent thread, that is, the thread that encountered the parallel construct. The thread with the next smallest thread number executes on the next place in the place partition, and so on, with wrap around with respect to the place partition of the master thread.
- T > P. Each place P will contain S_p threads with consecutive thread numbers, where floor (T/P) <= S_p <= ceiling(T/P). The first S_0 threads (including the master thread) are assigned to the place of the parent thread. The next S_1 threads are assigned to the next place in the place partition, and so on, with wrap around with respect to the place partition of the master thread. When P does not divide T evenly, the exact number of threads in a particular place is implementation defined.

The purpose of the *spread* thread affinity policy is to create a sparse distribution for a team of T threads among the P places of the parent's place partition. A sparse distribution is achieved by first subdividing the parent partition into T subpartitions if T <= P, or P subpartitions if T > P. Then one thread (T <= P) or a set of threads (T > P) is assigned to each subpartition. The *place-partition-var* ICV of each implicit task is set to its subpartition. The subpartitioning is not only a mechanism for achieving a sparse distribution, it also defines a subset of places for a thread to use when creating a nested parallel region. The assignment of threads to places is as follows:

- T <= P. The parent thread's place partition is split into T subpartitions, where each subpartition contains floor(P/T) or ceiling(P/T) consecutive places. A single thread is assigned to each subpartition. The master thread executes on the place of the parent thread and is assigned to the subpartition that includes that place. The thread with the next smallest thread number is assigned to the first place in the next subpartition, and so on, with wrap around with respect to the original place partition of the master thread.
- T > P. The parent thread's place partition is split into P subpartitions, each consisting of a single place. Each subpartition is assigned S_p threads with consecutive thread numbers,

where floor(T/P) <= S_p <= ceiling(T/P). The first S_0 threads (including the master thread) are assigned to the subpartition containing the place of the parent thread. The next S_1 threads are assigned to the next subpartition, and so on, with wrap around with respect to the original place partition of the master thread. When P does not divide T evenly, the exact number of threads in a particular subpartition is implementation defined.

Note - Wrap around is needed if the end of a place partition is reached before all thread assignments are done. For example, wrap around may be needed in the case of *close* and T <= P, if the master thread is assigned to a place other than the first place in the place partition. In this case, thread 1 is assigned to the place after the place of the master place, thread 2 is assigned to the place after that, and so on. The end of the place partition may be reached before all threads are assigned. In this case, assignment of threads is resumed with the first place in the place partition.

5.3 SUNW_MP_PROCBIND

`SUNW_MP_PROCBIND` is a legacy, environment variable specific to Oracle for specifying processor binding. This section describes the values you can set for this variable.

Note - Non-negative integers used as values for `SUNW_MP_PROCBIND` denote logical hardware thread IDs, which might be different from actual hardware thread IDs. Although hardware thread IDs may be consecutive, gaps can occur. For example, on a 16-core SPARC system, the hardware thread IDs could be 0, 1, 2, 3, 8 9, 10, 11, 512, 513, 514, 515, 520, 521, 522, 523. However, logical processor IDs are consecutive integers that start with 0. If the number of hardware threads available in the system is *n*, their logical processor IDs are 0, 1, ..., *n*-1.

The possible values for `SUNW_MP_PROCBIND` are:

- The strings `FALSE`, `TRUE`, `COMPACT`, or `SCATTER` in uppercase or lowercase. For example:

```
% setenv SUNW_MP_PROCBIND "TRUE"
```

  - `FALSE` – The OpenMP threads will not be bound to any processors. This is the default setting.
  - `TRUE` – The OpenMP threads will be bound to hardware threads in a round-robin fashion. The starting hardware thread for the binding is determined by the runtime library with the goal of achieving best performance.
  - `COMPACT` – The OpenMP threads will be bound to hardware threads that are as close together as possible on the system. `COMPACT` allows threads to share data caches and thus improve data locality.

- `SCATTER` – The OpenMP threads will be bound to hardware threads that are far apart. This setting enables higher memory bandwidth for each of the threads.

