Sun Studio 12 Update 1: OpenMP API User's Guide

Contents

Preface

The *OpenMP API User's Guide* summarizes the OpenMP Fortran 95, C, and C++ application program interface (API) for building multiprocessing applications. Sun Studio compilers support the OpenMP API. This guide is intended for scientists, engineers, and programmers who have a working knowledge of the Fortran, C, or C++ languages, and the OpenMP parallel programming model. Familiarity with the Solaris operating environment or UNIX in general is also assumed.

Typographic Conventions

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

TABLE P–1   Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file.
		Use `ls -a` to list all files.
		`machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`**
		`Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*.
		A *cache* is a copy that is stored locally.
		Do *not* save the file.
		Note: Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX® system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

TABLE P–2 Shell Prompts

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

Supported Platforms

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

In this document, these x86 related terms mean the following:

- "x86" refers to the larger family of 64–bit and 32–bit x86 compatible products.
- "x64" points out specific 64–bit information about AMD64 or EM64T systems.
- "32–bit x86" points out specific 32–bit information about x86 based systems.

For supported systems, see the hardware compatibility lists.

Accessing Sun Studio Documentation

You can access the documentation at the following locations:

- The documentation is available from the documentation index page at `http://developers.sun.com/sunstudio/documentation`.
- Online help for all components of the IDE is available through the Help menu, as well as through Help buttons on many windows and dialog boxes, in the IDE.
- Online help for the Performance Analyzer is available through the Help menu, as well as through Help buttons on many windows and dialog boxes, in the Performance Analyzer.

The `docs.sun.com` web site (`http://docs.sun.com`) enables you to read, print, and buy Sun Microsystems manuals through the Internet.

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

Documentation in Accessible Formats

The documentation is provided in accessible formats that are readable by assistive technologies for users with disabilities. You can find accessible versions of documentation as described in the following table.

Type of Documentation	Format and Location of Accessible Version
Manuals	HTML at `http://docs.sun.com`
Readmes	HTML on the developer portal at http://developers.sun.com/sunstudio/documentation/ss12u1/
Man pages	HTML on the developer portal at `http://developers.sun.com/sunstudio/documentation/ss12u1/`
Online help	HTML available through the Help menu and Help buttons in the IDE
Release notes	HTML at `http://docs.sun.com`

Resources for Developers

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

- Articles on programming techniques and best practices
- Documentation of the software, as well as corrections to the documentation that is installed with your software
- Tutorials that take you step-by-step through development tasks using Sun Studio tools
- Information on support levels
- User forums
- Downloadable code samples
- New technology previews

The Sun Studio portal is one of a number of additional resources for developers at the Sun Developer Network web site, http://developers.sun.com.

Contacting Technical Support

If you have technical questions about this product that are not answered in this document, go to http://www.sun.com/service/contacting

Sun Welcomes Your Comments

Sun is interested in improving its documentation and welcomes your comments and suggestions. Submit your comments to Sun at this URL: http://www.sun.com/hwdocs/feedback.

Please include the part number of the document in the subject line of your email. For example, the part number for this document is 820-7883–10.

CHAPTER 1

1

Introducing the OpenMP API

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

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

1.1　Where to Find the OpenMP Specifications

The material presented in this manual describes issues specific to the Sun Studio implementation of the OpenMP API. For complete details you must refer to the OpenMP specification documents. This manual makes direct references to sections in the OpenMP 3.0 API specification.

The OpenMP 3.0 specification for C, C++, and Fortran 95 can be found on the official OpenMP website, `http://www.openmp.org`.

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

Latest information about the Sun Studio compiler releases and their implementation of the OpenMP API can be found on the Sun Developer Network portal at `http://developers.sun.com/sunstudio`

1.2   Special Conventions Used Here

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

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

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

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

The terms "directive" and "pragma" are used interchangeably in this manual. The OpenMP specification prefers the term "directive".

2

Compiling and Running OpenMP Programs

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

To run an OpenMP program in a multithreaded environment, you must set the number of threads in the the program to be greater than one. The number of threads is controlled by the **OMP_SET_NUM_THREADS** environment variable, which must be set greater than one prior to running the program. The number of threads can also be set by calling **omp_set_num_threads()** in the program with a value greater than one, or by using the **num_threads** clause with a **PARALLEL** directive.

The latest information regarding Sun Studio compilers and OpenMP can be found on the Sun Developer Network portal, http://developers.sun.com/sunstudio

2.1   Compiler Options To Use

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

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

-xopenmp=parallel	Enables recognition of OpenMP directives.
	The minimum optimization level for **-xopenmp=parallel** is **-xO3**.
	The compiler changes the optimization from a lower level to **-xO3** if necessary, and issues a warning.

`-xopenmp=noopt`	Enables recognition of OpenMP directives.
	The compiler does not raise the optimization level if it is lower than `-xO3`.
	If you explicitly set the optimization level lower than `-xO3`, as in `-xO2 -openmp=noopt` the compiler will issue an error.
	If you do not specify an optimization level with `-openmp=noopt`, the OpenMP directives are recognized, the program is parallelized accordingly, but no optimization is done.
`-xopenmp=stubs`	This option is no longer supported.
	An OpenMP stubs library is provided for users' convenience.
	To compile an OpenMP program that calls OpenMP library routines but ignores the OpenMP directives, compile the program without an `-xopenmp` option, and link the object files with the `libompstubs.a` library.
	For example, `% cc omp_ignore.c -lompstubs`
	Linking with both `libompstubs.a` and the OpenMP runtime library `libmtsk.so` is unsupported and may result in unexpected behavior.
`-xopenmp=none`	Disables recognition of OpenMP directives and does not change the optimization level.

Additional Notes:

- If you do not specify **–xopenmp** on the command line, the compiler assumes **–xopenmp=none** (disabling recognition of OpenMP directives).

- If you specify **–xopenmp** but without a keyword sub-option, the compiler assumes **–xopenmp=parallel**.

- Specifying **-xopenmp=parallel** or **noopt** will define the **_OPENMP** preprocessor token to be YYYYMM (specifically **200805L** for C/C++ and **200805** for Fortran 95).

- When debugging OpenMP programs with **dbx**, compile with **-xopenmp=noopt -g**

- The default optimization level for **-xopenmp** might change in future releases. Compilation warning messages can be avoided by specifying an appropriate optimization level explicitly.

- With Fortran 95, **-xopenmp** , **-xopenmp=parallel**, **-xopenmp=noopt** will add **-stackvar** automatically.

- When compiling and linking an OpenMP program in separate steps, include **-xopenmp** on each of the compile and the link steps.

- Use the **-xvpara** C/C++/Fortran option to display compiler parallelization messages.

