Sun Studio 12: OpenMP API User's Guide

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054
U.S.A.

Part No: 819–5270

070518@17466

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

TABLE P–1 Typeface Conventions

Typeface	Meaning	Examples
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `% You have mail.`

`AaBbCc123`	What you type, when contrasted with on-screen computer output	`%` **`su`**
		`Password:`

AaBbCc123	Book titles, new words or terms, words to be emphasized	Read Chapter 6 in the *User's Guide*. These are called *class* options. You *must* be superuser to do this.
AaBbCc123	Command-line placeholder text; replace with a real name or value	To delete a file, type **`rm`** *filename*.

TABLE P–2 Code Conventions

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

Shell Prompts

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

Supported Platforms

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

In this document, 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 that is installed with the software on your local system or network at `file:/opt/SUNWspro/docs/index.html` on Solaris platforms and at `file:/opt/sun/sunstudio12/docs/index.html` on Linux platforms.

  If your software is not installed in the `/opt` directory on a Solaris platform or the `/opt/sun` directory on a Linux platform, ask your system administrator for the equivalent path on your system.

- Most manuals are available from the `docs.sun.com`^{sm} web site. The following titles are available through your installed software on Solaris platforms only:
  - *Standard C++ Library Class Reference*
  - *Standard C++ Library User's Guide*
  - *Tools.h++ Class Library Reference*
  - *Tools.h++ User's Guide*

  The release notes for both Solaris platforms and Linux platforms are available from the `docs.sun.com` web site.

- Online help for all components of the IDE is available through the Help menu, as well as through Help buttons on many windows and dialog boxes, in the IDE.

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

Note – Sun is not responsible for the availability of third-party Web sites mentioned in this document. Sun does not endorse and is not responsible or liable for any content, advertising, products, or other materials that are available on or through such sites or resources. Sun will not be responsible or liable for any actual or alleged damage or loss caused by or in connection with 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. If your software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Type of Documentation	Format and Location of Accessible Version
Manuals (except third-party manuals)	HTML at `http://docs.sun.com`
Third-party manuals: ■ *Standard C++ Library Class Reference* ■ *Standard C++ Library User's Guide* ■ *Tools.h++ Class Library Reference* ■ *Tools.h++ User's Guide*	HTML in the installed software on Solaris platforms through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Readmes	HTML on the developer portal at `http://developers.sun.com/sunstudio/documentation/ss12/`
Man pages	HTML in the installed software through the documentation index at `file:/opt/SUNWspro/docs/index.html` on Solaris platforms, and at `file:/opt/sun/sunstudio12/docs/index.html` on Linux platforms,
Online help	HTML available through the Help menu and Help buttons in the IDE
Release notes	HTML at `http://docs.sun.com`

Related Sun Studio Documentation

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

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

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

Accessing Related Solaris Documentation

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

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

Resources for Developers

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

- Articles on programming techniques and best practices
- A knowledge base of short programming tips
- Documentation of compilers and tools components, as well as corrections to the documentation that is installed with your software
- Information on support levels
- User forums
- Downloadable code samples
- New technology previews

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

Contacting Sun Technical Support

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

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

Sending Your Comments

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

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

Please include the part number (819-5270) of your document in the subject line of your email.

CHAPTER 1

Introducing the OpenMP API

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

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 2.5 API specification.

The OpenMP 2.5 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, `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, `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.

CHAPTER 2

Compiling and Running OpenMP Programs

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

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

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`**. This flag can take an optional keyword argument. (The **`f95`** compiler accepts both **`-xopenmp`** and **`-openmp`** as synonyms.)

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

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

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

Additional Notes:

- If you do not specify **–xopenmp** on the command line, the compiler assumes **–xopenmp=none** (disabling recognition of OpenMP pragmas).
- If you specify **–xopenmp** but without a keyword sub-option, the compiler assumes **–xopenmp=parallel**.
- Specifying **-xopenmp=parallel** or **noopt** will define the **_OPENMP** preprocessor token to be YYYYMM (specifically **200505L** for C/C++ and **200505** 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 option or the **–vpara** Fortran 95 option to display compiler parallelization messages.
- For best performance and functionality, make sure that the latest OpenMP runtime library, l**ibmtsk.so**, is installed on the running system.

