CONVEX

Fortran

Fourth Edition

CONVEX Fortran Quick Reference

Order No. DSW-039
Document No. 720-002430-004
Fourth Edition
October 1994

Released with CONVEX Fortran V9.1

Printed in the United States of America.

Contents

Compilation

The `fc` command compiles CONVEX Fortran source code and has the following form on both C Series and SPP Series machines:

`fc` [*options*] *files*

where

options

is one or more compiler options.

files

is one or more Fortran source files, object files, symbolic assembly-language files, or libraries.

Loader options are specified by using the `-W` compiler option.

Compiler options

The following sections list options available for use with the CONVEX Fortran compiler and the preprocessor.

Language-compatibility options

These options support various non-native Fortran compiler features.

`-ansi77`

Perform strict checking for violations of the ANSI FORTRAN 77 standard.

`-ansi90`

Perform strict checking for violations of the ANSI Fortran 90 standard.

`-cfc`

Accept and use certain syntax, defaults, and features of the Cray Fortran language. Store variables and arrays using Cray Fortran's data type lengths. This option implies `-pd8` and cannot be used with the `-i` or `-r` options.

`-cfcwpa`

Compute word addresses when performing Cray pointer arithmetic on explicitly declared Cray pointers. Must be used with the `-cfc` option.

`-dfc`

Cause all references to VAX records in an I/O statement to be decomposed. Useful only when using `-vfc` and performing I/O on VAX records.

`-F66`

Accept and use certain syntax, defaults, and features of FORTRAN 66.

`-nof90`

Disable the Fortran 90 language compatibility normally provided by CONVEX Fortran.

`-sa`
(C Series only)

Disable generation of precompiled argument packets in the text segment. All arguments are placed on the stack. Use only when a Fortran program calls user-supplied C programs.

`-sfc`

Accept and use certain syntax, defaults, and features of the Sun Fortran language.

`-vfc`

Accept and use additional defaults of the VAX Fortran language.

Optimization options

These options enable optimizations designed to make code execute more efficiently.

`-blockloop` *n*
(SPP Series only)

Block *m*-1 loops in an *m* loop nest. The compiler uses a block factor of *n* and automatically selects which loops to block.

`-br`
(SPP Series only)

Enables base register optimizations. This option is on by default at optimization levels `-O1` and higher. To disable base register optimizations use the `-nbr` option.

`-cache` *n*
(SPP Series only)

Assume a single processor direct-mapped cache size of *n* kilobytes. The compiler uses this information when determining a loop's blocking factor.

`-ds`
(C Series only)

Automatically select loops to replicate and compile scalar, vector, and possibly parallel and parallel/vector versions of such loops. The compiler then dynamically selects the version of each loop to be executed.

`-ep` *n*
(C Series only)

Specify the expected number of processors (*n*) on which the program is going to run. Must be used with the `-O3` option.

`-gr`
(SPP Series only)

Enable global register allocation. This is the default and is effective optimization levels `-O1` and higher.

`-il`
(C Series only)

Prepare an intermediate-language (`.fil`) file for inline substitution.

`-is` *directory*
(C Series only)

Attempt inline substitution of each subprogram for which there exists a `.fil` file (intermediate-language file) file in the specified *directory*.

`-mo`
(SPP Series only)

Enable the generation of multi-op instructions. The `-mo` option is the default on SPP Series machines.

`-nbr`
(SPP Series only)

Disable base register optimizations. By default, CONVEX Fortran provides base register optimizations at optimization levels `-O1` and higher.

`-nga`
(SPP Series only)

Disable global register allocation for arguments passed by reference. Global register allocation is enabled by default at optimization levels `-O1` and higher.

`-ngr`
(SPP Series only)

Disable global register allocation, which by default is enabled at optimization levels `-O1` and higher.

`-ngs`
(SPP Series only)

Disable global register allocation for shared memory variables. Global register allocation is enabled by default at optimization levels `-O1` and higher.

`-nmo`
(SPP Series only)

Disable the generation of multi-op instructions. Multi-op instructions are generated by default on SPP Series machines (and via the `-mo` option).

`-no`

Perform no optimization. This option is the default if the `-O1`, `-O2`, and `-O3` options are not specified.

`-noblock`
(SPP Series only)

Prevent the compiler from blocking loops in the specified Fortran source files.

`-nopeel`

Disallow loop boundary value peeling, which is enabled by default at optimization levels `-O2` and `-O3`.

`-nopm`
(C Series only)

Disable pattern matching used by the compiler to aid in vectorization.

`-noptst`

Disallow test promotion, which is enabled by default at optimization levels `-O2` and `-O3`.

`-nsr`

Disable scalar replacement, which is enabled by default. Scalar replacement, when enabled, occurs at optimization levels `-O2` and `-O3`.