- Non-negative integer – Denotes the starting logical ID of the hardware threads to which OpenMP threads should be bound. OpenMP threads will be bound to hardware threads in a round-robin fashion starting with the hardware thread with the specified logical ID and wrapping around to the hardware thread with logical ID 0 after binding to the hardware thread with logical ID *n*-1.

  For example:

  ```
  % setenv SUNW_MP_PROCBIND "2"
  ```

- A list of two or more non-negative integers – The OpenMP threads will be bound in a round-robin fashion to hardware threads with the specified logical IDs. Hardware threads with logical IDs other than those specified will not be used.

  The following example binds two threads to hardware thread 2, one to hardware thread 4, and one to hardware thread 6 if four threads are used.

  ```
  % setenv SUNW_MP_PROCBIND "2 2 4 6"
  ```

- Two non-negative integers separated by a hyphen ("-") – The OpenMP threads will be bound in a round-robin fashion to hardware threads in the range that begins with the first logical ID and ends with the second logical ID. The first integer must be less than or equal to the second integer. Hardware threads with logical IDs other than those in the range will not be used.

  For example:

  ```
  % setenv SUNW_MP_PROCBIND "0-6"
  ```

If the value specified for `SUNW_MP_PROCBIND` is invalid, or if an invalid logical ID is given, an error message results and execution of the program will terminate.

If the number of OpenMP threads is greater than the number of hardware threads available, then some hardware threads will have more than one OpenMP thread bound to them. This situation can negatively impact performance.

5.4 Interaction With Processor Sets

A processor set is a subset of the system's processors set aside for exclusive use by specified processes. Processor sets allow the binding of processes to groups of processors, rather than just a single processor. A processor set can be specified using the `psrset`(1M) utility on Oracle Solaris platforms, or the `taskset` command on Linux platforms. Processor binding does not currently respect processor sets specified using the `taskset` command on Linux.

♦ ♦ ♦ CHAPTER 6

Automatic Scoping of Variables

Determining the data-sharing attributes of variables referenced in an OpenMP construct is called *scoping*. This chapter describes automatic scoping of variables.

6.1 Variable Scoping Overview

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 the variable with a *data-sharing attribute clause*, or the compiler automatically applies rules for predetermined and implicitly determined scopes according to Section 2.14.1, "Data-Sharing Attribute Rules", in the OpenMP 4.0 specification. For more information about data-sharing attributes, see Section 2.14.3, "Data-Sharing Attribute Clauses", of the OpenMP 4.0 specification.

Explicitly scoping variables can be tedious and error-prone, especially with large and complicated programs. Moreover, the data-sharing attribute rules can yield some unexpected results. The `task` directive adds to the complexity and difficulty of scoping.

The automatic scoping feature (called *autoscoping*) supported by the Oracle Developer Studio compilers 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 simple user model.

Earlier compiler releases limited autoscoping to variables in a `parallel` construct. Current Oracle Developer Studio compilers extend the autoscoping feature to scalar variables referenced in a `task` construct.

6.2 Autoscoping Data Scope Clause

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

6.2.1 `__auto` Clause

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

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

The `__auto` clause can appear on a `parallel` directive (including `parallel for/do`, `parallel sections`, and Fortran `parallel workshare` directive) or on a `task` directive.

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

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

6.2.2 `default(__auto)` Clause

Syntax: `default(__auto)`

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

The `default(__auto)` clause can appear on a `parallel` directive (including `parallel for/do`, `parallel sections`, and Fortran `parallel workshare` directive), or on a `task` directive.

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

6.3 Scoping Rules for a parallel Construct

When doing 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 for/do loops.

6.3.1 Scoping Rules for Scalar Variables in a parallel Construct

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 construct is free of data race conditions for the threads in the team executing the construct, then the variable is scoped as shared.
- PS2: If in each thread executing the parallel construct the variable is always written before being read by the same thread, then the variable is scoped as private. The variable is scoped as lastprivate if it can be scoped private and it is read before it is written after the parallel construct, and the construct is either a parallel for/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 as reduction with that particular operation type.

6.3.2 Scoping Rule for Arrays in a parallel Construct

- PA1: If the use of the array in the parallel construct is free of data race conditions for the threads in the team executing the construct, then the array is scoped as shared.

