OpenMP API User's Guide

Forte Developer 7

Sun Microsystems, Inc.
4150 Network Circle
Santa Clara, CA 95054 U.S.A.
650-960-1300

Part No. 816-2468-10
May 2002, Revision A

Send comments about this document to: docfeedback@sun.com

Please
Recycle

Adobe PostScript

Contents

Tables

Before You Begin

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

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 are the typographic conventions used in the text and code examples in this manual:

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`	In code examples, what you type, when contrasted with on-screen computer output.	`% su` `Password:`
	In text, identifies tokens in the language, API, or library function names.	**`ATOMIC`** directives
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 and Command-Line Conventions

Code Symbol	Meaning	Notation	Code Example
[ ]	Brackets contain arguments that are optional.	`O[`*`n`*`]`	`O4, O`
{ }	Braces contain a set of choices for 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	%
Bourne shell and Korn shell	$
C shell, Bourne shell, and Korn shell superuser	#

Accessing Forte Developer Development Tools and Man Pages

The Forte Developer product components and man pages are not installed into the standard `/usr/bin/` and `/usr/share/man` directories. To access the Forte Developer compilers and tools, you must have the Forte Developer component directory in your `PATH` environment variable. To access the Forte Developer man pages, you must have the Forte Developer man page directory in your `MANPATH` environment variable.

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

Note – The information in this section assumes that your Forte Developer products are installed in the `/opt` directory. If your product software is not installed in the `/opt` directory, ask your system administrator for the equivalent path on your system.

Accessing Forte Developer Compilers and Tools

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

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

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

```
% echo $PATH
```

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

   If you find the path, your PATH variable is already set to access Forte Developer development tools. If you do not find the path, set your PATH environment variable by following the instructions in the next section.

▼ To Set Your PATH Environment Variable to Enable Access to Forte Developer Compilers and Tools

1. **If you are using the C shell, edit your home .cshrc file. If you are using the Bourne shell or Korn shell, edit your home .profile file.**

2. **Add the following to your PATH environment variable.**

   /opt/SUNWspro/bin

Accessing Forte Developer Man Pages

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

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

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

```
% man dbx
```

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

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

▼ To Set Your MANPATH Environment Variable to Enable Access to Forte Developer Man Pages

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

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

Accessing Forte Developer Documentation

You can access Forte Developer product documentation at the following locations:

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

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

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

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 installed with the product on your local system or network.

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

Product Documentation in Accessible Formats

Forte Developer 7 product 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 product 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 product through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Readmes and man pages	HTML in the installed product through the documentation index at `file:/opt/SUNWspro/docs/index.html`
Release notes	Text file on the product CD at `/cdrom/devpro_v10n1_sparc/release_notes.txt`

Related Forte Developer Documentation

The following table describes related documentation that is available at `file:/opt/SUNWspro/docs/index.html`. If your product 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`.
C User's Guide	Describes the compile-time environment and command-line options for the `cc` compiler.
C++ User's Guide	Describes the compile-time environment and command-line options for the `CC` compiler.
Numerical Computation Guide	Describes issues regarding the numerical accuracy of floating-point computations.

Accessing Related Solaris Documentation

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

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

Sending Your Comments

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

`docfeedback@sun.com`

CHAPTER 1

OpenMP API Summary

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

The OpenMP API is the recommended parallel programming model for all Forte Developer compilers. See Chapter 4 for guidelines on converting legacy Fortran and C parallelization directives to OpenMP.

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

1.1 Where to Find the OpenMP Specifications

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

The Fortran 2.0 and C/C++ 1.0 OpenMP specifications can be found on the official OpenMP website, **`http://www.openmp.org/`**, and are hyper linked to the Forte Developer documentation index installed with the software, at:

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

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 from the block.

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

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

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

1.3 Directive Formats

Only one *directive-name* can be specified on a directive line.