`-nuj`

Disable the loop unroll and jam transformation. On SPP Series machines, unroll and jam occurs by default. This transformation affects optimization levels `-O2` and `-O3`.

`-nur`

Disable loop unrolling. On SPP Series machines, loop unrolling is enabled by default.

`-O0`

Basic block machine-independent scalar optimization.

`-O1`

Program unit level scalar optimizations and global register allocation.

`-O2`

Global instruction scheduling, software pipelining, and data localization optimizations. On C Series machines, vectorization also is provided.

`-O3`

Parallel optimizations.

`-peel`

Increase limit for code expansion at optimization levels `-O2` and `-O3`.

`-peelall`

Same as `-peel`, but allows code expansion without bound.

`-ptst`

Increase code expansion limit for test promotion at optimization levels `-O2` and `-O3`.

`-ptstall`

Same as `-ptst`, but allows code replication without bound.

`-rl`

Automatically select loops and replicate them by unrolling or dynamic selection.

`-sr`

Enable scalar replacement. This option is enabled by default and occurs at optimization levels `-O2` and `-O3`. To disable scalar replacement, use the `-nsr` option.

`-uj`

Enable the loop unroll and jam transformation. On SPP Series machines the `-uj` option is enbaled by default. Unroll and jam, when enabled, occurs at `-O2` and `-O3`.

`-ujn` *n*

Enable the loop unroll and jam transformation and specify the desired loop unrolling factor (*n*, where *n* is the number of times to replicate the body of the loop).

`-uo`

Perform potentially unsafe optimizations.

`-ur`

Automatically select and unroll (or partially unroll) loops. On SPP Series machines this is the default. When enabled, loop unrolling occurs at optimization levels `-O2` and `-O3`.

`-urn` *n*

Automatically select and unroll loops, and use a loop unrolling factor of *n*, where *n* is the number of times to replicate the body of the loop.

Code-generation options

The options listed in this section control various aspects of code generation, including default variable and constant sizes.

`-c`

Suppress the loading phase of the compilation.

`-except precise`
(C Series only)

Ensure that any arithmetic exceptions caused by a subprogram before it returns will be received by the subprogram. Generates code specific to the target architecture for compilation. Degrades performance.

`-except default`
(C Series only)

Cancel the effects of `-except precise`.

`-fi`
(C Series only)

Use IEEE format floating-point format. This option cannot be used with `REAL*16` data.

`-fn`
(C Series only)

Use native floating-point format.

`-i`*n*

Specify the default length (*n*) for `INTEGER` and `LOGICAL` variables, either 2, 4, or 8. Obsolete; use `-p8` or `-pd8` instead.

`-mi` *n*
(C Series only)

Specify the expected memory interleave on the target machine. *n* is an integer representing the expected memory interleave, which you can obtain for your machine with the `getsysinfo` command. By default, the interleave of the machine on which the compiler is running is used.

`-nore`

Generate code that passes arguments to subroutines using argument packets instead of the stack. This option is the default on C Series computers.

`-p8`

Specify a default length of 8 bytes for `INTEGER`, `LOGICAL`, and `REAL` variables and constants with unspecified lengths. `DOUBLE PRECISION` and `COMPLEX` values occupy 16 bytes of storage.

`-pd8`

Specify that `INTEGER`, `LOGICAL`, `REAL`, and `DOUBLE PRECISION` variables and constants with unspecified lengths are to occupy 8 bytes of storage; `COMPLEX` values are to occupy 16 bytes.

`-r`*n*

Specify that `REAL` variable declarations with unspecified lengths are to occupy *n* bytes of storage, where *n* can be 4 (the default) or 8. Obsolete; use the `-p8` or `-pd8` option instead.

`-re`

Generate reentrant code for recursive or, when the Application Compiler is used, parallel invocation of subprograms. The `-re` option is the default on SPP Series machines.

`-S`

Generate symbolic assembly code for each program unit in a source file.

`-tm` *target*

Generate code for the specified target machine. On SPP Series machines, `-tm spp1` is the default. On C Series machines, *target* can be any one of the following values:

target	**architecture for compilation**
`c1`	CONVEX C1 Series
`c2`	CONVEX C2 Series

`c32`	CONVEX C3200 Series
`c34`	CONVEX C3400 Series
`c38`	CONVEX C3800 Series
`c4`	CONVEX C4600 Series

Debugging and profiling options

The options listed in this section allow CONVEX Fortran executables to be used with program development tools.

`-al`

Treat noncharacter arrays declared with a last dimension of 1 as assumed-size (last dimension of *).

`-cs`

Check that each subscript is within its array bounds.

`-cxdb`

Generate symbolic debugging information for CXdb.

`-cxpa`

Instrument the compiled code for CXpa profiler analysis at the loop level and routine level. Do not use with the `-cxpab` option.