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

2.2 OpenMP Environment Variables

The OpenMP specification defines a number of environment variables that control the execution of OpenMP programs. These are summarized in the following table. For details, refer to the OpenMP API Version 3.0 specifications. See also "3.8 Environment Variables" on page 28

TABLE 2–1   OpenMP Environment Variables

Environment Variable	Function
OMP_SCHEDULE	Sets schedule type for **DO**, **PARALLEL DO**, **for**, **parallel for**, directives/pragmas with schedule type **RUNTIME** specified.
	If not set, a default value of **STATIC** is used. *value* is *"type[,chunk]"*
	Example: **setenv OMP_SCHEDULE 'GUIDED,4'**
OMP_NUM_THREADS	Sets the number of threads to use during execution of a parallel region.
	You can override this value by a **NUM_THREADS** clause, or a call to **OMP_SET_NUM_THREADS()**.
	If not set, a default of 1 is used. *value* is a positive integer.
	Example: **setenv OMP_NUM_THREADS 16**
OMP_DYNAMIC	Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions.
	If not set, a default value of **TRUE** is used. *value* is either **TRUE** or **FALSE**.
	Example: **setenv OMP_DYNAMIC FALSE**
OMP_NESTED	Enables or disables nested parallelism.
	value is either **TRUE** or **FALSE**.
	The default is **FALSE**.
	Example: **setenv OMP_NESTED FALSE**
OMP_STACKSIZE	Sets the size of the stack for threads created by OpenMP.
	Size may be specified as a positive integer in Kilobytes, or with a suffix **B, K, M,** or **G**, for Bytes, Kilobytes, Megabytes, or Gigabytes.
	Example: **setenv OMP_STACKSIZE 10M**
OMP_WAIT_POLICY	Set desired policy regarding waiting threads. The value is either **ACTIVE** or **PASSIVE**.
	ACTIVE threads consume processor time while waiting. **PASSIVE** threads do not and may yield the processor or go to sleep.

TABLE 2–1   OpenMP Environment Variables   *(Continued)*

Environment Variable	Function
`OMP_MAX_ACTIVE_LEVELS`	Sets the maximum number of levels of nested active parallel regions. Value is a non-negative integer.
`OMP_THREAD_LIMIT`	Sets the number of threads to use in the whole OpenMP program. Value is a positive integer.

Sun Studio supports additional multiprocessing environment variables that affect execution of OpenMP programs and are not part of the OpenMP specifications. These are summarized in the following table.

TABLE 2–2   Multiprocessing Environment Variables

Environment Variable	Function
`PARALLEL`	For compatibility with legacy programs, setting the `PARALLEL` environment variable has the same effect as setting `OMP_NUM_THREADS`. However, if both `PARALLEL` and `OMP_NUM_THREADS` are set, they must be set to the same value.
`SUNW_MP_WARN`	Controls warning messages issued by the OpenMP runtime library. If set to `TRUE` the runtime library issues warning messages to `stderr`; `FALSE` disables warning messages. The default is `FALSE`.
	The OpenMP runtime library has the ability to check for many common OpenMP violations, such as incorrect nesting and deadlocks. However, runtime checking does add overhead to the execution of the program.
	The runtime library issues warning messages to `stderr` if `SUNW_MP_WARN` is set to `TRUE`. The runtime library will also issue warning messages if the program registers a call-back function to accept warning messages. A program can register a user call-back function by calling the following function:
	`int sunw_mp_register_warn (void (*func)(void *));`
	The address of the call-back function is passed as argument to `sunw_mp_register_warn()`. This function returns 0 upon successfully registering the call-back function, 1 upon failure.
	If the program has registered a call-back function, `libmtsk` will call the registered function passing a pointer to the localized string containing the error message. The memory pointed to is no longer valid upon return from the call-back function.
	Example:
	`setenv SUNW_MP_WARN TRUE`

TABLE 2–2 Multiprocessing Environment Variables *(Continued)*

Environment Variable	Function
SUNW_MP_THR_IDLE	Controls the status of idle threads in an OpenMP program that are waiting at a barrier or waiting for new parallel regions to work on. You can set the value to be one of the following: **SPIN**, **SLEEP**, **SLEEP(** *time***s)**, **SLEEP(***time***ms)**, **SLEEP(** *time***mc)**, where *time* is an integer that specifies an amount of time, and **s, ms**, and **mc** specify the time unit (seconds, milli-seconds, and micro-seconds, respectively). **SLEEP, SLEEP(0), SLEEP(0s), SLEEP(0ms)**, and **SLEEP(0mc)** are all equivalent.
	SPIN specifies that an idle thread should spin while waiting at barrier or waiting for new parallel regions to work on. **SLEEP** without a time argument specifies that an idle thread should sleep immediately. **SLEEP** with a time argument specifies the amount of time a thread should spin-wait before going to sleep.
	The default idle thread status is to sleep after possibly spin-waiting for some amount of time.
	Note that if both **SUNW_MP_THR_IDLE** and **OMP_WAIT_POLICY** are set, they must have consistent values.
	Examples:
| | ```
setenv SUNW_MP_THR_IDLE SPIN
setenv SUNW_MP_THR_IDLE SLEEP
setenv SUNW_MP_THR_IDLE SLEEP(2s)
setenv SUNW_MP_THR_IDLE SLEEP(20ms)
setenv SUNW_MP_THR_IDLE SLEEP(150mc)
``` |
| **SUNW_MP_PROCBIND** | This environment variable works on both Solaris and Linux systems. The **SUNW_MP_PROCBIND** environment variable can be used to bind threads of an OpenMP program to virtual processors on the running system. Performance can be enhanced with processor binding, but performance degradation will occur if multiple threads are bound to the same virtual processor. See for details. |

TABLE 2–2   Multiprocessing Environment Variables   *(Continued)*

Environment Variable	Function
SUNW_MP_MAX_POOL_THREADS	Specifies the maximum size of the thread pool. The thread pool contains only non-user threads that the OpenMP runtime library creates. It does not contain the master thread or any threads created explicitly by the user's program. If this environment variable is set to zero, the thread pool will be empty and all parallel regions will be executed by one thread. The default, if not specified, is 1023. See "4.2 Control of Nested Parallelism" on page 32 for details.
	Note that **SUNW_MP_MAX_POOL_THREADS** specifies the maximum number of *non-user* OpenMP threads to use for the whole program, while **OMP_THREAD_LIMIT** specifies the maximum number of *user and non-user* OpenMP threads for the whole program. If both **SUNW_MP_MAX_POOL_THREADS** and **OMP_THREAD_LIMIT** are set they must have consistent values such that **OMP_THREAD_LIMIT** is set to one more than the value of **SUNW_MP_MAX_POOL_THREADS**.
SUNW_MP_MAX_NESTED_LEVELS	Specifies the maximum depth of active nested parallel regions. Any parallel region that has an active nested depth greater than the value of this environment variable will be executed by only one thread. A parallel region is considered not active if it is an OpenMP parallel region that has a false **IF** clause. The default, if not specified, is 4. See "4.2 Control of Nested Parallelism" on page 32 for details.
	Note that if both **SUNW_MP_MAX_NESTED_LEVELS** and **OMP_MAX_ACTIVE_LEVELS** are set, they must be set to the same value.
STACKSIZE	Sets the stack size for each thread. The value is in kilobytes. The default stack size for a helper thread is 4 Megabytes for 32-bit applications, and 8 Megabytes for 64-bit applications.
	Example:
	setenv STACKSIZE 8192 *sets the thread stack size to 8 Mb*
	The **STACKSIZE** environment variable also accepts numerical values with a suffix of either **B**, **K**, **M**, or **G** for bytes, kilobytes, megabytes, or gigabytes respectively. The default is kilobytes.
	Note that if both **STACKSIZE** and **OMP_STACKSIZE** are set, they must be set to the same value.
SUNW_MP_GUIDED_WEIGHT	Sets the weighting factor used to determine the size of chunks assigned to threads in loops with **GUIDED** scheduling. The value should be a positive floating-point number, and will apply to all loops with **GUIDED** scheduling in the program. If not set, the default value assumed is 2.0.

2.3   Processor Binding

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

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

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

For example, each UltraSPARC IV physical processor has two cores; from the Solaris OS point of view, each of these cores is a virtual processor onto which a thread can be scheduled to run. The UltraSPARC T1 physical processor, on the other hand, has eight cores, and each core can run four simultaneous processing threads; from the Solaris OS point of view, there are 32 virtual processors onto which threads can be scheduled to run. On the Solaris Operating System, the number of virtual processors can be determined by using the **psrinfo**(1M) command. On Linux systems, the file /proc/cpuinfo provides information about available processors.

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

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

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

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

Virtual Processor IDs:

Each virtual processor in a system has a unique processor ID. On Solaris platforms, you can use the **psrinfo**(1M) command to display information about the processors in a system, including their processor IDs. You can use **psrinfo -pv** to list all physical processors in the system and the virtual processors that are associated with each physical processor. Moreover, you can use the **prtdiag**(1M) command to display system configuration and diagnostic information.

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

Logical IDs:

As mentioned above, the non-negative integers specified for **SUNW_MP_PROCBIND** are logical IDs. Logical IDs are consecutive integers that start with 0. If the number of virtual processors available in the system is *n*, then their logical IDs are 0, 1, ..., *n*-1, in the order presented by **psrinfo**(1M). The following Korn shell script can be used to display the mapping from virtual processor IDs to logical IDs.