Fortran:

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

C/C++:

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

OpenMP Fortran 2.0

Fixed Format:

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

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

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

OpenMP Fortran 2.0

Free Format:

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

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

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

OpenMP C/C++ 1.0

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

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

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

1.4 Conditional Compilation

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

OpenMP Fortran 2.0

Fixed Format:

`!$` *fortran_95_statement*
`C$` *fortran_95_statement*
`*$` *fortran_95_statement*
`c$` *fortran_95_statement*

The sentinel must start in column 1 with no intervening blanks. With OpenMP compilation enabled, the sentinel is replaced by two blanks. The rest of the line must conform to standard Fortran fixed format conventions, otherwise it is treated as a comment. Example:

OpenMP Fortran 2.0

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

Free Format:

!$ *fortran_95_statement*

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

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

Preprocessor:

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

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

OpenMP C/C++ 1.0

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

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

1.5 **PARALLEL** - Parallel Region Construct

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

OpenMP Fortran 2.0

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

OpenMP C/C++ 1.0

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

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

1.6 Work-Sharing Constructs

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

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

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

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

OpenMP Fortran 2.0

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

The **DO** directive specifies that the iterations of the **DO** loop that immediately follows must be executed in parallel. This directive must appear within a parallel region to be effective.

OpenMP C/C++ 1.0

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

The **for** pragma specifies that the iterations of the *for-loop* that immediately follows must be executed in parallel. This pragma must appear within a parallel region to be effective. The **for** pragma places restrictions on the structure of the corresponding **for** loop, and it must have *canonical shape*:

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

where:

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

1.6.2 SECTIONS

SECTIONS encloses a non-iterative block of code to be divided among threads in the team. Each block is executed once by a thread in the team.

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

OpenMP Fortran 2.0

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

OpenMP C/C++ 1.0

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

1.6.3 SINGLE

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

OpenMP Fortran 2.0

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

OpenMP C/C++ 1.0

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

1.6.4 Fortran **WORKSHARE**

Divides the work of executing the enclosed code block into separate units of work, and causes the threads of the team to share the work such that each unit is executed only once.

OpenMP Fortran 2.0

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

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

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

1.7 Combined Parallel Work-sharing Constructs

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

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

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

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

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

OpenMP Fortran 2.0

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

OpenMP C/C++ 1.0

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

1.7.2 PARALLEL SECTIONS

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

OpenMP Fortran 2.0

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

OpenMP C/C++ 1.0

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

1.7.3 PARALLEL WORKSHARE

Provides a shortcut for specifying a parallel region that contains a single **WORKSHARE** directive. *clause* can be one of the clauses accepted by either the **PARALLEL** or **WORKSHARE** directive.

OpenMP Fortran 2.0

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

There is no C/C++ equivalent.

1.8 Synchronization Constructs

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

1.8.1 MASTER

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

OpenMP Fortran 2.0

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

OpenMP C/C++ 1.0

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

1.8.2 CRITICAL

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

OpenMP Fortran 2.0

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

OpenMP C/C++ 1.0

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

1.8.3 BARRIER

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

OpenMP Fortran 2.0

```
!$OMP BARRIER
```

OpenMP C/C++ 1.0

```
#pragma omp barrier
```

1.8.4 ATOMIC

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

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

OpenMP Fortran 2.0

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

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

x = x operator expression

x = expression operator x

x = intrinsic(x, expr-list)

x = intrinsic(expr-list, x)

where:

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

OpenMP C/C++ 1.0

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

The pragma applies only to the immediately following statement, which must be in one of these forms:

x binop = *expr*

x++

++*x*

x--

--*x*

where:

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

1.8.5 FLUSH

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

OpenMP Fortran 2.0

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

OpenMP C/C++ 1.0

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

1.8.6 ORDERED

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

OpenMP Fortran 2.0

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

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

OpenMP C/C++ 1.0

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

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

1.9 Data Environment Directives

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

1.9.1 THREADPRIVATE

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

See the OpenMP specifications (section 2.6.1 in the Fortran 2.0 specifications, secton 2.7.1 in the C/C++) for the complete details and restrictions.