6.4 Scoping Rules for Scalar Variables in a task Construct

When doing automatic scoping, the compiler applies the rules described in this section to determine the scope of a scalar variable in a task construct.

Note - In this release of Oracle Developer Studio, autoscoping for tasks does not handle arrays.

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 rules `TS1` - `TS5` in the numeric order. These rules do not apply to variables scoped implicitly by the OpenMP specification, such as loop index variables of `parallel for/do` loops.

- `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 construct 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, 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` construct, and some read in the task might get the value assigned outside the task, and the value assigned to the variable inside the task is not used outside the task, then the variable is autoscoped as `firstprivate`.

6.5 Notes About Autoscoping

Specifying the `_auto(`*list-of-variables*`)` or `default(_auto)` clause on a `parallel` construct doesn't imply that the 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 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, the compiler will scope the variable as `shared` and treat the `parallel` or `task` construct as if an `if(0)` (`if(.false.)` in Fortran) clause was specified. For more information, see "6.6 Restrictions When Using Autoscoping" on page 65.

A variable generally cannot be autoscoped if the use of the variable does not match any of the rules or if 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.

6.6 Restrictions When Using Autoscoping

- To enable autoscoping, the program must be compiled with the `-xopenmp` option at an optimization level of `-xO3` or higher. Autoscoping is not enabled if the program is compiled with `-xopenmp=noopt`.
- Parallel and task autoscoping 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 /* task 1 */
{
  ...
  #pragma omp task /* task 2 */
  {
    ...
  }
 ...
}
```

  In this example, the compiler does not attempt autoscoping for `task 2` because it is lexically nested in `task 1`. The compiler will scope all variables referenced in `task 2` as `shared` and will treat `task 2` as if an `if(0)` (`if(.false.)` in Fortran) 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 the `critical` directive if possible.
- Only synchronizations specified 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.

6.7 Checking the Results of Autoscoping

Detailed autoscoping results are displayed in the *compiler commentary*, The compiler produces an inline commentary when the source is compiled with the `-g` option. The commentary 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 Developer Studio software. For more information, see the `er_src`(1) man page or the *Oracle Developer Studio 12.6: Performance Analyzer*.

For a quick check of autoscoping results, compile with the `-xvpara` option. Compiling with –`xvpara` will give you a general idea about whether autoscoping for a particular construct was successful.

EXAMPLE 16 Checking Autoscoping Results With `-xvpara`

```
% 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 -xvpara -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 when `-xvpara` is specified, as shown in the following example.

EXAMPLE 17 Autoscoping Failure With `-xvpara`

```
% 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 -xvpara -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 autoscoping information appears in the compiler commentary displayed by `er_src`, as shown in the following example.

EXAMPLE 18 Detailed Autoscoping Results Displayed 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
     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 _$d1A2.MAIN_
      along with 0 inner loops
   L1 could not be pipelined because it contains calls
     3.         DO I=1, 100
     4.                 T = Y(I)
     5.                 CALL FOO(X)
     6.                 X(I) = T*T
     7.         END DO
     8. C$OMP END PARALLEL DO
     9.         END
    10.
```

6.8 Autoscoping Examples

This section provides some examples to illustrate how the autoscoping rules work. The rules are described in "6.3 Scoping Rules for a `parallel` Construct" on page 63 and "6.4 Scoping Rules for Scalar Variables in a task Construct" on page 63.

EXAMPLE 19 Complex Example Illustrating Autoscoping Rules

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

In this example, function FOO() contains a parallel construct, which contains a single construct, a worksharing do construct, and a critical construct.