```
#!/bin/ksh

NUMV=`psrinfo | fgrep "on-line" | wc -l `
set -A VID  `psrinfo | cut -f1 `

echo "Total number of on-line virtual processors = $NUMV"
echo

let "I=0"
let "J=0"
while [[ $I -lt $NUMV ]]
do
  echo "Virtual processor ID ${VID[I]} maps to logical ID ${J}"
  let "I=I+1"
  let "J=J+1"
done
```

On systems where a single physical processor maps to several virtual processors, it may be useful to know which logical IDs correspond to virtual processors that belong to the same physical processor. The following Korn shell script can be used with later Solaris releases to display this information.

```
#!/bin/ksh

NUMV= `psrinfo | grep "on-line" | wc -l `
set -A VLIST  `psrinfo | cut -f1 `
set -A CHECKLIST  `psrinfo | cut -f1 `

let "I=0"
```

```
while [ $I -lt $NUMV ]
do
  let "COUNT=0"
  SAMELIST="$I"

  let "J=I+1"

  while [ $J -lt $NUMV ]
  do
    if [ ${CHECKLIST[J]} -ne -1 ]
    then
      if [ `psrinfo -p ${VLIST[I]} ${VLIST[J]} ` = 1 ]
      then
        SAMELIST="$SAMELIST $J"
        let "CHECKLIST[J]=-1"
        let "COUNT=COUNT+1"
      fi
    fi
    let "J=J+1"
  done

  if [ $COUNT -gt 0 ]
  then
    echo "The following logical IDs belong to the same physical processor:"
    echo "$SAMELIST"
    echo " "
  fi

  let "I=I+1"
done
```

Interpreting the Value Specified for `SUNW_MP_PROCBIND`:

If the value specified for `SUNW_MP_PROCBIND` is `TRUE`, then the threads will be bound to virtual processors in a round-robin fashion, starting with the processor whose logical ID is 0. (Specifying `TRUE` is equivalent to specifying the value 0 for `SUNW_MP_PROCBIND`.)

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

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

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

If the value specified for **SUNW_MP_PROCBIND** does not conform to one of the forms described above, or if an invalid logical ID is given, then an error message will be emitted and execution of the program will terminate.

Note that the number of threads created by the microtasking library, libmtsk, depends on environment variables, API calls in the user's program, and the **num_threads** clause. **SUNW_MP_PROCBIND** specifies the logical IDs of virtual processors to which the threads should be bound. Threads will be bound to that set of processors in a round-robin fashion. If the number of threads used in the program is less than the number of logical IDs specified by **SUNW_MP_PROCBIND**, then some virtual processors will not be used by the program. If the number of threads is greater than the number of logical IDs specified by **SUNW_MP_PROCBIND**, them some virtual processors will have more than one thread bound to them.

Interaction with OS Processor Sets

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

2.4 Stacks and Stack Sizes

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

In general, the default main stack size is 8 megabytes. Compiling Fortran programs with the **f95 -stackvar** option forces the allocation of local variables and arrays on the stack as if they were automatic variables. Use of **-stackvar** with OpenMP programs is implied However, this may lead to stack overflow if not enough memory is allocated for the stack.

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

Each helper thread of an OpenMP program has its own thread stack. This stack mimics the initial (or main) thread stack but is unique to the thread. The thread's **PRIVATE** arrays and variables (local to the thread) are allocated on the thread stack. The default size is 4 megabytes

on 32-bit SPARC V8 and x86 platforms, and 8 megabytes on 64-bit SPARC V9 and x86 platforms. The size of the helper thread stack is set with the **OMP_STACKSIZE** environment variable.

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

demo$ OMP_STACKSIZE=16384             <-Same, using Bourne/Korn shell
demo$ export OMP_STACKSIZE
```

Finding the best stack size might have to be determined by trial and error. If the stack size is too small for a thread to run it may cause silent data corruption in neighboring threads, or segmentation faults. If you are unsure about stack overflows, compile your Fortran, C, or C++ programs with the **-xcheck=stkovf** compiler option to force a segmentation fault on stack overflow. This stops the program before any data corruption can occur.

2.5   Checking the Correctness of OpenMP Programs

The following are various mechanisms and tools that are available for checking the correctness of an OpenMP program.

