MIPSpro™ 7 Fortran 90 Commands and Directives Reference Manual

Document Number 007–3696–003

New Features

MIPSpro™ 7 Fortran 90 Commands and Directives Reference Manual 007–3696–003

This manual describes the commands and directives supported by the MIPSpro 7 Fortran 90 compiler, release 7.3. Additions to this manual to support this release include the following:

- Support for the Workshop ProMP toolset.
- The `-fb`*file* option, which specifies the feedback file to be used.
- The `-fullwarn` option, which generates comment-level messages.
- The `-LANG:IEEE_minus_zero option`, which controls whether or not a minus sign (`-`) is written for negative zero.
- The `-LNO:local_pad_size=`*n* option, which specifies the amount by which to pad local array dimensions.
- The `-LNO:parallel_overhead=`*setting* option, which overrides internal compiler estimates concerning the efficiency to be gained by executing certain loops in parallel rather than serially.
- The `-LNO:pure=`*n* option, which specifies the extent to which the compiler should consider the effect of a `PURE` procedure or a `!DIR$ NOSIDEEFFECTS` directive when performing parallel analysis.
- The `-noappend` option, which prevents the compiler from appending a trailing underscore character (_) on external names.
- The `-OPT:speculative_ptr_deref=`*setting* option, which allows speculative loads of memory locations that differ by a small offset from some referenced memory location.
- The `-pad_char_literals` option, which blanks pads all character literal constants that are shorter than the size of the default integer type and that are passed as actual arguments.
- The `-static_threadprivate` option, which makes all static variables private to each thread.
- The `!DIR$ NOSIDEEFFECTS` directive, which allows the compiler to keep information in registers across a single call to a subprogram without reloading the information from memory after returning from the subprogram.

Additional new options were added to the `-DEBUG:` and `-IPA:` option groups. For information on these options, see the `debug_group`(5) and `ipa`(5) man pages.

Record of Revision

Version	*Description*
3.0	August 1997 Original Printing. This printing supports the MIPSpro 7 Fortran 90 compiler, release 7.2, running on IRIX systems.
3.0.2	March 1998 This revision supports the MIPSpro 7 Fortran 90 compiler, release 7.2.1, running on IRIX systems. It includes miscellaneous corrections and additions to the 3.0 revision.
003	April 1999 This revision supports the MIPSpro 7 Fortran 90 compiler, release 7.3, running on IRIX systems. It includes miscellaneous corrections and additions to the 3.0.2 revision.

Contents

About This Manual

This manual describes the commands and directives for using the MIPSpro 7 Fortran 90 compiler, which is invoked through the `f90`(1) command. It is organized into the following chapters:

- Chapter 1, page 1, introduces the content of the manual and provides a general description of the compiler.
- Chapter 2, page 5, describes the `f90`(1) command, which you use to invoke the compiler. This chapter includes information about using the `f90`(1) command line options, CPU targeting, obtaining a listing, and other aspects of compiling with the MIPSpro 7 Fortran 90 compiler.
- Chapter 3, page 69, introduces the compiler directives and describes the general compiler directives that the MIPSpro 7 Fortran 90 compiler recognizes.
- Chapter 4, page 89, describes the OpenMP Fortran API multiprocessing directives.
- Chapter 5, page 143, describes parallel processing on Origin2000, Origin200, or Cray Origin2000 systems using OpenMP directives and Silicon Graphics extensions to the OpenMP directives.
- Chapter 6, page 171, describes CF90 compiler directives that are also supported by the MIPSpro 7 Fortran 90 compiler.
- Chapter 7, page 185, describes the source preprocessor.
- Chapter 8, page 197, describes the interlanguage calling conventions used when calling a C/C++ function from a Fortran procedure and a Fortran procedure from a C function.
- Chapter 9, page 213, describes the Auto-Parallelizing Option (APO) and the directives that accompany this feature. APO requires an additional license from Silicon Graphics, Inc. Please contact your sales representative if you are interested in using this feature.
- Appendix A, page 239, describes library routines available to you from Fortran programs.
- Appendix B, page 249, describes debugging Fortran programs.
- Appendix C, page 253, describes the Autotasking directives. These directives are still supported, but they are outmoded. It is suggested that you develop

new codes using the OpenMP Fortran API directives described in Chapter 4, page 89.