OpenMP Fortran 2.0

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

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

OpenMP C/C++ 1.0

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

Each variable of *list* must have a file-scope or namespace-scope declaration preceding the pragma.

1.10 OpenMP Directive Clauses

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

1.10.1 Data Scoping Clauses

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

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

There are important restrictions on the use of these scoping clauses. Refer to section 2.6.2 in the Fortran 2.0 specification, and section 2.7.2 in the C/C++ specification for complete details.

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

1.10.1.1 PRIVATE

private(*list***)**

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

1.10.1.2 SHARED

shared(*list***)**

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

1.10.1.3 DEFAULT

Fortran

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

C/C++

```
default(shared | none)
```

Specify scoping attribute for all variables within a parallel region. **THREADPRIVATE** variables are not affected by this clause. If not specified, **DEFAULT(SHARED)** is assumed.

1.10.1.4 FIRSTPRIVATE

firstprivate(*list***)**

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

1.10.1.5 LASTPRIVATE

lastprivate(*list***)**

Variables on the list are **PRIVATE**. In addition, when the **LASTPRIVATE** clause appears on a **DO** or **for** directive, the thread that executes the sequentially last iteration updates the version of the variable before the construct. On a **SECTIONS** directive, the thread that executes the lexically last **SECTION** updates the version of the object it had before the construct.

1.10.1.6 COPYIN

Fortran

COPYIN(*list*)

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

C/C++

copyin(*list*)

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

1.10.1.7 COPYPRIVATE

Fortran

COPYPRIVATE(*list*)

Uses a private variable to broadcast a value, or a pointer to a shared object, from one member of a team to the other members. Variables in list must not appear in a **PRIVATE** or **FIRSTPRIVATE** clause of the **SINGLE** construct specifying **COPYPRIVATE**.

There is no C/C++ equivalent.

1.10.1.8 REDUCTION

Fortran

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

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

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

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

C/C++

reduction(*operator*:*list*)

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

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

1.10.2 Scheduling Clauses

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

There are important restrictions on the use of these scheduling clauses. Refer to section 2.3.1 in the Fortran 2.0 specification, and section 2.4.1 in the C/C++ specification for complete details.

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

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

1.10.2.1 **`STATIC`** Scheduling

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

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

1.10.2.2 **`DYNAMIC`** Scheduling

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

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

1.10.2.3 **`GUIDED`** Scheduling

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

With **`GUIDED`**, the *chunk* size is reduced in an exponentially decreasing manner with each dispatched piece of the iterations. *chunk* specifies the minimum number of iterations to dispatch each time. (The size of the initial chunk of the iterations is implementation dependent; see Chapter 2.). When no *chunk* is specified, it defaults to 1.

1.10.2.4 **`RUNTIME`** Scheduling

`schedule(runtime`)

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

1.10.3 **NUM_THREADS** Clause

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

OpenMP Fortran 2.0

NUM_THREADS(*scalar_integer_expression***)**

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

There is no C/C++ equivalent.

1.10.4 Placement of Clauses on Directives

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

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

TABLE 1-1 Pragmas Where Clauses Can Appear

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

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

1.11 OpenMP Runtime Library Routines

OpenMP provides a set of callable library routines to control and query the parallel execution environment, a set of general purpose lock routines, and two portable timer routines.

1.11.1 Fortran OpenMP Routines

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

OMP_ functions returning **INTEGER(4)** and **LOGICAL(4)** are not intrinsic and must be declared properly, otherwise the compiler will assume **REAL**. Interface declarations for the OpenMP Fortran runtime library routines summarized below are provided by the Fortran include file **omp_lib.h** and a Fortran **MODULE omp_lib**, as described in the Fortran OpenMP 2.0 specifications.

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

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

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

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

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

1.11.2 C/C++ OpenMP Routines

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

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

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

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

1.11.3 Run-time Thread Management Routines

For details, refer to the appropriate OpenMP specifications.