2.2 Fortran 95 OpenMP Validation

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

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

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

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

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

2.3 OpenMP Environment Variables

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

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 defined, a default value of **`STATIC`** is used. *value* is *“type[,chunk]”* Example: **`setenv OMP_SCHEDULE 'GUIDED,4'`**
`OMP_NUM_THREADS` *or* **`PARALLEL`**	Sets the number of threads to use during execution of a parallel region. 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. For compatibility with legacy programs, setting the **`PARALLEL`** environment variable has the same effect as setting **`OMP_NUM_THREADS`**. However, if they are both set to different values, the runtime library will issue an error message. Example: **`setenv OMP_NUM_THREADS 16`**
`OMP_DYNAMIC`	Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. If not set, a default value of **`TRUE`** is used. *value* is either **`TRUE`** or **`FALSE`**. Example: **`setenv OMP_DYNAMIC FALSE`**
`OMP_NESTED`	Enables or disables nested parallelism. *value* is either **`TRUE`** or **`FALSE`**. The default is **`FALSE`**. Example: **`setenv OMP_NESTED FALSE`**

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

TABLE 2–2 Multiprocessing Environment Variables

Environment Variable	Function
`SUNW_MP_WARN`	Controls warning messages issued by the OpenMP runtime library. If set to **`TRUE`** the runtime library issues warning messages to **`stderr`**; **`FALSE`** disables warning messages. The default is **`FALSE`**. The OpenMP runtime library has the ability to check for many common OpenMP violations, such as incorrect nesting and deadlocks. Runtime checking does add overhead to the execution of the program. See Chapter 3. 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).
	
	SPIN specifies that an idle thread should spin while waiting at barrier or waiting for new parallel regions to work on. **SLEEP** without a time argument specifies that an idle thread should sleep immediately. **SLEEP** with a time argument specifies the amount of time a thread should spin-wait before going to sleep.
	
	The default idle thread status is to sleep after possibly spin-waiting for some amount of time. **SLEEP, SLEEP(0), SLEEP(0s), SLEEP(0ms)**, and **SLEEP(0mc)** are all equivalent.
	
	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 Solaris systems only. 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 "2.4 Processor Binding on Solaris" on page 19 for details.
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 34 for details.

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

Environment Variable	Function
`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 34 for details.
`STACKSIZE`	Sets the stack size for each thread. The value is in kilobytes. The default thread stack sizes are 4 Mb on 32-bit SPARC V8 and x86 platforms, and 8 Mb on 64-bit SPARC V9 and x86 platforms. Example: **setenv STACKSIZE 8192** *sets the thread stack size to 8 Mb* 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.
`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.

2.4 Processor Binding on Solaris

With processor binding, the programmer instructs the Solaris Operating System that a thread in the program should run on the same processor throughout the execution of the program. (This feature is not available on Linux.)

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.

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

When running under the Solaris OS, 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. You can use the Solaris OS **`psrinfo`**(1M) command to display information about the processors in a system, including their processor IDs. Moreover, you can use the **`prtdiag`**(1M) command to display system configuration and diagnostic information.

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

Virtual processor IDs 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 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.

2.5 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 slave thread. Stacks are temporary memory address spaces used to hold arguments and automatic variables during invocation of a subprogram or function reference.

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

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

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

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

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

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

2.6 Checking OpenMP Programs With the Thread Analyzer

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.