`-cxpab`

Instrument the compiled code for CXpa analysis at the block level. Do not use with the -cxpa option.

`-cxpalib`

Link your program with libraries instrumented for use with CXpa. Programs profiled with `-cxpalib` may execute much slower. Significantly increases the amount of profiling data.

`-cxpamon="`*dirname*`"`

Use the monitor instrumentation library (cxpamon.o) in the specified directory. The full path of directory *dirname* is specified surrounded by quotation marks.

`-cxpar`

Generate routine level instrumentation for profiling under CXpa.

`-dc`

Compile lines with a `D` in column 1.

`-metrics`
(C Series only)

Write the CXmetrics data file *sourcefile*`.met`, where *sourcefile* is the name of your original Fortran source file without the `.f` or `.FOR` extension.

`-p`

Produce monitor routines for the `prof` profiler.

`-pb`
(C Series only)

Produce information for the `bprof` profiler.

`-pg`

Produce information for the `gprof` profiler.

`-sc`

Check syntax only.

Message and listing options

The options listed in this section provide control over program listings, cross-referencing, and advisory and warning messages.

`-errnames`

Output the message name when a compile-time error message is issued. These names correspond to the errors listed in Appendix C, "Compiler messages."

`-LST`

Generates a listing of compiled source program, inserts compiler error messages (if any) at the appropriate points, and writes it to `stdout`.

`-LSTI`

Generate listing similar to `-LST`, with all `INCLUDE` files expanded.

`-na`

Suppress all advisory diagnostic messages.

`-nw`

Suppress all warning diagnostic messages.

`-or` *table*

Specify the contents of the optimization report; *table* can be `all`, `none`, `loop`, `private` (SPP Series only), or `array`.

`-pl` *n*

Specify a default maximum page length of *n* lines per page for the `-LST`, `-LSTI`, `-xr`, and `-xra` options. Default is *n* = 60.

`-pw` *n*

Specify a desired header width of *n* columns per line for the `-LST`, `-LSTI`, `-xr`, and `-xra` options. Default is *n* = 80.

`-xr`

Invoke the cross-referencer (`fcxref`) and generate a cross-reference report at compilation time.

`-xra`

Invoke the cross-referencer (`fcxref`) and generate a data file but do not generate a report.

`-xro`
(C Series only)

Call the old cross-reference generator (`fxref`). Does not support `REAL*16` variables or Fortran 90 extensions.

`-xrl`
(C Series only)

Call the old cross-reference generator (`fxref`) and place all objects (such as variables and arrays) into one table. Does not support `REAL*16` variables or Fortran 90 extensions.

Preprocessor options

The options listed in this section concern the use of preprocessors.

`-fpp`

Run the Fortran preprocessor as the first step of compilation. If this option is not specified, the preprocessor is not used.

`-D`*name*`[=`*def*`]`

Defines *name* to the preprocessor as if it had been specified in a `#define` statement. If no definition is given, the name is defined as 1.

`-E`

Run only the Fortran preprocessor on the named Fortran programs and send the result to `stdout`.

`-U`*name*

Remove any initial definition of *name*. The only built-in names defined are "`__LINE__`", and "`__FILE__`".

`-pp`

Invoke a user-defined preprocessor.

Miscellaneous options

The options listed in this section are not otherwise categorizable.

`-align cseries`

Store `COMMON` blocks using "tight" packing (the ANSI standard). Tight `COMMON` block packing is the default on C Series machines.

`-align spp`

Align all data items in `COMMON` blocks to their natural boundaries (up to 8 bytes). This is the default on SPP Series machines.

`-ansic`
(C Series only)

Link `/usr/lib/libc.a` to your program. This link allows mixed ANSI C and Fortran programs.

`-B`*dir*

Find the substitute compiler (`fskel`, `fpp`, and `errmsgf`) in the directory named *dir*.

`-I`*dir*

Specify directories where `INCLUDE` files may be located. No more than eight directories can be specified.

`-link` *arg*

Pass *arg* to the loader, where *arg* is an arbitrary loader option. You must delimit *arg* within quotation marks if a blank space appears in the argument.

`-nosc`

Disable short circuit evaluation of conditionals.

`-noU77`
(SPP Series only)

Prevent trailing underscore (_) characters from being added to names of routines in the `libU77` Fortran library (or any variety of `libU77`, such as `libU77p8`).

`-o` *name*

Assign *name* as the name of the executable file produced by the loader. The default name is `a.out`.

`-pcc`
(C Series only)

Link a Portable C Compiler compatible `libc.a` to your program, thus allowing you to mix only the PCC dialect of C with Fortran programs. This option has no effect on SPP Series machines.

`-ppu`
(SPP Series only)

Append an underscore character (_) to the end of external name definitions and references.