The variables I, N, MM, T, W, M, X, and Y, are used in the parallel construct. The compiler determines the scopes of these variables as follows:

- Scalar I is the loop index of the worksharing do loop. The OpenMP specification mandates that I be scoped private.
- Scalar N is only read in the parallel construct and therefore will not cause a data race, so it is scoped as shared following rule PS1.
- Any thread executing the parallel construct will execute line 14, which sets the value of scalar MM to 0.0. This write will cause a data race, so rule PS1 does not apply. The write happens before any read of MM in the same thread, so MM is scoped as private according to rule PS2.
- Similarly, scalar T is scoped as private.
- Scalar W is read and then written at line 21, so rules PS1 and PS2 do not apply. The addition operation is both associative and communicative, therefore, W is scoped as reduction(+) according to rule PS3.
- Scalar M is written at line 11 which is inside a single construct. The implicit barrier at the end of the single construct ensures that the write at line 11 will not happen concurrently with either the read at line 28 or the write at line 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 construct do not cause a data race, and, following rule S1, M is scoped as shared.
- Array X is only read and not written in the construct, so it is scoped as shared by rule PA1.
- The writes to array Y are distributed among the threads, and no two threads will write to the same element of Y. Because no data race occurs, Y is scoped as shared according to rule PA1.

EXAMPLE 20 QuickSort Example

```
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 shows the following compiler commentary:

      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` construct. Therefore, they are autoscoped as `firstprivate` according to `TS1`.

EXAMPLE 21 Fibonacci Example

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

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

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

   #pragma omp taskwait
```

```
   return x + y;
}

er_src shows the following compiler commentary:

   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 and generated the tasks) exits `fib()`. This implies that *x* and *y* will be accessible while the two tasks are executing. Therefore, *x* and *y* are autoscoped as `shared`, according to `TS2`.

EXAMPLE 22 Example With `single` and `task` Constructs

```
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;
       }
    }

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

  return 0;
}

er_src shows the following compiler commentary:

   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` construct. 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 accessible while `task1` is executing. Therefore, according to `TS2`, *xx* is autoscoped as `shared` on `task1`.

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 accessible 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 23 Example With `task` and `taskwait` Constructs

```
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 shows the following compiler commentary:

   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.   }
```

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 gets the value of *x* defined outside the task (because *xx* is `shared` in the `parallel` construct) 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

7 Scope Checking

Oracle Developer Studio C, C++, and Fortran compilers provide a scope-checking feature whereby the compiler determines whether variables in an OpenMP program are correctly scoped. This chapter describes how to use the scope checking feature.

7.1 Scope Checking Overview

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

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 compiler checks the data-sharing attributes specified by the programmer, the predetermined and implicitly determined data-sharing attributes, and the autoscoping results.

7.2 Using the Scope Checking Feature

To enable scope checking, compile the OpenMP program with the `-xvpara` and `-xopenmp` options. The optimization level should be `-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, provide suggestions for the correct data-sharing attribute clauses to use. For example, warning messages are issued if the compiler detects the following situations:

- Loops are parallelized using OpenMP directives when there are data dependencies between different loop iterations
- OpenMP data-sharing attribute clauses can be problematic if, for example, you specify a variable to be shared in a parallel region when accesses to the variable in the parallel region might cause data race, or you specify a variable to be private in a parallel region when the value assigned to the variable in the parallel region is used after the parallel region.

The following example illustrates scope checking.

EXAMPLE 24 Scope Checking With `-xvpara`

```
% 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 illustrates how potential scoping errors are reported.

EXAMPLE 25 Scoping Errors 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];
  }

  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 example shows some typical errors 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 undefined values. The compiler points out this problem and suggests 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 `parallel for` loop. However, in the above example, *i* is not referenced at all after the loop. The compiler issues a warning and suggests scoping *i* as `private`. Using `private` instead of `lastprivate` can lead to better performance.
4. No data-sharing attribute for variable *b* was explicitly specified. According to the OpenMP specification, `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`.

7.3 Restrictions When Using Scope Checking

- 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 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 the `critical` directive if possible.
- Only synchronizations specified 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.

Note - Scope checking with the `-xvpara` compiler option determines potential problems in the program using static (compile-time) analysis. The Thread Analyzer tool, on the other hand, checks for data races and deadlocks in the program using dynamic (runtime) analysis. Use both of these approaches to detect as many errors as possible in the program.

♦ ♦ ♦ CHAPTER 8

Performance Considerations

Once you have a correct, working OpenMP application, consider its overall performance. This chapter provides some best practices to improve the efficiency and scalability of an OpenMP application.

8.1 General Performance Recommendations

This section describes some general techniques for improving the performance of OpenMP applications.

- Minimize synchronization.
  - Avoid or minimize the use of synchronizations such as `barrier`, `critical`, `ordered`, `taskwait`, 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 worksharing loop in the region that is not followed by any code in the region eliminates one redundant barrier.
  - Use named `critical` sections for fine-grained locking where appropriate so that not all `critical` sections in the program will use the same, default lock.
- Use the `OMP_WAIT_POLICY`, `SUNW_MP_THR_IDLE`, or `SUNW_MP_WAIT_POLICY` environment variables to control the behavior of waiting threads. 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 appropriate 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, then an active wait policy that would make waiting threads spin would give better performance. If an application runs simultaneously with other applications, then a passive wait policy that would put waiting threads to sleep would be better for system throughput.

- Parallelize at the highest level possible, such as outermost 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:

```
#pragma omp parallel
{
   #pragma omp for
   {
      ...
   }
}

#pragma omp parallel
{
   #pragma omp for
   {
      ...
   }
}
```

  A more efficient construct:

```
#pragma omp parallel
{
   #pragma omp for
   {
      ...
   }

   #pragma omp for
   {
      ...
   }
}
```

- Use a `parallel for/do` construct, instead of a worksharing `for/do` construct nested inside a `parallel` construct. For example, this construct is less efficient:

```
#pragma omp parallel
{
   #pragma omp for
   {
      ... statements ...
   }
}
```

This construct is more efficient:

```
#pragma omp parallel for
{
   ... statements ...
}
```

- When possible, merge parallel loops to avoid parallelization overhead. For example, merge the two `parallel for` loops:

```
#pragma omp parallel for
for (i=1; i<N; i++)
   {
     ... statements 1 ...
   }

#pragma omp parallel for
for (i=1; i<N; i++)
   {
     ... statements 2 ...
   }
```

The resulting single `parallel for` loop is more efficient:

```
#pragma omp parallel for
for (i=1; i<N; i++)
   {
     ... statements 1 ...
     ... statements 2 ...
   }
```

- Use `the OMP_PROC_BIND` or `SUNW_MP_PROCBIND` environment variable 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 Chapter 5, “Processor Binding (Thread Affinity)”.
- Use `master` instead of `single` where possible.
  - The `master` directive is implemented as an `if` statement with no implicit barrier: `if (omp_get_thread_num() == 0) {...}`
  - The `single` construct is implemented similarly to other worksharing constructs. Keeping track of which thread reaches `single` first adds additional runtime overhead. Moreover, there is an implicit barrier if `nowait` is not specified, which is less efficient.
- Choose the appropriate loop schedule.

  - The `static` loop schedule requires no synchronization and can maintain data locality when data fits in cache. However, the `static` schedule could lead to load imbalance.
  - The `dynamic` and `guided` loop schedules incur 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 efficient thread-safe memory management. An application could be using `malloc()` and `free()` functions explicitly, or implicitly in the compiler-generated code for dynamic arrays, allocatable arrays, vectorized intrinsics, and so on. The thread-safe `malloc()` and `free()` in the standard C library, `libc.so`, have a high synchronization overhead caused by internal locking. Faster versions can be found in other libraries, such as the `libmtmalloc.so` library. Specify `-lmtmalloc` to link with `libmtmalloc.so`.
- Small data sets could cause OpenMP parallel regions to underperform. Use the `if` clause on the `parallel` construct to specify that the region should be run in parallel only in those cases where some performance gain can be expected.
- Try nested parallelism if your application lacks scalability beyond a certain level. However, use nested parallelism with care as it adds synchronization overhead because the thread team of every nested parallel region has to synchronize at a barrier. Also, nested parallelism may oversubscribe the machine, leading to degraded performance.
- Use `lastprivate` with care, as it has the potential of high overhead.
  - Data needs to be copied from a thread's private memory to shared memory before the return from the region.
  - Extra checks are added for `lastprivate`. For example, the compiled code for a worksharing loop with the `lastprivate` clause checks which thread executes the sequentially last iteration. This imposes extra work at the end of each chunk in the loop, which may add up if there are many chunks.
- Use explicit `flush` with care. A flush causes data to be stored to memory, and subsequent data accesses may require reload from memory, all of which decrease efficiency.

8.2 Avoid False Sharing

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?

In most shared memory multiprocessor computers, each processor has its own local cache. The cache acts as a buffer between slow memory and the high speed registers of the processor. 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 to those around it are satisfied out of the cache until the system determines it is necessary to maintain the coherency between cache and memory.

False sharing occurs when threads on different processors modify variables that reside on the same cache line. This situation is called false sharing (to distinguish it from true sharing) because the threads are not accessing the same variable, but rather are accessing different variables that happen to reside on the same cache line.

When a thread modifies a variable in its cache, the whole cache line on which the variable resides is marked as invalid. If another thread attempts to access a variable on the same cache line, then the modified cache line is written back to memory and the thread fetches the cache line from memory. This occurs because cache coherency is maintained on a cache-line basis and not for individual variables or elements. With false sharing, a thread is forced to fetch a more recent copy of a cache line from memory, even though the variable it is attempting to access has not been modified.