- You can use the **-xvpara** C/C++/Fortran option to display compiler parallelization messages

- You can use the Sun Studio **dbx** tool to debug C, C++, and Fortran OpenMP programs. An OpenMP program should first be prepared for debugging with dbx by compiling it with the options **-xopenmp=noopt -g** . dbx allows the programmer to single-step into a **PARALLEL** region, set breakpoints in the body of an OpenMP construct, as well as print the values of shared, private, and other. variables for a given thread.

- You can use the **SUNW_MP_WARN** environment variable to enable runtime error checking and the issuing of warning messages by the OpenMP runtime library.

- You can check your OpenMP program for data races and deadlocks by using the Sun Studio Thread Analyzer tool. Refer to the Thread Analyzer manual and the tha(1) man page for details.

3

Implementation-Defined Behaviors

This chapter notes specific behaviors in the OpenMP 3.0 specification that are implementation dependent.

3.1   Task Scheduling Points

Task scheduling points in untied task regions occur at the same points as in tied task regions. So within untied task regions, task scheduling points only appear in the following:

- encountered task constructs
- encountered taskwait constructs
- encountered barrier directives
- implicit barrier regions
- at the end of the untied task region

3.2   Memory Model

There is no guarantee that memory accesses by multiple threads to the same variable without synchronization are atomic with respect to each other. Several implementation-dependent and application-dependent factors affect whether accesses are atomic or not. Some variables might be larger than the largest atomic memory operation on the target platform. Some variables might be mis-aligned or of unknown alignment and the compiler or the run-time system may need to use multiple loads/stores to access the variable. Sometimes there are faster code sequences that use more loads/stores.

3.3   Internal Control Variables

The following internal control variables are defined by the implementation:

- *nthreads-var*: Controls the number of threads requested for encountered parallel regions. The initial value of *nthreads-var* is 1.

- *dyn-var*: Controls whether dynamic adjustment of the number of threads is enabled for encountered parallel regions. The initial value of *dyn-var* is TRUE (that is, dynamic adjustment is enabled).

- *run-sched-var*: Controls the schedule that the runtime schedule clause uses for loop regions. The initial value of *run-sched-var* is static with no chunk size.

- *def-sched-var*: Controls the implementation defined default scheduling of loop regions. The initial value of *def-sched-var* is static with no chunk size.

- *stacksize-var*: Controls the stack size for threads that the OpenMP implementation creates. The initial value of *stacksize-var* is 4 MegaBytes for 32-bit applications and 8 MegaBytes for 64-bit applications.

- *wait-policy-var*: Controls the desired behavior of waiting threads. The initial value of *wait-policy-var* is PASSIVE.

- *thread-limit-var*: Controls the maximum number of threads participating in the OpenMP program. The initial value of thread-limit-var is 1024.

- *max-active-levels-var*: Controls the maximum number of nested active parallel regions. The initial value of *max-active-levels-var* is 4.

3.4   Dynamic Adjustment of Threads

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

When a thread encounters a parallel construct, the number of threads delivered by this implementation is determined according to Algorithm 2.1 pp. 35-36 in the OpenMP 3.0 Specification. In exceptional situations, such as when there is a lack of system resources, the number of threads supplied will be less than described in Algorithm 2.1. In these situations, if **SUNW_MP_WARN** is set to **TRUE** or a callback function is registered via a call to **sunw_mp_register_warn()**, a warning message will be issued.

3.5 Loop Directive

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

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

3.6 Constructs

3.6.1 SECTIONS

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

3.6.2 SINGLE

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

3.6.3 ATOMIC

The implementation replaces all **atomic** directives by enclosing the target statement with a special, named **critical** construct. This will enforce exclusive access between all atomic regions in the program, whether or not these regions update the same or different storage locations.

3.7 Routines

3.7.1 omp_set_schedule

The behavior for the Sun-specific **sunw_mp_sched_reserved** schedule is the same as static with no chunk size.

3.7.2 omp_set_max_active_levels

If **omp_set_max_active_levels()** is called from within an active parallel region, then the call will be ignored. 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 the argument to `omp_set_max_active_levels()` is not a non-negative integer, then the call will be ignored. A warning message will be issued if **SUNW_MP_WARN** is set to TRUE or a callback function is registered by a call to `sunw_mp_register_warn()`.

3.7.3 `omp_get_max_active_levels`

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

3.8 Environment Variables

Variable Name	Implementation
OMP_SCHEDULE	If the schedule type specified for the **OMP_SCHEDULE** is not one of the valid types (**static**, **dynamic**, **guided**, or **auto**), then the environment variable will be ignored, and the default schedule (**static** with no chunk size) 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 the schedule type specified for the **OMP_SCHEDULE** environment variable is **static**, **dynamic**, or **guided**, but the chunk specified size is a negative integer, then the chunk size used will be as follows: For **static**, there will be no chunk size . For **dynamic** and **guided**, the chunk size will be 1. A warning message will be issued if **SUNW_MP_WARN** is set to TRUE or a callback function is registered by a call to `sunw_mp_register_warn()`.
OMP_NUM_THREADS	If the value of the variable is not a positive integer, then the environment variable will be ignored 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 the value of the variable is greater than the number of threads the implementation can support, the following actions are taken:
	- if *dynamic* adjustment of the number of threads is enabled, then the number of threads will be reduced and a warning message will be issued if **SUNW_MP_WARN** is set to TRUE or a callback function is registered by a call to `sunw_mp_register_warn()`.
	- if, on the other hand, dynamic adjustment of the number of threads is disabled, then an error message will be issued and the program will stop.
OMP_DYNAMIC	If the value specified for **OMP_DYNAMIC** is neither TRUE nor FALSE, then the value will be ignored, and the default value TRUE will be used. A warning message will be issued if **SUNW_MP_WARN** is set to TRUE or a callback function is registered by a call to `sunw_mp_register_warn()`.

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

3.9   Fortran Issues

The following apply to Fortran only.

3.9.1   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 currently allocated".

3.9.2   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. This copying into and out of temporary storage can occur only if conditions a, b, and c in OpenMP 3.0 Specification, section 2.9.3.2 on p. 88 hold,namely: .

- The actual argument is one of the following:
  - A shared variable

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

3.9.3 Runtime Library Definitions

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

On 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.

4

Nested Parallelism

This chapter discusses the features of OpenMP nested parallelism.

4.1   The Execution Model

OpenMP uses a fork-join model of parallel execution. When a thread encounters a parallel construct, the thread creates a team composed of itself and some additional (possibly zero) number of threads. The encountering thread becomes the master of the new team. The other threads of the team are called helper threads of the team. All team members execute the code inside the parallel construct. When a thread finishes its work within the parallel construct, it waits at the implicit barrier at the end of the parallel construct. When all team members have arrived at the barrier, the threads can leave the barrier. The master thread continues execution of user code beyond the end of the parallel construct, while the 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 new team created by a thread encountering a parallel construct inside a parallel region consists only of the encountering thread. If nested parallelism is enabled, then the new team may consist of more than one thread.