`-tl` *n*
(C Series only)

Set the maximum CPU time limit for compilation to *n* minutes.

`-vn`

Display the version number of the compiler that is being used. Output goes to `stderr`.

`-W`*subproc*`,`*sparg*`[,`*sparg*`...]`

Pass specified argument(s) to subprocess *subproc*. Each argument (*sparg*) takes the form *-arg*`[,` *argvalue*`]`, where *arg* is the name of an option recognized by the subprocess and *argvalue* is a separate argument that is included if required. The following values are recognized for *subproc*:

`p`	preprocessor
`c`	compiler
`o`	compiler
`a`	assembler
`l`	loader

`-72`

Process only the first 72 characters of each program line. The compiler normally processes all characters.

`error` utility

The `error` utility inserts any error messages produced by the compiler into the source code at the point where the error occurred. The `error` program has the following form:

```
fc [options] sourcefile.f |& error
```

where *options* includes any command line options and *sourcefile.f* is the name of the file containing the Fortran source code. The file *sourcefile.f* is modified by `error` program.

Directives

The general form of a compiler directive is as follows:

```
C$DIR [SPP|CSERIES] directive [, directive...]
```

If one of the optional target machine attributes (`SPP` or `CSERIES`) is specified, the directive line will apply only when compiled for the target machine.

If two or more directives are specified, they are separated by commas. A directive must fit on one line; it cannot be continued.

`BARRIER` (SPP Series only)

Declare one or more variables of data type `BARRIER`. Has the form

```
C$DIR BARRIER(barr-name [, barr-name...])
```

where *barr-name* is the name of the `BARRIER` variable to be declared.

`BEGIN_TASKS, NEXT_TASK, END_TASKS`

Identify a group of tasks for independent, parallel execution. A group of tasks begins with a `BEGIN_TASKS` directive and ends with an `END_TASKS` directive. A `NEXT_TASK` directive precedes the second task and all subsequent tasks in the group.

The `BEGIN_TASKS` directive has the form

```
C$DIR BEGIN_TASKS [(attribute-list)]
```

where the optional *attribute-list* qualifies the way in which tasks are run in parallel. Only the ORDERED attribute is available on C Series machines.

On CONVEX SPP Series machines, *attribute-list* can be any one of the following combinations of attributes:

```
ORDERED
NODES
THREADS
MAX_THREADS=m
ORDERED, NODES
ORDERED, THREADS
ORDERED, MAX_THREADS=m
NODES, MAX_THREADS=m
THREADS, MAX_THREADS=m
ORDERED, NODES, MAX_THREADS=m
ORDERED, THREADS, MAX_THREADS=m
```

BLOCK_LOOP (SPP Series only)

Use a specified block factor to strip mine the immediately following loop. Has the form

```
C$DIR BLOCK_LOOP [ (BLOCK_FACTOR = n) ]
```

where *n* is the blocking factor requested for strip mining. If the block factor is omitted, an appropriate loop blocking factor is selected.

BLOCK_SHARED (SPP Series only)

Declare the specified allocatable arrays as being of memory type block-shared. Block-shared arrays are distributed equally among all hypernodes on which a program is running; pages of the array are distributed in same-size blocks across all the subcomplex's hypernodes to accomplish this. It takes the form

```
C$DIR BLOCK_SHARED(alloc-arr [, alloc-arr...])
```

where *alloc-arr* is an allocatable array that appears in a prior ALLOCATABLE statement.

CRITICAL_SECTION (SPP Series only)

Mark a section of code that must be executed by one thread at a time. This directive has the form

```
C$DIR CRITICAL_SECTION[(gate-var)]
```

where the optional *gate-var* is a previously declared GATE variable to be used to control execution of the designated section of code.

The END_CRITICAL_SECTION directive specifies the point where the critical section ends. It does not accept a GATE variable.

FAR_SHARED (SPP Series only)

Store a specified list of variables in far-shared memory. This directive uses the following format:

```
C$DIR FAR_SHARED (namelist)
```

where *namelist* is a list of COMMON block names, array names, and scalar variable names.

FAR_SHARED_POINTER (SPP Series only)

Place a hidden, compiler-generated pointer to the specified variable in far-shared memory, regardless of the memory class to which the variable is allocated. This directive has the form

```
C$DIR FAR_SHARED_POINTER(alloc-var)
```

where *alloc-var* is the name of a variable that appears in a prior ALLOCATABLE statement.

FORCE_PARALLEL (C Series only)

Parallelize the loop that follows, regardless of apparent dependencies between iterations. FORCE_PARALLEL does *not* allow interchange or distribution of outer loops for vectorization. To enable those optimizations, use FORCE_PARALLEL_EXT.

FORCE_PARALLEL_EXT (C Series only)

Parallelize the loop that follows, regardless of apparent dependencies between iterations. FORCE_PARALLEL_EXT allows interchange of outer loops for vectorization.