1.11.3.1 OMP_SET_NUM_THREADS

Sets the number of threads to use for subsequent parallel regions

Fortran

```
SUBROUTINE OMP_SET_NUM_THREADS(int_expr)
```

C/C++

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

1.11.3.2 OMP_GET_NUM_THREADS

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

Fortran

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

C/C++

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

1.11.3.3 OMP_GET_MAX_THREADS

Returns maximum value that can be returned by calls to the OMP_GET_NUM_THREADS function.

Fortran

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

C/C++

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

1.11.3.4 OMP_GET_THREAD_NUM

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

Fortran

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

C/C++

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

1.11.3.5 OMP_GET_NUM_PROCS

Return the number of processors available to the program.

Fortran

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

C/C++

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

1.11.3.6 OMP_IN_PARALLEL

Determine if called from within the dynamic extent of a region executing in parallel.

Fortran

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

Returns `.TRUE.` if called within a parallel region, `.FALSE.` otherwise.

C/C++

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

Returns nonzero if called within a parallel region, zero otherwise.

1.11.3.7 OMP_SET_DYNAMIC

Enables or disables dynamic adjustment of the number of available threads. (Dynamic adjustment is enabled by default.)

Fortran

```
SUBROUTINE OMP_SET_DYNAMIC(logical_expr)
```

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

C/C++

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

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

1.11.3.8 OMP_GET_DYNAMIC

Determine whether or not dynamic thread adjustment is enabled.

Fortran

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

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

C/C++

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

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

1.11.3.9 OMP_SET_NESTED

Enables or disables nested parallelism. (*Nested parallelism is not supported, and is disabled by default*.)

Fortran

```
SUBROUTINE OMP_SET_NESTED(logical_expr)
```

C/C++

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

1.11.3.10 OMP_GET_NESTED

Determine whether or not nested parallelism is enabled. (*Nested parallelism is not supported, and is disabled by default*.)

Fortran

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

Returns **.FALSE.**. *Nested parallelism is not supported.*

C/C++

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

Returns zero. *Nested parallelism is not supported.*

1.11.4 Routines That Manage Synchronization Locks

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

Fortran:

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

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

C/C++:

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

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

1.11.4.1 **OMP_INIT_LOCK** and **OMP_INIT_NEST_LOCK**

Initialize a lock variable for subsequent calls.

Fortran

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

C/C++

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

1.11.4.2 **OMP_DESTROY_LOCK** and **OMP_DESTROY_NEST_LOCK**

Disassociates a lock variable from any locks.

Fortran

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

C/C++

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

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

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

Fortran

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

C/C++

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

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

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

Fortran

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

C/C++

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

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

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

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

Fortran

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

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

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

Returns nesting count if lock set successfully, zero otherwise.

C/C++

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

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

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

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

1.11.5 Timing Routines

Two functions support a portable wall clock timer.

1.11.5.1 OMP_GET_WTIME

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

Fortran

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

C/C++

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

1.11.5.2 OMP_GET_WTICK

Returns the number of seconds between successive clock ticks.

Fortran

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

C/C++

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

CHAPTER 2

Implementation Dependent Issues

This chapter details specific issues in the OpenMP Fortran 2.0 and OpenMP C/C++ 1.0 specifications that are implementation dependent.

Scheduling

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

Number of Threads

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

Dynamic Threads

- Without an explicit call to the **`omp_set_dynamic()`** function, or an explicit definition of the **`OMP_DYNAMIC`** environment variable, the default is to enable dynamic thread adjustment.

Nested Parallelism

- Nested parallelism is not supported in this implementation, and is disabled by default.

`ATOMIC` Directive

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

`GUIDED` Initial Chunk

- The default chunk size with **`SCHEDULE(GUIDED,`** *chunk*`)` is 1. The size of the initial set of iterations is the number of iterations in the loop divided by the number of threads executing the loop.