CHAPTER 3

Implementation-Defined Behaviors

This chapter notes specific behaviors in the OpenMP 2.5 specification that are implementation dependent. For last-minute information regarding the latest compiler releases, see the compiler documentation on the Sun Developer Network portal,
`http://developers.sun.com/sunstudio`

3.1 Implementation-Defined Behaviors

- **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.
- **Internal Control Variables**

  The OpenMP runtime library maintains the following internal control variables:

  nthreads-var - stores the number of threads requested for future parallel regions.

  dyn-var - controls whether dynamic adjustment of the number of threads to be used for future parallel regions is enabled.

  nest-var - controls whether nested parallelism is enabled for future parallel regions.

  run-sched-var - stores scheduling information to be used for loop regions using the **`RUNTIME`** schedule clause.

  def-sched-var - stores implementation defined default scheduling information for loop regions.

The runtime library maintains separate copies of each of *nthreads-var*, *dyn-var*, and *nest-var* for each thread. On the other hand, the runtime library maintains one copy of each of *run-sched-var* and *def-sched-var* that applies to all threads.

- **Number of Threads**

  The default value of *nthreads-var* is 1. That is, without an explicit **`num_threads()`** clause, a call to the **`omp_set_num_threads()`** routine, or an explicit definition of the **`OMP_NUM_THREADS`** environment variable, the default number of threads in a team is 1.

  A call to **`omp_set_num_threads()`** modifies the value of *nthreads-var* for the calling thread only and applies to parallel regions at the same or inner nesting level encountered by the calling thread.

  If the requested number of threads is greater than the number of threads an implementation can support or if the value is not a positive integer, then if **`SUNW_MP_WARN`** is set to **`TRUE`** or a callback function is registered by a call to **`sunw_mp_register_warn()`**, a warning message will be issued.

- **Nested Parallelism**

  Nested parallelism is supported. Nested parallel regions can be executed by multiple threads.

  The default value of *nest-var* is false. That is, nested parallelism is disabled by default. Set the **`OMP_NESTED`** environment variable, or call the **`omp_set_nested()`** routine to enable it.

  A call to **`omp_set_nested()`** modifies the value of *nest-var* for the calling thread only and applies to parallel regions at the same or inner nesting level encountered by the calling thread.

  By default, the maximum number of active nesting levels supported is 4. You can change that maximum by setting the environment variable **`SUNW_MP_MAX_NESTED_LEVELS`**.

- **Dynamic Adjustment of Threads**

  The default value of *dyn-var* is true. That is, dynamic adjustment is enabled by default. Set the **`OMP_DYNAMIC`** environment variable, or call the **`omp_set_dynamic()`** routine to disable dynamic adjustment.

  A call to **`omp_set_dynamic()`** modifies the value of *dyn-var* for the calling thread only and applies to parallel regions at the same or inner nesting level encountered by the calling thread.

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

  - the number of threads the user requested
  - 1 + the number of available threads in the pool
  - the number of available virtual processors

  On the other hand, if dynamic adjustment is disabled, then the number of threads in the team will be the minimum of:

  - the number of threads the user requested
  - 1 + the number of available threads in the pool

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

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

- **Loop Scheduling**

  The default value of *def-sched-var* is **STATIC** scheduling. To specify a different schedule for a loop region, use the SCHEDULE clause.

  The default value of *run-sched-var* is also **STATIC** scheduling. You can change the default by setting the **OMP_SCHEDULE** environment variable

- **GUIDED: Determination of Chunk** *Sizes*

  The default chunk size for **SCHEDULE(GUIDED)** when chunksize is not specified is 1. The OpenMP runtime library uses the following formula for computing the chunk sizes for a loop with **GUIDED** scheduling: chunksize = unassigned_iterations / (weight * num_threads) where: unassigned_iterations is the number of iterations in the loop that have not yet been assigned to any thread; weight is a floating-point constant that can be specified by the user at runtime with the **SUNW_MP_GUIDED_WEIGHT** environment variable ("2.3 OpenMP Environment Variables" on page 16). The current default, if not specified, assumes weight is 2.0; num_threads is the number of threads used to execute the loop.Choice of the weight value affects the sizes of the initial and subsequent chunks of iterations assigned to threads in loops, and has a direct affect on load balancing. Experimental results show that the default weight of 2.0 works well generally. However some applications could benefit from a different weight value.