FORCE_VECTOR (C Series only)

Vectorize the loop that follows, regardless of apparent recurrences. Causes the compiler to ignore any apparent dependencies between iterations. When you use this directive on a loop, you may not get correct results. Check answers generated by the vectorized code.

GATE (SPP Series only)

Declare one or more variables of type GATE. The format of this directive is

```
C$DIR GATE(gate-name [, gate-name...])
```

where *gate-name* is the name of the GATE variable to be declared.

LOOP_PARALLEL (SPP Series only)

Run the immediately following loop in parallel. A loop marked with a LOOP_PARALLEL directive must have a known number of iterations at loop invocation time.

LOOP_PARALLEL differs from the PREFER_PARALLEL directive in that LOOP_PARALLEL parallelizes the loop that follows regardless of any loop-carried dependencies.

The format of the LOOP_PARALLEL directive is as follows:

```
C$DIR LOOP_PARALLEL [(attribute-list)]
```

where the optional *attribute-list* can contain any one of the following combination of attributes:

THREADS — causes thread-level parallelism.

NODES — causes node-level parallelism.

CHUNK_SIZE = *n* — divides the loop into chunks of *n* iterations and distributes the chunks round-robin to the processors.

MAX_THREADS = *m* — allows no more than *m* threads to be allocated to the execution of the loop. *m* must be a constant integer which has a value at compile time.

ORDERED — ordered execution of the loop; provides no automatic synchronization.

ORDERED, NODES — ordered execution across hypernodes.

ORDERED, THREADS — ordered execution across threads.

NODES, CHUNK_SIZE = *n* — node-level parallelism by chunks.

THREADS, CHUNK_SIZE = *n* — thread-level parallelism by chunks.

CHUNK_SIZE = *n*, MAX_THREADS = *m* — chunk parallelism on no more than *m* threads.

ORDERED, MAX_THREADS = *m* — ordered parallelism on no more than *m* threads.

NODES, MAX_THREADS = *m* — node-level parallelism on no more than *m* hypernodes.

THREADS, MAX_THREADS = *m* — thread-level parallelism on no more than *m* threads.

ORDERED, NODES, MAX_THREADS = *m* — ordered node-level parallelism on no more than *m* hypernodes.

ORDERED, THREADS, MAX_THREADS=*m* — ordered thread-level parallelism on no more than *m* threads.

NODES, CHUNK_SIZE=*n*, MAX_THREADS=*m* — node-level parallelism by chunks of size *n* on no more than *m* hypernodes

THREADS, CHUNK_SIZE=*n*, MAX_THREADS=*m* — thread-level parallelism by chunks of size n on no more than m threads.

IVAR=*ivarname* — specifies the primary induction variable (*ivarname*) of the loop. Required for DO WHILE and "hand-rolled" loops.

When using this directive check results generated by the LOOP_PARALLEL code.

LOOP_PRIVATE

Declare a list of variables and/or arrays private to the immediately following DO loop.

Variables declared LOOP_PRIVATE are assumed to have no loop-carried dependencies. No starting or ending values can be assumed for these variables. This directive has the form

```
C$DIR LOOP_PRIVATE(varlist)
```

where *varlist* is a list of scalar variables or arrays, separated by commas, that are to be private to the following loop.

MAX_TRIPS (C Series only)

Indicate that the following loop is never executed more than a specific number of times. This directive's format is

```
C$DIR MAX_TRIPS (n)
```

where the value of *n* is less than or equal to the vector register length of 128.

NEAR_SHARED (SPP Series only)

Store a specified list of variables in near-shared memory. This directive uses the following format:

```
C$DIR NEAR_SHARED (namelist)
```

where *namelist* is a list of names of COMMON blocks, arrays, and scalar variables that are to be stored in near-shared memory.

NEAR_SHARED_POINTER (SPP Series only)

Place a hidden, compiler-generated pointer to the specified variable in near-shared memory, regardless of the memory class to which the variable is allocated. Has the form

```
C$DIR NEAR_SHARED_POINTER(alloc-var)
```

where *alloc-var* is the name of a variable that appears in a prior ALLOCATABLE statement.

NO_BLOCK_LOOP (SPP Series only)

Perform no blocking on the immediately following loop.

NODE_PRIVATE (SPP Series only)

Store a specified list of variables in node-private memory. This directive uses the format

```
C$DIR NODE_PRIVATE (namelist)
```

where *namelist* is a list of COMMON block names, array names, and scalar variable names.

NODE_PRIVATE_POINTER (SPP Series only)

Place a hidden, compiler-generated pointer to the specified variable in node-private memory, regardless of the memory class to which the variable is allocated. This directive has the form

```
C$DIR NODE_PRIVATE_POINTER(alloc-var)
```

where *alloc-var* is the name of a variable that appears in a prior ALLOCATABLE statement.