C++ Implementation

- For C++, the implementation is restricted to the OpenMP C specifications. In particular, the use of class objects as private data items within any OpenMP clause is not supported in this release. Also, any exceptions thrown in parallel regions will have unspecified behavior.

CHAPTER 3

Compiling for OpenMP

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

To run a parallelized program in a multithreaded environment, you must set the **OMP_NUM_THREADS** environment variable prior to program execution. This tells the runtime system the maximum number of threads the program can create. The default is 1. In general, set **OMP_NUM_THREADS** to the available number of processors on the target platform.

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

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

3.1 Fortran 95

To enable explicit parallelization with OpenMP directives, compile the program with the **f95** option flag **-openmp**. This flag is a macro for the following combination of **f95** options:

```
-mp=openmp -explicitpar -stackvar -D_OPENMP=200011
```

-openmp=stubs links with the stubs routines for the OpenMP API routines. Use this option if you need to compile your application to execute serially.
-openmp=stubs also defines the **_OPENMP** preprocessor token.

See the **f95**(1) man page for details on these options.

3.1.1 Validation of OpenMP Directives with `-XlistMP`

You can obtain a static, interprocedural validation of a 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:

Violations in the specifications of parallel directives:

- If ordered sections are contained in the dynamic extent of the **DO** directive, the **ORDERED** clause must be present in **DO** directive.
- The variable in the enclosing **PARALLEL** region must be **SHARED** if it is specified on the **LASTPRIVATE** list of a **DO** directive.
- An **ORDERED** directive can appear only in the dynamic extent of a **DO** or **PARALLEL DO** directive.
- If a variable is **PRIVATE** (explicitly or implicitly) or **THREADPRIVATE** in a **PARALLEL** region and this variable is set in this region then it is incorrect to use this variable after this **PARALLEL** region.
- Variables in the **COPYPRIVATE** list must be private in the enclosing context.
- Variables that appear on the **FIRSTPRIVATE**, **LASTPRIVATE**, and **REDUCTION** clauses on a work-sharing directive must have shared scope in the enclosing parallel region.
- **DO**, **SECTIONS**, **SINGLE**, and **WORKSHARE** directives that bind to the same **PARALLEL** directive are not allowed to be nested one inside the other.
- **DO**, **SECTIONS**, **SINGLE**, and **WORKSHARE** directives are not permitted in the dynamic extent of **CRITICAL**, **ORDERED**, and **MASTER** directives.
- **BARRIER** directives are not permitted in the dynamic extent of **DO**, **SECTIONS**, **SINGLE**, **WORKSHARE**, **MASTER**, **CRITICAL**, and **ORDERED** directives.
- **MASTER** directives are not permitted in the dynamic extent of **DO**, **SECTIONS**, **SINGLE**, **WORKSHARE**, **MASTER**, **CRITICAL**, and **ORDERED** directives.
- **ORDERED** directives are not allowed in the dynamic extent of **SECTIONS**, **SINGLE**, **WORKSHARE**, **CRITICAL**, and **MASTER** directives.
- Multiple **ORDERED** sections are not permitted in the dynamic extent of **PARALLEL DO**.

Obstacles to parallelization determined by interprocedural data dependence analysis:

- Variables declared as **PRIVATE** are undefined for each thread on entering the construct.
- It is incorrect to use **REDUCTION** variable outside reduction statement.
- Variables that are declared **LASTPRIVATE** or **REDUCTION** for a work-sharing directive for which **NOWAIT** appears must not be used prior to a barrier.

- Assignment into shared scalar variable inside parallel construct may lead to incorrect results.
- Using a **SHARED** variable as an **ATOMIC** variable may cause performance degradation.
- Value of a private variable can be undefined if this variable was assigned inside **MASTER** or **SINGLE** blocks.