If false sharing occurs frequently, interconnect traffic increases, and the performance and scalability of an OpenMP application suffer significantly. False sharing degrades performance when all of the following conditions occur:

- Shared data is modified by multiple threads
- Multiple threads modify data within the same cache line
- Data is modified very frequently (as in a tight loop)

Note that accessing shared data that is read-only does not lead to false sharing.

8.2.2 Reducing False Sharing

False sharing can typically be detected when accesses to certain variables seem particularly expensive. Careful analysis of 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 using the following techniques:

- Make use of private or threadprivate data as much as possible.
- Use the compiler’s optimization features to eliminate memory loads and stores.

- Pad data structures so that each thread's data resides on a different cache line. The size of the padding is system-dependent, and is the size needed to push a thread's data onto a separate cache line.
- Modify data structures so there is less sharing of data among the threads.

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 *chunk_size* value) can also lead to a reduction in false sharing.

8.3 Oracle Solaris OS Tuning Features

The Oracle Solaris operating system supports features that improve the performance of OpenMP programs. These features include Memory Placement Optimizations (MPO) and Multiple Page Size Support (MPSS).

8.3.1 Memory Placement Optimizations

Shared memory multiprocessor computers contain multiple processors. Each processor can access all of the memory in the machine. In some shared memory multiprocessors, the memory architecture enables each processor to access some areas of memory more quickly than other areas. Therefore, allocating memory close to the processor that accesses it will reduce latency and improve application performance.

The Oracle Solaris operating system introduced the locality group (`lgroup`) abstraction, which is part of the MPO feature. An `lgroup` is a set of processor-like and memory-like devices in which each processor in the set can access any memory in that set within a bounded latency interval. The library `liblgrp.so` exports the `lgroup` abstraction for applications to use for observability and performance tuning. Applications can use the `liblgrp.so` APIs to perform the following tasks:

- Traverse the group hierarchy
- Discover the contents and characteristics of a given `lgroup`
- Affect the thread and memory placement on `lgroups`

By default, the Oracle Solaris operating system attempts to allocate resources for a thread from the thread's home `lgroup`. For example, by default the operating system attempts to schedule a thread to run on processors in the thread's home `lgroup` and allocate the thread's memory in the thread's home `lgroup`.

The following mechanisms can be used to discover and affect thread and memory placement with respect to `lgroups`:

- The `meminfo()` system call can be used to discover memory placement.
- The `lgrp_home()` function can be used to discover thread placement.
- The `lgrp_affinity_set()` function can be used to affect thread and memory placement by setting a thread's affinity for a given `lgroup`.
- The `madvise()` function in the standard C library can be used to advise the operating system that a region of user virtual memory is expected to follow a particular pattern of use. The `MADV_ACCESS` flags passed to `madvise()` are used to affect memory allocation among `lgroups`. For example, calling `madvise()` with the `MADV_ACCESS_LWP` flag advises the operating system that the next thread to touch the specified address range is the thread that will access the memory region the most. The OS places the memory for this range and the thread accordingly.

. For more information about the `madvise()` function, see the `madvise`(3C) man page.

8.3.2 Multiple Page Size Support

The Multiple Page Size Support (MPSS) feature in Oracle Solaris allows an application to use different page sizes for different regions of virtual memory. The default page size on a specific platform can be obtained with the `pagesize` command. The `-a` option on this command lists all the supported page sizes. For details, see the `pagesize`(1) man page.

The Translation Lookaside Buffer (TLB) is a data structure used to map virtual memory addresses to physical memory addresses. Some performance penalty is associated with accessing memory which does not have the virtual-to-physical mapping information available in the TLB. Larger page sizes let the TLB map more physical memory using the fixed number of TLB entries. Larger pages may therefore reduce the cost of virtual-to-physical memory mapping and increase overall system performance.