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

4.2   Control of Nested Parallelism

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

4.2.1   OMP_NESTED

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

The following example has three levels of nested parallel constructs.

EXAMPLE 4–1   Nested Parallelism Example

```
#include <omp.h>
#include <stdio.h>
void report_num_threads(int level)
{
    #pragma omp single
    {
        printf("Level %d: number of threads in the team - %d\n",
                level, omp_get_num_threads());
    }
 }
int main()
{
    omp_set_dynamic(0);
    #pragma omp parallel num_threads(2)
    {
        report_num_threads(1);
        #pragma omp parallel num_threads(2)
        {
            report_num_threads(2);
            #pragma omp parallel num_threads(2)
            {
                report_num_threads(3);
            }
        }
    }
    return(0);
}
```

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

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

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

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

4.2.2   OMP_THREAD_LIMIT

The **OMP_THREAD_LIMIT** environment variable sets the maximum number of OpenMP threads to use for the whole OpenMP program. The defaut number in Sun's implementation is 1024. If this environment variable is set to one, then all parallel regions will be executed by one thread.

The following example shows that a parallel region can get fewer threads if **OMP_THREAD_LIMIT** is set. The code is the same as above. The number of threads needed for all the parallel regions to be active at the same time is 8. If we set **OMP_THREAD_LIMIT** to 6, two of the four inner-most parallel regions may not be able to get all the helper threads they ask for. One possible result is shown below.

```
% setenv OMP_NESTED TRUE
% setenv OMP_THREAD_LIMIT 6
% a.out
Level 1: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 2: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 2
Level 3: number of threads in the team - 1
Level 3: number of threads in the team - 1
```

4.2.3   OMP_MAX_ACTIVE_LEVELS

The environment variable **OMP_MAX_ACTIVE_LEVELS** controls the maximum depth of nested active parallel regions that require more than one thread.

Any active parallel region that has an active nested depth greater than the value of this environment variable will be executed by only one thread. A parallel region is considered active if it it has no **IF** clause, or if it has an **IF** clause that evaluates to *true*. The default maximum number of active nesting levels is 4.

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

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

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

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

```
% setenv 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
```

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

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

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

4.3   Using OpenMP Library Routines Within Nested Parallel Regions

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

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

The 'set' calls affect future parallel regions at the same or inner nesting levels encountered by the calling thread only. They do not affect parallel regions encountered by other threads.

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

EXAMPLE 4–2 Calls to OpenMP Routines Within Parallel Regions

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

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

```
                 * Inner: 6
                 */
                printf("Inner: %d\n", omp_get_num_threads());
            }
            omp_set_num_threads(7);      /* line C */
        }

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

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

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

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

4.4 Some Tips on Using Nested Parallelism

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

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

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

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

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

◆ ◆ ◆ **C H A P T E R 5**

5

Tasking

This chapter describes the OpenMP 3.0 Tasking Model.

5.1   The Tasking Model

OpenMP specification version 3.0 introduced a new feature called *tasking*. Tasking facilitates the parallelization of applications where units of work are generated dynamically, as in recursive structures or *while* loops.

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

When a thread encounters a task construct, it may choose to execute the task immediately or defer its execution until a later time. If task execution is deferred, then the task 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. A thread that executes a task may be different from the thread that originally encountered it.

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

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

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

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

As implemented in the Sun Studio compilers, the above scheduling points are also the task scheduling points for *untied* tasks.

In addition to explicit tasks specified using the task directive, the OpenMP specification version 3.0 introduces the notion of *implicit* tasks. An implicit task is a task generated by the implicit parallel region, or generated when a parallel construct is encountered during execution. The code for each implicit task is the code inside the **parallel** construct. Each implicit task is assigned to a different thread in the team and is tied; that is, an implicit task is always executed from beginning to end by the thread to which it is initially assigned.

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

When an **if** clause is present on a **task** construct and the value of the scalar-expression evaluates to **false**, the thread that encounters the task must immediately execute the task. The **if** clause can be used to avoid the overhead of generating many finely grained tasks and placing them in the conceptual pool.

5.2　Data Environment

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

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

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

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

The data-sharing attributes of variables that are not listed in data attribute clauses of a **task** construct, and are not predetermined according to the OpenMP rules, are implicitly determined as follows:

- **(a)** In a **task** construct, if no **default** clause is present, a variable that is determined to be shared in all enclosing constructs, up to and including the innermost enclosing parallel construct, is *shared*.
- **(b)** In a **task** construct, if no **default** clause is present, a variable whose data-sharing attribute is not determined by rule **(a)** is *firstprivate*.

It follows that:

- **(a)** If a **task** construct is lexically enclosed in a **parallel** construct, then variables that are shared in all scopes enclosing the **task** construct remain shared in the generated task. Otherwise, variables are implicitly determined *firstprivate*.
- **(b)** If a **task** construct is orphaned, then variables are implicitly determined *firstprivate*.

5.3 **TASKWAIT** Directive

Completion of a subset of all explicit tasks bound to a given parallel region may be specified through the use of the **taskwait** directive. The **taskwait** directive specifies a wait on the completion of child tasks generated since the beginning of the current (implicit or explicit) task. Note that the **taskwait** directive specifies a wait on the completion of direct children tasks, not all descendant tasks.

5.4 Tasking Example

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

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

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

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

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

The example is compiled using the Sun Studio 12 Update 1 C++ compiler.

EXAMPLE 5–1   Tasking Example: Computing Fibonacci Numbers

```
#include <stdio.h>
#include <omp.h>
int fib(int n)
{
  int i, j;
  if (n<2)
    return n;
  else
    {
        #pragma omp task shared(i) firstprivate(n)
        i=fib(n-1);

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

        #pragma omp taskwait
        return i+j;
    }
}

int main()
{
  int n = 10;

  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(10) = 55
```

5.5 Programming Considerations

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

5.5.1   THREADPRIVATE and Thread-Specific Information

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

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

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

5.5.2   Locks

OpenMP 3.0 specifies that locks are no longer owned by threads, but by tasks. Once a lock is acquired, the current task owns it, and the same task must release it before task completion.

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

The change in lock ownership requires extra care when using locks. The following program (it appears as Example A.43.1c in the OpenMP Specification version 3.0) is conforming in OpenMP 2.5 because the thread that releases the lock lck in the parallel region is the same thread that acquired the lock in the sequential part of the program (the master thread of a parallel region and the initial thread are the same). However, it is not conforming in OpenMP 3.0, because the task region that releases the lock lck is different from the task region that acquires the lock.

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

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

Chapter 5 • Tasking   43

EXAMPLE 5–2   Example Using Locks: Non-Conformingn in OpenMP 3.0   *(Continued)*

```
int main()
{
  int x;
  omp_lock_t lck;

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

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

5.5.3   References to Stack Data

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

It is the programmer's responsibility to insert the needed synchronizations to ensure that variables are still on the stack when the task references them. Here are two examples.