NO_LOOP_DEPENDENCE (SPP Series only)

Safely ignore any potential loop-carried dependencies (LCDs) the compiler detects in specified arrays and variables. This directive has the form

```
C$DIR NO_LOOP_DEPENDENCE (namelist)
```

where *namelist* is a list of arrays whose potential LCDs will be ignored by the compiler. If *namelist* is not specified, the compiler assumes that there are no LCDs in any of the loop's arrays. Use NO_LOOP_DEPENDENCE for arrays only. Use LOOP_PRIVATE to specify dependence-free scalar variables.

NO_PARALLEL

Do *not* parallelize the loop that immediately follows. Vectorization (available on C Series machines) is not prevented.

NO_PEEL

Prevent the compiler from applying loop boundary value peeling to the loop that immediately follows. This directive overrides boundary level peeling at all levels: default, `-peel`, and `-peelall`.

NO_PROMOTE_TEST

Prevent the compiler from applying test promotion to the loop that immediately follows. This directive overrides test promotion at all levels: default, `-ptst`, and `-ptstall`.

NO_RECURRENCE (C Series only)

Disregard any apparent recurrence in a loop. If there is no other impediment to vectorization, the loop is vectorized.

NO_SIDE_EFFECTS

Specify functions that *do not* modify the value of a parameter or COMMON variable, perform a read or write, or call another routine that has side effects. The format of this directive is

```
C$DIR NO_SIDE_EFFECTS ( func [, func...])
```

where *func* specifies one or more user-defined functions.

NO_UNROLL_AND_JAM

Disallow the unroll and jam transformation for the immediately following loop only. On SPP Series machines the unroll and jam transformation is enabled by default. This transformation affects optimization levels `-O2` and `-O3`.

NO_VECTOR (C Series only)

Do *not* vectorize the loop that immediately follows; parallelization is not prevented.

ORDERED_SECTION (SPP Series only)

Force all threads to execute a designated section of code one thread at a time in thread order (the order in which the threads are spawned). Has the following form:

```
C$DIR ORDERED_SECTION (gate)
```

where *gate* specifies a previously declared GATE variable to be used to control entry into the ordered section.

The END_ORDERED_SECTION directive specifies the point where the ordered section ends and does not accept the *gate* parameter.

PEEL

Peel the loop immediately following the directive, expanding the code beyond the default conservative limit, but not without bound.

PEEL_ALL

Peel the loop immediately following the directive, expanding the code without bound.

PREFER_PARALLEL

Parallelize the loop immediately following the directive if it is safe to do so. The compiler checks first for actual loop-carried dependencies; if none is found, the loop is parallelized.

A loop marked with a PREFER_PARALLEL directive must have a known number of iterations at loop invocation time. The PREFER_PARALLEL directive has the form:

```
C$DIR PREFER_PARALLEL [(attribute-list)]
```

where the optional *attribute-list* qualifies the way in which the immediately following loop is parallelized.

On CONVEX C Series machines, only the CHUNK_SIZE and ORDERED attributes are available. On SPP Series machines, the optional *attribute-list* can contain any one of the following combinations of attributes:

THREADS — causes thread-level parallelism.

NODES — causes node-level parallelism.

CHUNK_SIZE = *n* — divides the loop into chunks of *n* iterations and distributes the chunks round-robin to the processors.

MAX_THREADS = *m* — allows no more than *m* threads to be allocated to the execution of the loop. *m* must be a constant integer which has a value at compile time.

ORDERED — ordered execution of the loop; provides no automatic synchronization.

ORDERED, NODES — ordered execution across hypernodes.

ORDERED, THREADS — ordered execution across threads.

NODES, CHUNK_SIZE = *n* — node-level parallelism by chunks.

THREADS, CHUNK_SIZE = *n* — thread-level parallelism by chunks.

CHUNK_SIZE = *n*, MAX_THREADS = *m* — chunk parallelism on no more than *m* threads.

ORDERED, MAX_THREADS = *m* — ordered parallelism on no more than *m* threads.

`NODES, MAX_THREADS = m` — node-level parallelism on no more than *m* hypernodes.

`THREADS, MAX_THREADS = m` — thread-level parallelism on no more than *m* threads.

`ORDERED, NODES, MAX_THREADS = m` — ordered node-level parallelism on no more than *m* hypernodes.

`ORDERED, THREADS, MAX_THREADS=m` — ordered thread-level parallelism on no more than *m* threads.

`NODES, CHUNK_SIZE = n, MAX_THREADS=m` — node-level parallelism by chunks of size *n* on no more than *m* hypernodes

`THREADS, CHUNK_SIZE = n, MAX_THREADS=m` — thread-level parallelism by chunks of size *n* on no more than *m* threads.