There are several ways to change the default page size for an application:

- Use the Oracle Solaris command `ppgsz`(1).
- Compile the application with the `-xpagesize`, `-xpagesize_heap`, or the `-xpagesize_stack` options. See the `cc`(1), `CC`(1), or `f95`(1) man pages for details.
- Preload the `mpss.so.1` shared object, which allows the use of environment variables to set the page sizes. See the `mpss.so.1`(1) man page for details.

♦ ♦ ♦ CHAPTER 9

OpenMP Implementation-Defined Behaviors

This chapter documents how certain OpenMP features behave when the program is compiled using Oracle Developer Studio compilers. See Appendix D of the OpenMP 4.0 Specification for a summary of behaviors described as implemenation-defined in the specification.

9.1 OpenMP 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. Sometimes there are faster code sequences that use more loads (or stores). Therefore, the compiler and the runtime system might need to use multiple loads (or stores) to access the variable.

When a memory update is on a bit-field variable, the minimum size at which the memory update may also read and write back adjacent variables that are part of another variable such as array or structure elements is the same as that required by the base language. When a memory update is on a variable that is not a bit-field variable, the update will not read and write back adjacent variables that are part of another variable such as array or structure elements.

9.2 OpenMP Internal Control Variables

The following internal control variables are defined by the implementation:

- *bind-var*: Controls the binding of threads to places. The initial value of *bind-var* is `FALSE`.
- *default-device-var*: Controls the default target device. The initial value of *default-device-var* is 0 (the host device).
- *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.

- *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).
- *max-active-levels-var*: Controls the maximum number of nested active parallel regions. The initial value of *max-active-levels-var* is 4.
- *nthreads-var*: Controls the number of threads requested for encountered parallel regions. The initial value of *nthreads-var* is equal to the number of cores, capped at 32.
- *place-partition-var*: Controls the place partition available to the execution environment for encountered parallel regions. The initial value of *place-partition-var* is `cores`.
- *run-sched-var*: Controls the schedule that the `schedule(runtime)` clause uses for loop regions. The initial value of *run-sched-var* is `static` with no chunk size.
- *stacksize-var*: Controls the stack size for threads that the OpenMP implementation creates (also known as helper threads). The initial value of *stacksize-var* is 4 Megabytes for 32-bit applications, and 8 Megabytes for 64-bit applications.
- *thread-limit-var*: Controls the maximum number of threads participating in the OpenMP program. The initial value of *thread-limit-var* is 1024.
- *wait-policy-var*: Controls the desired behavior of waiting threads. The initial value of *wait-policy-var* is `PASSIVE`.

9.3 Dynamic Adjustment of the Number of Threads

This 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 a *false* 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 4.0 Specification. In exceptional situations, such as lack of system resources, the number of threads supplied will be less than that described in Algorithm 2.1.

If the implementation cannot supply the requested number of threads and dynamic adjustment of the number of threads is enabled, then program execution will continue with the smaller number of threads. 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.

If the implementation cannot supply the requested number of threads and dynamic adjustment of the number of threads is disabled, then the program will issue an error message and the program will stop execution.

9.4 OpenMP 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.

9.5 OpenMP Constructs

`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.

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

`atomic` The implementation handles all `atomic` directives by enclosing the target statement or structured block 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 memory locations.

9.6 Processor Binding (Thread Affinity)

The OpenMP 4.0 specification defines the term *processor* as an implementation defined hardware unit on which one or more OpenMP threads can execute.

In this implementation, the term *processor* is defined as the smallest hardware execution unit on which one or more OpenMP threads can be scheduled, bound, and executed, as documented in the `processor_bind`(2) Oracle Solaris man page. Synonyms for *processor* include CPU, virtual processor, and hardware thread. For clarity, the term hardware thread is used consistently in this manual.

