APPENDIX **E**

Fortran 95 OpenMP Directives

This information is extracted from the Fortran Programming Guide, Appendix E.

The Sun Fortran 95 compiler supports the OpenMP 2.0 Fortran API. The `-openmp` compiler flag enables these directives.

This section lists the OpenMP directives, library routines, and environment variables supported by `f95`. For details about parallel programming with OpenMP, see the OpenMP 2.0 Fortran specification at `http://www.openmp.org/`.

The following table summarizes the OpenMP directives supported by `f95`. Items enclosed in square brackets ([...]) are optional. The compiler permits comments to follow an exclamation mark (`!`) on the same line as the directive. When compiling with `-openmp`, the CPP/FPP variable `_OPENMP` is defined and may be used for conditional compilation within `#ifdef _OPENMP` and `#endif`.

TABLE E-1 Summary of OpenMP Directives in Fortran 95

Directive Format (Fixed)	`C$OMP` *directive optional_clauses...* `!$OMP` *directive optional_clauses...* `*$OMP` *directive optional_clauses...* Must start in column one; continuation lines must have a non-blank or non-zero character in column 6
Directive Format (Free)	`!$OMP` *directive optional_clauses...* May appear anywhere, preceded by whitespace; an ampersand (`&`) at the end of the line identifies a continued line.
Conditional Compilation	Source lines beginning with `!$`, `C$`, or `*$` in columns 1 and 2 (fixed format), or `!$` preceded by white space (free format) are compiled only when compiler option `-openmp`, or `-mp=openmp` is specified.

TABLE E-1 Summary of OpenMP Directives in Fortran 95 *(Continued)*

PARALLEL *Directive*	!$OMP PARALLEL [*clause*[[,] *clause*]...] *block of Fortran statements with no transfer in or out of block* !$OMP END PARALLEL Defines a *parallel region*: a block of code that is to be executed by multiple threads in parallel. *clause* can be one of the following: PRIVATE(*list*), SHARED(*list*), DEFAULT(*option*), FIRSTPRIVATE(*list*), REDUCTION(*list*), IF(*logical_expression*), COPYIN(*list*), NUM_THREADS(*integer_expression*).
DO *Directive*	!$OMP DO [*clause*[[,] *clause*]...] *do_loop statements block* [!$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. *clause* can be one of the following: PRIVATE(*list*), FIRSTPRIVATE(*list*), LASTPRIVATE(*list*), REDUCTION(*list*), SCHEDULE(*type*), ORDERED.
SECTIONS *Directive*	!$OMP SECTIONS [*clause*[[,] *clause*]...] [!$OMP SECTION] *block of Fortran statements with no transfer in or out* [!$OMP SECTION *optional block of Fortran statements*] ... !$OMP END SECTIONS [NOWAIT] Encloses a non-iterative section of code to be divided among threads in the team. Each section is executed once by a thread in the team. *clause* can be one of the following: PRIVATE(*list*), FIRSTPRIVATE(*list*), LASTPRIVATE(*list*), REDUCTION(*list*). Each section is preceded by a SECTION directive, which is optional for the first section.
SINGLE *Directive*	!$OMP SINGLE [*clause*[[,] *clause*]...] *block of Fortran statements with no transfer in or out* !$OMP END SINGLE [*end-modifier*] The statements enclosed by SINGLE is to be executed by only one thread in the team. Threads in the team that are not executing the SINGLE block of statements wait at the END SINGLE directive unless NOWAIT is specified. *clause* can be one of: PRIVATE(*list*), FIRSTPRIVATE(*list*). *end-modifier* is either COPYPRIVATE(*list*)[[,]COPYPRIVATE(*list...*)] or NOWAIT.