- **Explicitly Threaded Programs**

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

- **Runtime Warnings**

  Setting the **SUNW_MP_WARN** environment variable ("2.3 OpenMP Environment Variables" on page 16) enables runtime validity checking by the OpenMP runtime library.

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

```
% cat bad1.c

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

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

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

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

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

```
% setenv SUNW_MP_WARN TRUE
% ./a.out
WARNING (libmtsk): Environment variable SUNW_MP_WARN is set to
    TRUE. Runtime error checking will be enabled.
At barrier 1.
At barrier 1.
WARNING (libmtsk): Threads at barrier from different directives.
    Thread at barrier from bad1.c:8.
    Thread at barrier from bad1.c:13.
    Possible Reasons:
    Worksharing constructs not encountered by all threads in the
        team in the same order.
    Incorrect placement of barrier directives.
WARNING (libmtsk): Runtime shutting down while some parallel region
    is still active.
```

The C and C++ compilers also provide a function that can be used to register a callback function when errors are detected. When an error is detected, the registered callback function is called and passed a pointer to an error message string as an argument.

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

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

For example:

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

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

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

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

% cc -xopenmp -xO3 bad2.c
% a.out
WARNING (libmtsk): Environment variable SUNW_MP_WARN is set to
    TRUE. Runtime error checking will be enabled.
handle_warn: WARNING (libmtsk): at bad2.c:15. BARRIER is not
    permitted in the dynamic extent of FOR / DO.
```

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

- **Regarding Specific Constructs:**

  `sections` construct

  The structured blocks in a **`sections`** construct are divided among the members of the team executing the sections region, so that the threads execute an approximately equal number of sections.

  `single` construct

  The structured block of a **`single`** construct will be executed by the thread that encounters the single region first.

  `atomic` construct

  This implementation replaces all **`ATOMIC`** directives and pragmas by enclosing the target statement in a **`CRITICAL`** construct.

- **Binding Thread Set for OpenMP Library Routines:**

  `omp_set_num_threads` routine

  When called from within an explicit parallel region, the binding thread set for the **`omp_set_num_threads`** region is the calling thread.

  `omp_get_max_threads` routine

  When called from within an explicit parallel region, the binding thread set for the **`omp_get_max_threads`** region is the calling thread.

  `omp_set_dynamic` routine

  When called from within any explicit parallel region, the binding thread set for the **`omp_set_dynamic`** region is the calling thread only.

  `omp_get_dynamic` routine

  When called from within an explicit parallel region, the binding thread set for the **`omp_get_dynamic`** region is the calling thread only.

  `omp_set_nested` routine

  When called from within an explicit parallel region, the binding thread set for the **`omp_set_nested`** region is the calling thread only.

  `omp_get_nested` routine

  When called from within an explicit parallel region, the binding thread set for the **`omp_get_nested`** region is the calling thread only.

- **Fortran 95-Specific Issues:**

  `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 may be "not currently allocated".

`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 Section 2.8.3.2 of the OpenMP 2.5 Specification hold.

Include and module files

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

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

CHAPTER 4

Nested Parallelism

This chapter discusses the features of OpenMP nested parallelism.

4.1 The Execution Model

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

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

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

4.2 Control of Nested Parallelism

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

4.2.1 `OMP_NESTED`

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

The following example has three levels of nested parallel constructs.

EXAMPLE 4–1 Nested Parallelism Example

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

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

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

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 SUNW_MP_MAX_POOL_THREADS

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

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

The following example shows that a parallel region can get fewer threads if there are not sufficient threads in the pool.The code is the same as above. The number of threads needed for all the parallel regions to be active at the same time is 8. The pool needs to contain at least 7 threads. If we set **SUNW_MP_MAX_POOL_THREADS** to 5, two of the four inner-most parallel regions may not be able to get all the slave threads they ask for. One possible result is shown below.