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

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

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

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

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

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

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

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

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

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

void work()
 {
   int i;

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

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

```
   #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 our second example, j in the **task** construct is shared with the j in the **sections** construct. So the task accesses the **firstprivate** copy of j in the **sections** construct, which (in some implementations, including the Sun Studio compilers) is a local variable on the stack of the outlined routine for the **sections** construct.

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

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

EXAMPLE 5–5 Second Example — Incorrect Version

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

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

   #pragma omp parallel shared(j)
```

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

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

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

EXAMPLE 5–6 Second Example — Corrected Version

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

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

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

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

EXAMPLE 5–6 Second Example — Corrected Version *(Continued)*

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

5.5.4   Data Scoping Attributes

The OpenMP 3.0 specification version 3.0 (in section 2.9) describes how the data-sharing attributes of variables referenced in parallel, task, and worksharing regions are determined.

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

The rules for how the data-sharing attributes of variables are implicitly determined may not always be obvious (see "5.2 Data Environment" on page 40). It is therefore recommended that to avoid any surprises, the programmer explicitly scope all variables that are referenced in a task construct (using the **default**, **shared**, **private**, and **firstprivate** clauses), rather than rely on the OpenMP implicit scoping rules.

◆ ◆ ◆ **CHAPTER 6**

6

Automatic Scoping of Variables

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

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

The Sun Studio C, C++, and Fortran 95 compilers provide an automatic scoping feature. The compilers analyze the execution and synchronization pattern of a parallel region and determine automatically what the scope of a variable should be, based on a set of scoping rules.

6.1 The Autoscoping Data Scope Clause

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

6.1.1 __auto Clause

Syntax:

__auto(*list-of-variables***)**

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

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

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

6.1.2   `default(__auto)` **Clause**

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

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

6.2   Autoscoping Rules

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

These rules do not apply to variables whose scopes are predetermined by the OpenMP specification, such as loop iteration variables of worksharing **DO** or **FOR** loops. Refer to OpenMP 3.0 Specification (section 2.9.1.1, page 78) for a complete listing of variables whose scopes are predetermined.

6.2.1   Autoscoping Rules For Scalar Variables

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

6.2.2   Autoscoping Rules for Arrays

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

6.3 General Comments About Autoscoping

When autoscoping a variable that does not have predetermined scope, the compiler checks the use of the variable against the above rules S1–S3 in the given order if it is a scalar, and against the above rule A1 if it is an array. If a rule matches, the compiler will scope the variable according to the matching rule. If a rule does not match, the compiler tries the next rule. If the compiler is unable to find a match, the compiler gives up attempting to determine the scope of that variable and it is scoped **SHARED** and the binding parallel region is serialized as if an **IF (.FALSE.)** or **if(0)** clause were specified.

There are two reasons why autoscoping fails. One is that the use of the variable does not match any of the rules. The other is that the source code is too complex for the compiler to do a sufficient analysis. Function calls, complicated array subscripts, memory aliasing, and user-implemented synchronizations are some typical causes. (See "6.5 Known Limitations of the Current Implementation" on page 55.)

Autoscoping in C and C++ applies only to basic data types: integer, floating point, and pointer. If a user specifies a structure variable or class variable to be autoscoped, the compiler will scope the variable as shared and the enclosing parallel region will be executed by a single thread.

6.4 Checking the Results of Autoscoping

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

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

A good place to start is to compile with the **-xvpara** option. A warning message will be printed out if autoscoping fails, as shown below.

EXAMPLE 6–1 Compiling With **-vpara**

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

EXAMPLE 6–1   Compiling With **-vpara**   *(Continued)*

```
%f95 -xopenmp -xO3 -vpara -c t.f
"t.f", line 2: Warning: parallel region will be executed
   by a single thread because the autoscoping
   of following variables failed - x
```

Compile with **-vpara** with **f95**, **-xvpara** with **cc**. (This option has not yet been implemented in **CC**.)

EXAMPLE 6–2   Using Compiler Commentary

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

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

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

Source OpenMP region below has tag R1
Variables autoscoped as SHARED in R1: x, y
Variables autoscoped as PRIVATE in R1: t, i
Private variables in R1: i, t
Shared variables in R1: y, x
    2. C$OMP PARALLEL DO DEFAULT(__AUTO)
       <Function: _$d1A2.MAIN_>
Source loop below has tag L1
L1 parallelized by explicit user directive
L1 parallel loop-body code placed in function _$d1A2.MAIN_ along with 0
inner loops
Copy in M-function of loop below has tag L2
L2 scheduled with steady-state cycle count = 3
L2 unrolled 4 times
L2 has 0 loads, 0 stores, 2 prefetches, 0 FPadds, 0 FPmuls, and 0 FPdivs
per iteration
L2 has 1 int-loads, 1 int-stores, 4 alu-ops, 1 muls, 0 int-divs and 1
```

EXAMPLE 6–2   Using Compiler Commentary   *(Continued)*

```
shifts per iteration
     3. DO I=1, 100
     4. T = Y(I)
     5. X(I) = T*T
     6. END DO
     7. C$OMP END PARALLEL DO
     8. END
```

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

EXAMPLE 6–3   A More Complicated Example

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

EXAMPLE 6–3   A More Complicated Example   *(Continued)*