`IVAR=`*ivarname* — specifies the primary induction variable (*ivarname*) of the loop. Required for `DO WHILE` and "hand-rolled" loops.

PREFER_PARALLEL_EXT (C Series only)

Parallelize the loop immediately following the directive only if it appears safe to do so. The compiler checks first for actual loop-carried dependencies; if none are found, the loop is parallelized.

PREFER_VECTOR (C Series only)

Vectorize the loop immediately following the directive if it is safe to do so. The compiler checks first for actual recurrences. If no recurrences are found, the compiler tries to interchange the loop to be the innermost loop and vectorize it.

PROMOTE_TEST

Promote tests out of the loop immediately following the directive, replicating code beyond the default conservative limit, but not without bound.

PROMOTE_TEST_ALL

Promote tests out of the loop immediately following the directive, replicating code without bound.

PSTRIP (C Series only)

Strip mine the parallel loop immediately following the directive using the specified length. The format of this directive is

```
C$DIR PSTRIP (integer_constant)
```

where *integer_constant* is an integer constant that specifies the strip-mine length.

ROW_WISE

Tell the compiler that the designated arrays have their dimensions reversed. Thus, array elements are stored in row-major order (as in C and Ada) rather than column-major order. The format of this directive is

```
C$DIR ROW_WISE (array-name [,array-name...])
```

SAVE_LAST (SPP Series only)

Specify that all LOOP_PRIVATE variables in the immediately following loop have their values from the final iteration saved for use after the loop's termination.

SCALAR

Prevent the DO loop that follows from being vectorized or parallelized. The body of the loop can still be vectorized or parallelized if an outer loop is interchanged with the SCALAR loop. The results of a vectorized loop can differ from its scalar equivalent.

SELECT (C Series only)

Generate multiple versions of a loop and select, at runtime, which version to execute based on specified trip counts. The compiler generates up to four versions of a loop: scalar, vector, parallel, and parallel-vector. The format of this directive is

```
C$DIR SELECT (vtrip, ptrip, pvtrip )
```

The arguments *vtrip*, *ptrip*, and *pvtrip* specify the trip (iteration) count at which the compiler is to select vector, parallel, or parallel-vector execution, respectively, for the loop.

SYNCH_PARALLEL (C Series only)

Execute the following loop in parallel; however, instead of ignoring dependencies, insert synchronization code that causes the dependencies to be honored at runtime.

TASK_PRIVATE

Declare a list of variables and/or arrays to be private to the immediately following task. (In CONVEX Fortran, tasks are defined using the BEGIN_TASKS, NEXT_TASK, and END_TASKS directives.) The TASK_PRIVATE directive has the form

```
C$DIR TASK_PRIVATE(varlist)
```

where *varlist* is a list of variables or arrays, separated by commas, that are to be private to each following task.

THREAD_PRIVATE (SPP Series only)

Force one or more specified variables to be stored in thread-private memory. This directive uses the following format:

```
C$DIR THREAD_PRIVATE (namelist)
```

where *namelist* is a list of COMMON block names, array names, and scalar variable names.

THREAD_PRIVATE_POINTER (SPP Series only)

Place a hidden, compiler-generated pointer to the specified variable in thread-private memory, regardless of the memory class to which the variable is allocated. This directive has the form

```
C$DIR THREAD_PRIVATE_POINTER(alloc-var)
```

where *alloc-var* is the name of a variable that appears in a prior ALLOCATABLE statement.

UNROLL

Replace the body of the loop that follows with a linear series of statements. Unrolling is performed only on scalar loops. This directive has the form

```
C$DIR UNROLL[(UNROLL_FACTOR=n)]
```

where the optional UNROLL_FACTOR argument specifies *n*, the number of times to replicate the body of the loop.

UNROLL_AND_JAM

Enable the loop unroll and jam transformation for the immediately following loop. Exploits the use of registers and thus decreases the number of slower main memory accesses. The UNROLL_AND_JAM directive has the following form:

```
C$DIR UNROLL_AND_JAM[(UNROLL_FACTOR=n)]
```

where the optional UNROLL_FACTOR argument specifies *n*, the number of times to replicate the body of the loop.

VSTRIP (C Series only)

Strip mine the vector loop immediately following using the specified length. This directive has the following format:

```
VSTRIP (integer-constant)
```

where *integer-constant* is an integer constant that specifies the strip-mine length, which must be less than or equal to 128.

Specification statements

Specification statements are nonexecutable statements that appear before the first executable statement in a program unit.

ALLOCATABLE statement

Declare allocatable arrays. This statement takes the following form

```
[type arrexp]
ALLOCATABLE arrexp [,...]
```

where *type* is an optional type definition for the array and *arrexp* is the array name or rank definition if it was not given in a preceding type statement.