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

4.2.3 SUNW_MP_MAX_NESTED_LEVELS

The environment variable **SUNW_MP_MAX_NESTED_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 **SUNW_MP_MAX_NESTED_LEVELS** is set to 2, then nested parallel regions at nested depth of 3 and 4 are executed single-threaded.

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

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

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

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

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

```
% setenv SUNW_MP_MAX_NESTED_LEVELS 2
% a.out |sort
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
            {
```

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

```
                /* The following statement will print out
                 *
                 * Inner: 4
                 * Inner: 6
                 */
                printf("Inner: %d\n", omp_get_num_threads());
            }
            omp_set_num_threads(7);      /* line C */
        }

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

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

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

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

4.4 Some Tips on Using Nested Parallelism

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

  For example, suppose you have a program that contains two levels of parallelism and the degree of parallelism at each level is 2. Also, suppose your system has four cpus and you want use all four CPUs to speed up the execution of this program. Just parallelizing any one level will use only two CPUs. You want to parallelize both levels.
- Nesting parallel regions can easily create too many threads and oversubscribe the system. Set **`SUNW_MP_MAX_POOL_THREADS`** and **`SUNW_MP_MAX_NESTED_LEVELS`** appropriately to limit the number of threads in use and prevent runaway oversubscription.
- Creating nested parallel regions adds overhead. If there is enough parallelism at the outer level and the load is balanced, generally it will be more efficient to use all the threads at the outer level of the computation than to create nested 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 slave threads for the inner parallel regions.

◆ ◆ ◆ CHAPTER 5

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.

5.1 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.

5.1.1 `__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.

5.1.2 `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.

5.2 Scoping Rules

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

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

5.2.1 Scoping 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.

5.2.2 Scoping Rules for Arrays

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

5.3 General Comments About Autoscoping

When autoscoping a variable that does not have implicit 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 "5.5 Known Limitations of the Current Implementation" on page 47.)

5.3.1 Autoscoping Rules for Fortran 95:

For Fortran, if a variable is autoscoped by an **`__auto`** or **`default(__auto)`** clause and the variable has a predetermined scope according to the OpenMP Specification, then the compiler will scope it according to that predetermined scope.

For Fortran, the following variables have predetermined scopes:

- Variables and common blocks appearing in **`threadprivate`** directives are *threadprivate*.
- The loop iteration variable in the do-loop of a do or parallel do construct is *private* in that construct.
- Variables used as loop iteration variables in sequential loops in a parallel construct are *private* in the parallel construct.
- Implied **`DO`** or **`FORALL`** indices are *private*.
- Cray pointees inherit the sharing attribute of the storage with which their Cray Fortran pointers are associated.

5.3.2 Autoscoping Rules for C/C++:

For C/C++, if a variable is autoscoped by an **`__auto`** or **`default(__auto)`** clause and the variable has a predetermined scope according to the OpenMP Specification, then the compiler will scope it according to that predetermined scope.

For C/C++, the following variables have predetermined scopes:

- Variables appearing in threadprivate directives are *threadprivate*.

- Variables with automatic storage duration which are declared in a scope inside the construct are *private*.
- Variables with heap allocated storage are *shared*.
- Static data members are *shared*.
- The loop iteration variable in the for-loop of a for or parallel for construct is *private* in that construct.
- Variables with const-qualified type having no mutable member are *shared*.

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.

5.4 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 5–1 Compiling With **`-vpara`**