TABLE E-1 Summary of OpenMP Directives in Fortran 95 *(Continued)*

WORKSHARE *Directive*	!$OMP WORKSHARE *block of Fortran statements* !$OMP END WORKSHARE [NOWAIT] 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.
PARALLEL DO *Directive*	!$OMP PARALLEL DO [*clause*[[,] *clause*]...] *do_loop statements block* [!$OMP END PARALLEL DO] Shortcut for specifying a parallel region that contains a single DO loop: a PARALLEL directive followed immediately by a DO directive. *clause* can be any of the clauses accepted by the PARALLEL and DO directives.
PARALLEL SECTIONS *Directive*	!$OMP PARALLEL SECTIONS [*clause*[[,] *clause*]...] [!$OMP SECTION] *block of Fortran statements with no transfer in or out* [!$OMP SECTION *optional block of Fortran statements*] ... !$OMP END PARALLEL SECTIONS Shortcut for specifying a parallel region that contains a single SECTIONS directive: a PARALLEL directive followed by a SECTIONS directive. *clause* can be any of the clauses accepted by the PARALLEL and SECTIONS directives.
PARALLEL WORKSHARE *Directive*	!$OMP PARALLEL WORKSHARE[*clause*[[,] *clause*]...] *block of Fortran statements* !$OMP END 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.
Synchronization Directives	
MASTER *Directive*	!$OMP MASTER *block of Fortran statements with no transfers in or out* !$OMP END MASTER The block of statements enclosed by these directives is executed only by the master thread of the team. The other threads skip this block and continue. There is no implied barrier on entry to or exit from the master section.

TABLE E-1 Summary of OpenMP Directives in Fortran 95 *(Continued)*

`CRITICAL` *Directive*	`!$OMP CRITICAL` [(*name*)] *block of Fortran statements with no transfers in or out* `!$OMP END CRITICAL` [(*name*)] Restrict access to the statement block enclosed by these directives 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. If a name conflicts with any other entity, the behavior of the program is undefined. If *name* appears on the `CRITICAL` directive, it must also appear on the `END CRITICAL` directive.
`BARRIER` *Directive*	`!$OMP BARRIER` Synchronizes all the threads in a team. Each thread waits until all the others in the team have reached this point.
`ATOMIC` *Directive*	`!$OMP ATOMIC` Ensures that a specific memory location is to be updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads. The directive applies only to the immediately following statement, which must be one of these forms: *x = x operator expression* *x = expression operator x* *x = intrinsic(x, expression)* *x = intrinsic(expression, x)* where: • *x* is a scalar of intrinsic type • *expression* is a scalar expression that does not reference *x* • *intrinsic* is one of `MAX, MIN, IAND, IOR,` or `IEOR`. • *operator* is one of `+ - * / .AND. .OR. .EQV. .NEQV.` *This implementation replaces all ATOMIC directives by enclosing the target statement in a critical section.*

TABLE E-1 Summary of OpenMP Directives in Fortran 95 *(Continued)*

FLUSH *Directive*	!$OMP FLUSH [(*list*)] Thread-visible variables are written back to memory at the point at which this directive appears. The FLUSH directive only provides consistency between operations within the executing thread and global memory. The optional *list* consists of a comma-separated list of variables that need to be flushed. The FLUSH directive is implied for the following directives: BARRIER, CRITICAL/ENDCRITICAL, ENDDO, END SECTIONS, ENDSINGLE, ENDWORKSHARE, ORDERED/ENDORDERED, PARALLEL/ENDPARALLEL, PARALLEL/ENDPARALLELDO, PARALLELSECTIONS/ENDPARALLELSECTIONS, PARALLELWORKSHARE/ENDPARALLELWORKSHARE. FLUSH is not implied if NOWAIT is specified. It is not implied by: DO, MASTER/ENDMASTER, SECTIONS, SINGLE, and WORKSHARE.
ORDERED *Directive*	!$OMP ORDERED *block of Fortran statements with no transfers in or out* !$OMP END ORDERED The enclosed block of statements are executed in the order that iterations would be executed in a sequential execution of the loop. It can appear only in the dynamic extent of a DO or PARALLEL DO directive. The ORDERED clause must be specified on the closest DO directive enclosing the block.
Data Environment Directives	
THREADPRIVATE *Directive*	!$OMP THREADPRIVATE(*list*) Makes the *list* of variables and named common blocks private to a thread but global within the thread. Common block names must appear between slashes. To make a common block THREADPRIVATE, this directive must appear after every COMMON declaration of that block.
Data Scoping Clauses	
Several directives noted above accept clauses to control the scope attributes of variables enclosed by the directive. If no data scope clause is specified for a directive, the default scope for variables affected by the directive is SHARED. *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/)	
PRIVATE *Clause*	PRIVATE(*list*) Declares the variables in the comma separated *list* to be private to each thread in a team.