AUTOMATIC statement

Store a specified list of variables in stack-based storage, rather than static storage. The AUTOMATIC statement is available only when the `-sfc` option (Sun Fortran compatibility) is supplied. This has the following format:

```
AUTOMATIC varlist
```

where *varlist* is a list of variables (separated by commas) that will be allocated static storage.

COMMON statement

Allow variables or arrays in a main program or subprogram to share the same storage location with variables and arrays in other subprograms. This statement has the form

```
COMMON [/[cbn]/] nlist [ [,]/[cbn]/nlist]...
```

where *cbn* is a symbolic name for a COMMON block and *nlist* is a list of variable names, array names, and array declarators.

EQUIVALENCE statement

Cause two or more entities within a program unit to refer to the same storage area. Thus, the same storage unit can be referenced by more than one name. This statement has the form

```
EQUIVALENCE (list) [,(list)]...
```

where *list* is a list of two or more variables, array elements, array names, or character substring names separated by commas.

EXTERNAL statement

Identify a symbolic name as representing an externally-defined procedure or dummy procedure. This statement indicates that a given name is the name of a subprogram, not a variable or array name. It has the form

```
EXTERNAL n [,n]...
```

where *n* is the symbolic name of a user-supplied subprogram, block data subprogram, or dummy procedure.

IMPLICIT statement

Override the implied data typing of symbolic names within a program unit. The IMPLICIT statement has the forms:

```
IMPLICIT typ (a[,a]...) ...
```

or

```
IMPLICIT NONE
```

where

typ is an INTEGER[**len*], REAL[**len*], DOUBLE PRECISION, DOUBLE COMPLEX, COMPLEX[**len*], LOGICAL[**len*], or CHARACTER[**len*] data type.

a is one letter or a range of letters; the range is expressed as first letter of range, minus sign (-), last letter of range (for example, F-H).

and

len is an optional length specifier for the data type.

The IMPLICIT NONE form of this statement overrides all implicit defaults except intrinsic function types.

INCLUDE statement

Insert source code from the specified file into the program being compiled. The contents of the file are inserted at the place where the INCLUDE statement appears. The statement has the form

```
INCLUDE 'filename'
```

or

```
#include "filename"
```

where *filename* is the path of the file from which source code is to be read.

INTRINSIC statement

Permit the name of a specific intrinsic function to be used as an actual argument. This statement has the form

```
INTRINSIC intrname [,intrname]...
```

where *intrname* is one of the Fortran intrinsic functions.

The following categories of intrinsic functions cannot be used as actual arguments: type conversion (for instance, INT, IFIX, IDINT, REAL, FLOAT, SNGL, DBLE, CMPLX, ICHAR, CHAR), and maximum and minimum value (such as MAX, MAX0, AMAX1, AMAX0, MAX1, MIN, MIN0, AMIN1, DMIN1, AMIN0, and MIN1).

NAMELIST statement

Associate a single unique name with a list of variables or array names. This name defines a list of entities that can be modified (read) or transferred (written). The NAMELIST statement has the form

```
NAMELIST /nlgrpname/varlist...
```

where *nlgrpname* is a symbolic name representing the list of entities to be read or written, and *varlist* is a list of variable or array names (separated by commas) to be associated with the symbolic name *nlgrpname*.

OPTIONS statement

Select compiler options via a Fortran specification statement. The options in an OPTIONS statement override those specified on the fc command line. This statement has the form

```
OPTIONS option [option...]
```

PARAMETER statement

Assign a symbolic name to a constant. It takes either of the following two forms. The first is

```
PARAMETER (name = exp[, name = exp] ...)
```

where *name* is a symbolic name and *exp* is a constant or constant expression. The other is

```
PARAMETER p=c [, p=c ]...
```

where *p* is a symbolic name and *c* is a constant, the symbolic name of a constant, or a compile-time constant expression.

PROGRAM statement

Assign a name to the main program unit; its use is optional. The PROGRAM statement has the form:

```
PROGRAM pgm
```

where *pgm* is the symbolic name for the main program.

STATIC statement

Store a specified list of variables in static storage, rather than stack-based storage. The STATIC statement is available only when Sun Fortran compatibility is requested via the `-sfc` option. The format is as follows:

```
STATIC varlist
```

where *varlist* is a list of variables (separated by commas) that will be allocated static storage.

SAVE statement

Retain the values of designated variables and arrays in a subroutine or function after a RETURN or END statement is executed. The SAVE statement has the form

```
SAVE [n [,n]... ]
```

where *n* is a variable name, statically sized array name, or a named COMMON block contained between slashes (for example, /NCOM/). If the SAVE statement contains no arguments, the values of all allowable entities are retained.

ORDER NUMBER
DSW-039

DOCUMENT NUMBER
720-002430-004