```
34.
35.        FOO = W - M
36.
37.        RETURN
38.        END
```

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

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

Let's see how the compiler uses the autoscoping rules in "6.2 Autoscoping Rules" on page 50to find the appropriate scopes for the variables in the parallel region.

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

- Scalar **I** is the loop iteration variable of the work-sharing **DO** loop. According to the OpenMP specification, the scope of **I** is predetermined to be **PRIVATE**.

- Scalar **N** is only read in the parallel region and therefore will not cause data race, so it is scoped as **SHARED** following rule **S1**.

- Any thread executing the parallel region will execute statement 14, which sets the value of scalar **MM** to 0.0. This write will cause data race, so rule **S1** does not apply. The write happens before any read of **MM** in the same thread, so **MM** is scoped as **PRIVATE** according to rule **S2**.

- Similarly, scalar **T** is scoped as **PRIVATE**.

- Scalar **W** is read and then written at statement 21, so rules **S1** and **S2** do not apply. The addition operation is both associative and communicative, therefore, **W** is scoped as **REDUCTION(+)** according to rule **S3**.

- Scalar **M** is written in statement 11 which is inside a **SINGLE** construct. The implicit barrier at the end of the **SINGLE** construct ensures that the write in statement 11 will not happen concurrently with either the read in statement 28 or the write in statement 29, and the latter two will not happen at the same time because both are inside the same **CRITICAL** construct. No two threads can access **M** at the same time. Therefore, the writes and reads of **M** in the parallel region do not cause a data race, and, following rule **S1**, **M** is scoped **SHARED**.

- Array **X** is only read and not written in the region, so it is scoped as **SHARED** by rule **A1**.

- The writes to array **Y** is distributed among the threads, and no two threads will write to the same elements of **Y**. As there is no data race, **Y** is scoped **SHARED** according to rule **A1**.

6.5 Known Limitations of the Current Implementation

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

- Only OpenMP directives are recognized and used in the analysis. Calls to OpenMP runtime routines are not recognized. For example, if a program uses **OMP_SET_LOCK()** and **OMP_UNSET_LOCK()** to implement a critical section, the compiler is not able to detect the existence of the critical section. Use **CRITICAL** and **END CRITICAL** directives if possible.

- Only synchronizations specified by using OpenMP synchronization directives, such as **BARRIER** and **MASTER**, are recognized and used in the analysis. User-implemented synchronizations, such as busy-waiting, are not recognized.

- Autoscoping is not supported when compiling with **-xopenmp=noopt**.

◆ ◆ ◆ **C H A P T E R 7**

7

Performance Considerations

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

For additional information, see *Solaris Application Programming*, by Darryl Gove, which is available from `http://www.sun.com/books/catalog/solaris_app_programming.xml`

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

7.1 Some General Recommendations

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

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

By default, idle threads will be put to sleep after a certain time out period. It could be that the default time out period is not sufficient for your application, causing the threads to go to

sleep too soon or too late. The `SUNW_MP_THR_IDLE` environment variable can be used to override the default time out period, even up to the point where the idle threads will never be put to sleep and remain active all the time.

- Parallelize at the highest level possible, such as outer **DO/FOR** loops. Enclose multiple loops in one parallel region. In general, make parallel regions as large as possible to reduce parallelization overhead. For example:

This construct is less efficient:

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

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

than this one:

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

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

!$OMP END PARALLEL
```

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

This construct is less efficient:

```
!$OMP PARALLEL
```

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

than this one:

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

- Use **SUNW_MP_PROCBIND** to bind threads to processors. Processor binding, when used along with static scheduling, benefits applications that exhibit a certain data reuse pattern where data accessed by a thread in a parallel region will be in the local cache from a previous invocation of a parallel region. See "2.3 Processor Binding" on page 19.
- Use **MASTER** instead of **SINGLE** wherever possible.
  - The **MASTER** directive is implemented as an **IF**-statement with no implicit **BARRIER**: **IF(omp_get_thread_num() == 0) {...}**
  - The **SINGLE** directive is implemented similar to other worksharing constructs. Keeping track of which thread reached **SINGLE** first adds additional runtime overhead. There is an implicit **BARRIER** if **NOWAIT** is not specified. It is less efficient.

  Choose the appropriate loop scheduling.
  - **STATIC** causes no synchronization overhead and can maintain data locality when data fits in cache. However, **STATIC** may lead to load imbalance.
  - **DYNAMIC,GUIDED** incurs a synchronization overhead to keep track of which chunks have been assigned. And, while these schedules could lead to poor data locality, they can improve load balancing. Experiment with different chunk sizes.

  Use **LASTPRIVATE** with care, as it has the potential of high overhead.
  - Data needs to be copied from private to shared storage upon return from the parallel construct.
  - The compiled code checks which thread executes the logically last iteration. This imposes extra work at the end of each chunk in a parallel **DO**/**FOR**. The overhead adds up if there are many chunks.

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

Small data cases may cause OpenMP parallel loops to underperform. Use the **IF** clause on **PARALLEL** constructs to indicate that a loop should run parallel only in those cases where some performance gain can be expected.

- When possible, merge loops. For example:

merge two loops

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

statements_1

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

statements_2

```
  end do
```

into a single loop

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

statements_1

statements_2

```
  end do
```

- Try nested parallelism if your application lacks scalability beyond a certain level. See "1.2 Special Conventions Used Here" on page 12 for more information about nested parallelism in OpenMP.

7.2 False Sharing And How To Avoid It

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

7.2.1   **What Is** *False Sharing***?**

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

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

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

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

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

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

7.2.2   **Reducing False Sharing**

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

- making use of private data as much as possible;
- utilizing the compiler's optimization features to eliminate memory loads and stores.

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

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

7.3   Solaris OS Tuning Features

New features have been introduced in the Solaris OS that improve the performance of OpenMP programs. These include Memory Placement Optimizations, locality groups, and Multiple Page Size Support.

7.3.1   Locality Groups

The concept of a locality group (*lgroup*) has been introduced in Solaris to represent a set of CPU-like and memory-like hardware resources that are within some latency of each other.

Solaris OS assigns a thread to an lgroup when the thread is created. That lgroup is called the thread's *home lgroup*. Solaris OS runs the thread on the CPUs in the thread's home lgroup and allocates memory from that lgroup whenever possible. If resources from the home lgroup are unavailable, Solaris allocates resources from other lgroups. When a thread has affinity for more than one lgroup, the OS allocates resources from lgroups chosen in order of affinity strength.

The lgroup APIs export the lgroup abstraction for applications to use for observability and performance tuning. A new library, called `liblgrp`, contains the new APIs. Applications can use the 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

For example, the `lgrp_affinity_set()` function sets the affinity that a thread or set of threads have for a given lgroup. The OS uses the lgroup affinities as advice as to where to run a thread and allocate its memory.

The `madvise()` Standard C library function can be used to advise the OS that a region of user virtual memory is expected to follow a particular pattern of use. For example, calling `madvise()` with the `MADV_ACCESS_LWP` argument tells the kernel that the next thread to touch the specified region of memory will access it most heavily, so the kernel will try to allocate the memory and other resources for this range and the thread accordingly. Use of the `madvise()` function can increase system performance when used by programs that have specific knowledge of their access patterns over memory. The kernel needs information about the likely pattern of an application's memory use in order to allocate memory resources efficiently.

For more information about locality groups, refer to the manual *Solaris: Memory and Thread Placement Optimization Developer's Guide*

7.3.2   Multiple Page Size Support

The Multiple Page Size Support (MPSS) feature allows a program to use different page sizes for different regions of virtual memory. The default Solaris page size is relatively small (8KB on UltraSPARC processors and 4KB on AMD64 Opteron processors). The default page size on a specific platform can be obtained with the Solaris OS command: **pagesize** . The **-a** option on this command lists all the supported page sizes. (See the **pagesize**(1) man page for details.)