Additional diagnostics:

- **ATOMIC** directive applies only to the immediately following statement which must be of the special form.
- Syntax or usage is wrong for a **REDUCTION** statement.
- The operator declared in a **REDUCTION** clause should be the same as in **REDUCTION** statement.
- **ATOMIC** variables must be scalars.
- **CRITICAL** directives with the same name are not allowed to be nested one inside the other.

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

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

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

3.2 C and C++

To enable explicit parallelization with OpenMP directives, compile your program with the option flag **-xopenmp**. This flag can take an optional keyword argument.

If you specify **-xopenmp** but do not include a keyword, the compiler assumes **-xopenmp=parallel**. If you do not specify **-xopenmp**, the compiler assumes **-xopenmp=none**.

-xopenmp=parallel enables recognition of OpenMP pragmas and applies to SPARC only. The optimization level under **-xopenmp=parallel** is **-xO3**. The compiler issues a warning if the optimization level of your program is changed from a lower level to **-xO3**. **-xopenmp=parallel** defines the **_OPENMP** preprocessor token to be YYYYMM (specifically 199810L).

-xopenmp=stubs links with the stubs routines for the OpenMP API routines. Use this option if you need to compile your application to execute serially. **-xopenmp=stubs** also predefines the **_OPENMP** preprocessor token.

-xopenmp=none does not enable recognition of OpenMP pragmas, makes no change to the optimization level of your program, and does not predefine any preprocessor tokens.

With C, do not compile with **-xopenmp** and **-xparallel** or **-xexplicitpar** together**.**

The C++ implementation is limited to just the OpenMP C Version 1.0 API specifications.

3.3 OpenMP Environment Variables

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

TABLE 3-1 OpenMP Environment Variables: **`setenv`** *VARIABLE value*

Environment Varialbe	Function
`OMP_SCHEDULE`	Sets schedule type for **`DO`**, **`PARALLEL DO`**, **`parallel for`**, **`for`**, directives/pragmas with schedule type **`RUNTIME`** specified. If not defined, a default value of **`STATIC`** is used. *value* is *"type[,chunk]"* Example: `setenv OMP_SCHEDULE "GUIDED,4"`
`OMP_NUM_THREADS` *or* **`PARALLEL`**	Sets the number of threads to use during execution, unless set 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. (*Current maximum is 128*). 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. (*Nested parallelism is not supported*). *value* is either **`TRUE`** or **`FALSE`**. (*This variable has no effect.*) 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 TABLE 3-2:

TABLE 3-2 Multiprocessing environment variables

Environment Variable	Function
SUNW_MP_WARN	Controls warning messages issued by the OpenMP runtime library. If set **TRUE** the runtime library issues warning messages to `stderr`; **FALSE** disables warning messages. The default is **FALSE**. Example: **setenv SUNW_MP_WARN FALSE**
SUNW_MP_THR_IDLE	Controls the end-of-task status of each thread executing the parallel part of a program. You can set the value to **spin**, **sleep** *n***s**, or **sleep** *n***ms**. The default is **SPIN** — a thread should spin (or busy-wait) after completing a parallel task, until a new parallel task arrives. Choosing **SLEEP** *time* specifies the amount of time a thread should spin-wait after completing a parallel task. If, while a thread is spinning, a new task arrives for the thread, the tread executes the new task immediately. Otherwise, the thread goes to sleep and is awakened when a new task arrives. *time* may be specified in seconds, (*n***s**), or just (*n*), or milliseconds, (*n***ms**). **SLEEP** with no argument puts the thread to sleep immediately after completing a parallel task. **SLEEP**, **SLEEP (0)**, **SLEEP (0s)**, and **SLEEP (0ms)** are all equivalent. Example: **setenv SUNW_MP_THR_IDLE (50ms)**
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 platforms, and 8 Mb on 64-bit SPARC V9 platforms. Example: **setenv STACKSIZE 8192** *sets the thread stack size to 8 Mb*

3.4 Stacks and Stack Sizes

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

The default main stack is about 8 megabytes. Compiling Fortran programs with the `f95` option `-stackvar` forces the allocation of local variables and arrays on the stack as if they were automatic variables. Use of `-stackvar` with OpenMP programs is required 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 helper thread of a multithreaded program has its own thread stack. This stack mimics the initial 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 systems and 8 megabytes on 64-bit systems. The size of the 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 or C programs with the `-xcheck=stkovf` flag to report on runtime stack overflow situations that occur in the compiled code.