TABLE E-1 Summary of OpenMP Directives in Fortran 95 *(Continued)*

SHARED *Clause*	SHARED(*list*) All the threads in the team share the variables that appear in *list*, and access the same storage area.
DEFAULT *Clause*	DEFAULT(PRIVATE | 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.
FIRSTPRIVATE *Clause*	FIRSTPRIVATE(*list*) Variables on list are PRIVATE. In addition, private copies of the variables are initialized from the original object existing before the construct.
LASTPRIVATE *Clause*	LASTPRIVATE(*list*) Variables on the list are PRIVATE. In addition, when the LASTPRIVATE clause appears on a DO 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.
REDUCTION *Clause*	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. The REDUCTION clause is intended to be used on a region in which the reduction variable is used only in reduction statements of the form shown previously for the ATOMIC directive. 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.
COPYIN *Clause*	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.

TABLE E-1 Summary of OpenMP Directives in Fortran 95 *(Continued)*

`COPYPRIVATE` *Clause*	`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`..
Scheduling Clauses on `DO` *and* `PARALLEL DO` *Directives*	
`SCHEDULE` *Clause*	`SCHEDULE(`*type* [*,chunk*]`)` Specifies how iterations of the DO loop are divided among the threads of the team. *type* can be one of the following. In the absence of a `SCHEDULE` clause, `STATIC` scheduling is used.
`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. *chunk* must be a scalar integer expression.
`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.
`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. (Default chunk size is 1. The size of the initial piece of the iterations is the number of iterations in the loop divided by the number of threads executing the loop.)
`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 `STATIC`.)

OpenMP Library Routines

OpenMP Fortran API library routines are external procedures. In the following summary, *int_expr* is a default scalar integer expression, and *logical_expr* is a default 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 95 `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.

TABLE E-2 Summary of Fortran 95 OpenMP Library Routines

Execution Environment Routines
`OMP_SET_NUM_THREADS` Subroutine `SUBROUTINE OMP_SET_NUM_THREADS(`*int_expr*`)` Sets the number of threads to use for the next parallel region.
`OMP_GET_NUM_THREADS` Function `INTEGER(4) FUNCTION OMP_GET_NUM_THREADS()` Returns the number of threads currently in the team executing the parallel region from which it is called.
`OMP_GET_MAX_THREADS` Function `INTEGER(4) FUNCTION OMP_GET_MAX_THREADS()` Returns the maximum value that can be returned by calls to the `OMP_GET_NUM_THREADS function.`
`OMP_GET_THREAD_NUM` Function `INTEGER(4) FUNCTION OMP_GET_THREAD_NUM()` Returns the thread number within the team. This is a number between 0 and `OMP_GET_NUM_THREADS()-1.` The master thread is thread 0.
`OMP_GET_NUM_PROCS` Function `INTEGER(4) FUNCTION OMP_GET_NUM_PROCS()` Returns the number of processors that are available to the program.

TABLE E-2 Summary of Fortran 95 OpenMP Library Routines *(Continued)*

`OMP_IN_PARALLEL` Function

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

Returns .TRUE. if called from within the dynamic extent of a region executing in parallel, and .FALSE. otherwise.

`OMP_SET_DYNAMIC` Subroutine

```
SUBROUTINE OMP_SET_DYNAMIC(logical_expr)
```

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

`OMP_GET_DYNAMIC` Function

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

Returns .TRUE. if dynamic thread adjustment is enabled and returns .FALSE. otherwise.