In this implementation, the precise definitions of the abstract names `threads`, `cores`, and `sockets` used with the `OMP_PLACES` environment variable are follows:

- `threads` refers to the hardware threads on the machine.

- `cores` refers to the physical cores on the machine.
- `sockets` refers to the physical sockets (processor chips) on the machine.

For more information, see "5.2 `OMP_PLACES` and `OMP_PROC_BIND`" on page 56. The implementation-defined behaviors of Oracle Developer Studio that are related to OpenMP 4.0 thread affinity are as follows:

- With the *close* thread binding policy, when T > P and P does not divide T evenly, the assignment of threads to places is as follows: First, each of the P places is assigned S = floor (T/P) threads; the IDs of the threads assigned to a place are a contiguous subset of the thread IDs in the team. Second, each of the first T - (P*S) places (starting with the place of the parent thread, and with wrap around) is assigned one additional thread.
- With the *spread* thread binding policy, when T > P and P does not divide T evenly, the assignment of threads to subpartitions is as follows: First, each of the P subpartitions is assigned S = floor(T/P) threads; the IDs of the threads assigned to a subpartition are a contiguous subset of the thread IDs in the team. Second, each of the first T - (P*S) subpartitions (starting with the subpartition containing the place of the parent thread, and with wrap around) is assigned one additional thread.
- If an affinity request cannot be fulfilled, the process is exited with a nonzero status.
- The numbers specified in the `OMP_PLACES` environment variable refer to hardware thread IDs.
- When creating a place list of *n* elements by appending the number *n* to an abstract name, the place list will consist of *N* consecutive resources beginning at the resource containing the hardware thread on which the main thread is executing at the time the place list is constructed, with wrap around occurring after the last available named resource is reached.
- If more resources are requested than are available on the machine, an error message is issued and the process is exited with a nonzero status. A resource is available if it contains at least one online hardware thread.
- When the execution environment cannot map a numeric value (either explicitly defined or implicitly derived from an interval) within the `OMP_PLACES` list to a hardware thread on the target platform, or if it maps to an unavailable hardware thread, an error message is issued and the process is exited with a nonzero status.
- When the `OMP_PLACES` environment variable is defined using an abstract name, each unit of the resource represented by the abstract name is allocated as a single place. The number of allocated units can be specified by a count *n* whose value is no greater than the total number of available units on the machine. On Oracle Solaris platforms, hardware threads pre-emptively reserved by an administrator using `psrset`(1M) are not considered available. If no available hardware threads remain in the set defined by `OMP_PLACES`, an error message is issued and the process is exited with a nonzero status.
- If the affinity request for a parallel construct cannot be fulfilled (because, for example, the system call to bind an OpenMP thread to a hardware thread fails), the resulting behavior is undefined.

- When using `OMP_PLACES`, intervals may be used to specify places. This implementation assumes that when an interval specifies a sequence of places, *length* is the number of places in the sequence, and *stride* is the number of hardware thread IDs separating successive places in the sequence. If no *stride* value is specified, then unit stride is assumed.
- The numerical identifiers returned by `omp_get_place_proc_ids()` refer to the hardware thread IDs. They are stored in the argument array in ascending numerical order.

9.7 Fortran Issues

The issues described in this section apply to Fortran only.

9.7.1 THREADPRIVATE Directive

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

9.7.2 SHARED Clause

Passing a shared variable to a non-intrinsic procedure may result in the value of the shared variable being copied into temporary storage before the procedure reference, and back out of the temporary storage into the actual argument storage after the procedure reference. Use of intervening temporary storage can occur only when the following three conditions hold regarding an actual argument:

1. The actual argument is one of the following arguments:
   - 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
2. The actual argument is one of the following arguments:
   - An array section
   - An array section with a vector subscript
   - An assumed-shape array
   - A pointer array

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

9.7.3 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.

Index