CHAPTER 4

Converting to OpenMP

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

4.1 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*.

4.1.1 Converting Sun-Style Directives

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

TABLE 4-1 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`

`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 4-2 `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. For the private variables only in the list you can achieve the same effect by using `firstprivate(`*list*`)`.
`STOREBACK(`*v1,v2,...*`)`	No exact equivalent. For the private variables only in the list you can achieve the same effect by using `lastprivate(`*list*`)`.
`SAVELAST`	No exact equivalent. For private variables only you can achieve the same effect by using `lastprivate(`*list*`)`.
`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 4-3)

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

TABLE 4-3 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 OpenMP equivalent.
SCHEDTYPE(GSS(*m*))	schedule(guided, *m*) Default *m* is 1.

4.1.1.1 Issues Between Sun-Style Directives and OpenMP

- Scoping of variables (shared or private) 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.
- Clauses on OpenMP directives do not accumulate; that is, there can be at most only one type of each clause on a directive.
- 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.
- Because OpenMP provides a richer parallelism model, it might often be possible to get better performance by redesigning the parallelism strategies of a program that uses Sun directives to take advantage of these features.

4.1.2 Converting Cray-Style 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 4-4 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.
`SAVELAST`	No exact equivalent. For private variables only you can achieve the same effect by using `lastprivate(`*list*`)`.
`MAXCPUS(`*n*`)`	`num_threads(`*n*`).` No exact equivalent.
`GUIDED`	`schedule(guided,` *m*`)` Default *m* is 1.
`SINGLE`	`schedule(dynamic,1)`
`CHUNKSIZE(`*n*`)`	`schedule(dynamic,`*n*`)`
`NUMCHUNKS(`*m*`)`	`schedule(dynamic,`*n/m*`)` where *n* is the number of iterations

4.1.2.1 Issues Between Cray-Style Directives and OpenMP Directives

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

4.2 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 4-5 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 4-6 `taskloop` Optional Clauses and OpenMP Equivalents

`taskloop` **Clause**	**OpenMP** `parallel for` **Equivalent Clause**
`maxcpus(`*n*`)`	No equivalent.
`private(`*v1,v2,...*`)`	`private(`*v1,v2,...*`)`
`shared(`*v1,v2,...*`)`	`shared(`*v1,v2,...*`)`
`readonly(`*v1,v2,...*`)`	No exact equivalent. For the private variables only in the list you can achieve the same effect by using `firstprivate(`*list*`)`.
`storeback(`*v1,v2,...*`)`	No exact equivalent. For the private variables only in the list you can achieve the same effect by using `lastprivate(`*list*`)`.
`savelast`	No exact equivalent. For private variables only you can achieve the same effect by using `lastprivate(`*list*`)`.
`reduction(`*v1,v2,...*`)`	`reduction(operator:`*v1,v2,...)* Must supply the reduction operator as well as the list.
`schedtype(`*spec*`)`	`schedule(`*spec*`)` (See TABLE 4-7)

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

TABLE 4-7 `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 OpenMP equivalent.
`SCHEDTYPE(GSS(`*m*`))`	`schedule(guided,` *m*`)` Default *m* is 1.

4.2.1 Issues Between Legacy C Pragmas and OpenMP

- Variables declared within a parallel construct are scoped `private`. A `default(none)` clause on a `#pragma omp parallel for` directive causes the compiler to flag variables not scoped explicitly.
- Clauses on OpenMP directives do not accumulate; that is, there can be at most only one type of each clause on a directive.
- 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 might often be 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