`OMP_SET_NESTED` Subroutine

```
SUBROUTINE OMP_SET_NESTED(logical_expr)
```

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

`OMP_GET_NESTED` Function

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

Returns .TRUE. if nested parallelism is enabled, .FALSE. otherwise. *Nested parallelism is not supported; this function will always return* .FALSE.

Lock Routines

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.

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
```

TABLE E-2 Summary of Fortran 95 OpenMP Library Routines *(Continued)*

`OMP_INIT_LOCK` Subroutine
 	
`SUBROUTINE OMP_INIT_LOCK(`*var*`)`
`SUBROUTINE OMP_INIT_NEST_LOCK(`*nvar*`)`
 	
Initializes a lock associated with lock variable *var* for use in subsequent calls. The initial state is unlocked.

`OMP_DESTROY_LOCK` Subroutine
 	
`SUBROUTINE OMP_DESTROY_LOCK(`*var*`)`
`SUBROUTINE OMP_DESTROY_NEST_LOCK(`*nvar*`)`
 	
Disassociates the given lock variable *var* from any locks.

`OMP_SET_LOCK` Subroutine
 	
`SUBROUTINE OMP_SET_LOCK(`*var*`)`
`SUBROUTINE OMP_SET_NEST_LOCK(`*nvar*`)`
Forces the executing thread to wait until the specified lock is available. The thread is granted ownership of the lock when it is available.

`OMP_UNSET_LOCK` Subroutine
 	
`SUBROUTINE OMP_UNSET_LOCK(`*var*`)`
`SUBROUTINE OMP_UNSET_NEST_LOCK(`*nvar*`)`
Releases the executing thread from ownership of the lock. Behavior is undefined if the thread does not own that lock.

`OMP_TEST_LOCK` Function
 	
`LOGICAL FUNCTION OMP_TEST_LOCK(`*var*`)`
`INTEGER FUNCTION OMP_TEST_NEST_LOCK(`*nvar*`)`
Attempts to set the lock associated with lock variable. Returns `.TRUE.` if the simple lock was set successfully, `.FALSE.` otherwise. `OMP_TEST_NEST_LOCK` returns the new nesting count if the lock associated with *nvar* was set successfully, otherwise it returns 0.

Timing Routines These two functions, returning double precision (`REAL(8)`), support a portable wall-clock timer.
`OMP_GET_WTIME` *Function* `REAL(8) FUNCTION OMP_GET_WTIME()` Returns a double precision value equal to the elapsed wallclock time in seconds since “some arbitrary time in the past”
`OMP_GET_WTICK` *Function* `REAL(8) FUNCTION OMP_GET_WTICK()` Returns a double precision value equal to the number of seconds between successive clock ticks.

OpenMP Environment Variables

TABLE E-3 and TABLE E-4 summarize the OpenMP Fortran API environment variables that control the execution of OpenMP programs.

TABLE E-3 Summary of OpenMP Fortran Environment Variables

`OMP_SCHEDULE` Sets schedule type for `DO` and `PARALLEL DO` directives specified with schedule type `RUNTIME`. If not defined, a default value of `STATIC` is used. Value is *"type*[*,chunk*]*"* Example: `setenv OMP_SCHEDULE "GUIDED,4"`.
`OMP_NUM_THREADS` Sets the number of threads to use during execution, unless set by a `NUM_THREADS` clause, or a call to `OMP_SET_NUM_THREADS`() subroutine. If not set, a default of 1 is used. Value is a positive integer. (*Current maximum is 128*). 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 `TRUE` or `FALSE`. Example: `setenv OMP_DYNAMIC FALSE`
`OMP_NESTED` Enables or disables nested parallelism. (*Nested parallelism is not supported*.) Value is `TRUE` or `FALSE`. The default, if not set, is `FALSE`. Example: `setenv OMP_NESTED TRUE`

TABLE E-4 Environment variables not part of the OpenMP Fortran API

`SUNW_MP_WARN` Controls warning messages issued by the 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 TRUE`
`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 thread stack size. The value is in kilobytes. Example: `setenv STACKSIZE 8192` sets the thread stack size to 8Mb.