```
%cat t.f
      INTEGER X(100), Y(100), I, T
C$OMP PARALLEL DO DEFAULT(__AUTO)
      DO I=1, 100
         T = Y(I)
         CALL FOO(X)
         X(I) = T*T
      END DO
C$OMP END PARALLEL DO
      END
%f95 -xopenmp -xO3 -vpara -c t.f
"t.f", line 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 5–2 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 _$d1A2.MAIN_ along with 0
inner loops
Copy in M-function of loop below has tag L2
L2 scheduled with steady-state cycle count = 3
L2 unrolled 4 times
L2 has 0 loads, 0 stores, 2 prefetches, 0 FPadds, 0 FPmuls, and 0 FPdivs
per iteration
L2 has 1 int-loads, 1 int-stores, 4 alu-ops, 1 muls, 0 int-divs and 1
shifts per iteration
     3. DO I=1, 100
     4. T = Y(I)
     5. X(I) = T*T
     6. END DO
     7. C$OMP END PARALLEL DO
     8. END
```

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

EXAMPLE 5–3 A More Complicated Example

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

     W = 0.0

C$OMP PARALLEL DEFAULT(__AUTO)

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

      MM = 0.0

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

C$OMP CRITICAL
      IF ( MM .GT. M ) THEN
         M = MM
      END IF
C$OMP END CRITICAL

C$OMP END PARALLEL

     FOO = W - M

     RETURN
     END
```

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

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

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

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

- Scalar **I** is the loop index of the work-sharing **DO** loop. The OpenMP specification mandates that **I** be scoped **PRIVATE**.
- Scalar **N** is only read in the parallel region and therefore will not cause data race, so it is scoped as **SHARED** following rule **S1**.
- Any thread executing the parallel region will execute statement 14, which sets the value of scalar **MM** to 0.0. This write will cause data race, so rule **S1** does not apply. The write happens before any read of **MM** in the same thread, so **MM** is scoped as **PRIVATE** according to rule **S2**.
- Similarly, scalar **T** is scoped as **PRIVATE**.
- Scalar **W** is read and then written at statement 21, so rules **S1** and **S2** do not apply. The addition operation is both associative and communicative, therefore, **W** is scoped as **REDUCTION(+)** according to rule **S3**.
- Scalar **M** is written in statement 11 which is inside a **SINGLE** construct. The implicit barrier at the end of the **SINGLE** construct ensures that the write in statement 11 will not happen concurrently with either the read in statement 28 or the write in statement 29, 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**.

5.5 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**.

CHAPTER 6

Performance Considerations

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

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

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

6.1 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
```

- On Solaris systems, use **SUNW_MP_PROCBIND** to bind threads to processors. Processor binding, when used along with static scheduling, benefits applications that exhibit a certain data reuse pattern where data accessed by a thread in a parallel region will be in the local cache from a previous invocation of a parallel region. See "2.4 Processor Binding on Solaris" 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.

6.2 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.

6.2.1 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.

6.2.2 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.

6.3 Solaris OS Tuning Features

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

MPO allows the OS to allocate pages close to the processors that access those pages. SunFire E20K, and SunFire E25K systems have different memory latencies within the same UniBoard™ versus between different UniBoards. The default MPO policy, called *first-touch*, allocates memory on the UniBoard containing the processor that first touches the memory. The first-touch policy can greatly improve the performance of applications where data accesses are made mostly to the memory local to each processor with first-touch placement. Compared to a random memory placement policy where the memory is evenly distributed throughout the system, the memory latencies for applications can be lowered and the bandwidth increased, leading to higher performance.

The MPSS feature is supported as of the Solaris 9 OS release, and 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). 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.

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

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

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

APPENDIX A

Placement of Clauses on Directives

The following table relates clauses to directives and pragmas:

TABLE A–1 Pragmas Where Clauses Can Appear

Clause/Pragma	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS	PARALLEL WORKSHARE
IF	Yes				Yes	Yes	Yes
PRIVATE	Yes	Yes	Yes	Yes	Yes	Yes	Yes
SHARED	Yes				Yes	Yes	Yes
FIRSTPRIVATE	Yes	Yes	Yes	Yes	Yes	Yes	Yes
LASTPRIVATE		Yes	Yes		Yes	Yes	
DEFAULT	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
__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

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 40 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