Applications that suffer from too many TLB misses may experience a performance boost by using a larger page size. TLB misses can be measured using the Sun Performance Analyzer.

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

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

7.4 Analyzing the Performance of an OpenMP Program

The Collector and Performance Analyzer are a pair of Sun Studio tools that can be used to collect and analyze performance data for an application. The Collector tool collects performance data using a statistical method called profiling and by tracing function calls. The Performance Analyzer processes the data recorded by the Collector, and displays various metrics of performance at program, function, OpenMP parallel region, OpenMP task, source-line, and assembly instruction levels. The Performance Analyzer can also display the raw data in a graphical format as a function of time.

See the **collect**(1) and **analyzer**(1) man pages and the *Sun Studio Performance Analyzer* manuals for more details.

◆ ◆ ◆   **APPENDIX A**

A

Placement of Clauses on Directives

The following table relates clauses to directives and pragmas:

TABLE A–1   Directives Where Clauses Can Appear

Clause/Directive	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS	PARALLEL WORKSHARE	TASK
IF	Yes				Yes	Yes	Yes	Yes
PRIVATE	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
SHARED	Yes				Yes	Yes	Yes	Yes
FIRSTPRIVATE	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
LASTPRIVATE		Yes	Yes		Yes	Yes		
DEFAULT	Yes				Yes	Yes	Yes	Yes
REDUCTION	Yes	Yes	Yes		Yes	Yes	Yes	
COPYIN	Yes				Yes	Yes	Yes	
COPYPRIVATE				Yes (1)				
ORDERED		Yes			Yes			
SCHEDULE		Yes			Yes			
NOWAIT		Yes (2)	Yes (2)	Yes (2)				
NUM_THREADS	Yes				Yes	Yes	Yes	
UNTIED								Yes
__AUTO	Yes				Yes	Yes	Yes	

1.   Fortran only: **COPYPRIVATE** can appear on the **END SINGLE** directive.

2. For Fortran, a **NOWAIT** modifier can only appear on the **END DO**, **END SECTIONS**, **END SINGLE**, or **END WORKSHARE** directives.

3. Only Fortran supports **WORKSHARE** and **PARALLEL WORKSHARE**.

APPENDIX B

B

Converting to OpenMP

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

Note – Legacy Sun and Cray parallelization directives are now deprecated and no longer supported by Sun Studio compilers.

B.1 Converting Legacy Fortran Directives

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

B.1.1 Converting Sun-Style Fortran Directives

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

TABLE B–1 Converting Sun Parallelization Directives to OpenMP

Sun Directive	Equivalent OpenMP Directive
C$PAR DOALL *[qualifiers]*	**!$omp parallel do** *[qualifiers]*
C$PAR DOSERIAL	No exact equivalent. You can use: **!$omp master** *loop* **!$omp end master**

TABLE B–1 Converting Sun Parallelization Directives to OpenMP *(Continued)*

Sun Directive	Equivalent OpenMP Directive
`C$PAR DOSERIAL*`	No exact equivalent. You can use: `!$omp master` *loopnest* `!$omp end master`
`C$PAR TASKCOMMON` *block[,...]*	`!$omp threadprivate (/block/[,...])`

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

TABLE B–2 **DOALL** Qualifier Clauses and OpenMP Equivalent Clauses

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

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

TABLE B–3 **SCHEDTYPE** Scheduling and OpenMP **schedule** Equivalents

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

TABLE B–3   **SCHEDTYPE** Scheduling and OpenMP **schedule** Equivalents   *(Continued)*

SCHEDTYPE(spec)	OpenMP schedule(*spec*) Clause Equivalent
SCHEDTYPE(GSS(*m***))**	**schedule(guided,** *m***)**
	Default *m* is 1.

B.1.1.1   Issues Between Sun-Style Fortran Directives and OpenMP

- Scoping of private variables must be declared explicitly with OpenMP. With Sun directives, the compiler uses its own default scoping rules for variables not explicitly scoped in a **PRIVATE** or **SHARED** clause: all scalars are treated as **PRIVATE**, and all array references are **SHARED**. With OpenMP, the default data scope is **SHARED** unless a **DEFAULT(PRIVATE)** clause appears on the **PARALLEL DO** directive. A **DEFAULT(NONE)** clause causes the compiler to flag variables not scoped explicitly. However, see "4.4 Some Tips on Using Nested Parallelism" on page 38 for information on autoscoping in Fortran.

- Since there is no **DOSERIAL** directive, mixing automatic and explicit OpenMP parallelization may have different effects: some loops may be automatically parallelized that would not have been with Sun directives.

- OpenMP provides a richer parallelism model by providing parallel regions and parallel sections. It could be possible to get better performance by redesigning the parallelism strategies of a program that uses Sun directives to take advantage of these features of OpenMP.

B.1.2   Converting Cray-Style Fortran Directives

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

TABLE B–4   OpenMP Equivalents for Cray-Style **DOALL** Qualifier Clauses

Cray DOALL Clause	OpenMP PARALLEL DO Equivalent Clauses
SHARED(*v1,v2,...***)**	**SHARED(***v1,v2,...***)**
PRIVATE(*v1,v2,...***)**	**PRIVATE(***v1,v2,...***)**
AUTOSCOPE	No equivalent. Scoping must be explicit, or with the **DEFAULT** clause, or with the **__AUTO** clause
SAVELAST	No exact equivalent. You can achieve the same effect by using **lastprivate**.
MAXCPUS(n)	**num_threads(***n***).** No exact equivalent.

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

Cray DOALL Clause	OpenMP PARALLEL DO Equivalent Clauses
`GUIDED`	`schedule(guided, `m`)`
	Default m is 1.
`SINGLE`	`schedule(dynamic,1)`
`CHUNKSIZE(n)`	`schedule(dynamic,`n`)`
`NUMCHUNKS(m)`	`schedule(dynamic,`n/m`)` where n is the number of iterations

B.1.2.1   Issues Between Cray-Style Fortran Directives and OpenMP Directives

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

B.2   Converting Legacy C Pragmas

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

The legacy parallelization pragmas are:

TABLE B–5 Converting Legacy C Parallelization Pragmas to OpenMP

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

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

TABLE B–6 **taskloop** Optional Clauses and OpenMP Equivalents

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

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

TABLE B–7 **SCHEDTYPE** Scheduling and OpenMP **schedule** Equivalents

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

B.2.1 Issues Between Legacy C Pragmas and OpenMP

- OpenMP scopes variables declared within a parallel construct as **private**. A **default(none)** clause on a **#pragma omp parallel for** directive causes the compiler to flag variables not scoped explicitly.

- Since there is no **serial_loop** directive, mixing automatic and explicit OpenMP parallelization may have different effects: some loops may be automatically parallelized that would not have been with legacy C directives.

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

Index