- Appendix D, page 267, describes the Silicon Graphics multiprocessing directives. These directives are still supported, but they are outmoded. It is suggested that you develop new codes using the OpenMP Fortran API directives described in Chapter 4, page 89.

Related MIPSpro 7 Fortran 90 Publications

This manual is one of a set of manuals that describes the MIPSpro 7 Fortran 90 compiler. The other manuals in the set are as follows:

- *Intrinsic Procedures Reference Manual*
- *Fortran Language Reference Manual, Volume 1*
- *Fortran Language Reference Manual, Volume 2*
- *Fortran Language Reference Manual, Volume 3*

MIPSpro 7 Fortran 90 Messages

You can obtain explanations for MIPSpro 7 Fortran 90 compiler messages by using the online `explain`(1) command.

MIPSpro 7 Fortran 90 Man Pages

In addition to printed and online prose documentation, several online man pages describe aspects of the MIPSpro 7 Fortran 90 compiler. Man pages exist for the library routines, the intrinsic procedures, and several programming environment tools.

You can print copies of online man pages by using the pipe symbol with the `man`(1), `col`(1), and `lpr`(1) commands. In the following example, these commands are used to print a copy of the `explain`(1) man page:

```
% man explain | col -b | lpr
```

Each man page includes a general description of one or more commands, routines, system calls, or other topics, and provides details of their usage (command syntax, routine parameters, system call arguments, and so on). If more than one topic appears on a page, the entry in the printed manual is alphabetized under its primary name; online, secondary entry names are linked to these primary names. For example, `egrep` is a secondary entry on the page with a primary entry name of `grep`. To access `grep` online, you can type `man grep`. To access `egrep` online, you can type either `man grep` or `man egrep`. Both commands display the `grep` man page on your terminal.

Related Fortran Publications

The following commercially available reference books are among those that you should consult for more information on the history of Fortran and the Fortran language itself:

- Adams, J., W. Brainerd, and J. Martin. *Fortran 95 Handbook : Complete ISO/ANSI Reference*. MIT Press, 1997. ISBN 0262510960.
- Chapman, S. *Fortran 90/95 for Scientists and Engineers*. McGraw Hill Text, 1998. ISBN 0070119384.
- Chapman, S. *Introduction to Fortran 90/95*. McGraw Hill Text, 1998. ISBN 0070119694.
- Counihan, M. *Fortran 95 : Including Fortran 90, Details of High Performance Fortran (HPF), and the Fortran Module for Variable-Length Character Strings*. UCL Press, 1997. ISBN 1857283678.
- Gehrke, W. *Fortran 95 Language Guide*. Springer Verlag, 1996. ISBN 3540760628.
- International Standards Organization. *ISO/IEC 1539–1:1997, Information technology — Programming languages — Fortran*. 1997.
- Metcalf, M. and J. Reid. *Fortran 90/95 Explained*. Oxford University Press, 1996. ISBN 0198518889.

Related Publications

The following documents contain information that may be useful when using the MIPSpro 7 Fortran 90 compiler:

- *Application Programmer's I/O Guide*
- *ProDev ProMP User's Guide*
- *MIPSpro Assembly Language Programmer's Guide*

- *MIPSpro Compiling and Performance Tuning Guide*
- *MIPSpro Fortran 77 Programmer's Guide*
- *MIPSpro 64-Bit Porting and Transition Guide*
- *SpeedShop User's Guide*

Obtaining Publications

Silicon Graphics maintains publications information at the following URL:

`http://techpubs.sgi.com/library`

The preceding website contains information that allows you to browse documents online, order documents, and send feedback to Silicon Graphics.

The *User Publications Catalog*, describes the availability and content of all Cray hardware and software documents that are available to customers. Customers who subscribe to the Cray Inform (CRInform) program can access this information on the CRInform system.

Cray Research also has documents available online at the following URL:

`http://www.cray.com/swpubs`

To order a document, call +1–651–683–5907. Silicon Graphics employees may send their orders via electronic mail to `orderdsk`.

Customers outside of the United States and Canada should contact their local service organization for ordering and documentation information.

Conventions

The following conventions are used throughout this document:

Convention	Meaning
`command`	This fixed-space font denotes literal items such as commands, files, routines, path names, signals, messages, and programming language structures.
variable	Italic typeface denotes variable entries and words or concepts being defined.

`user input`	This bold, fixed-space font denotes literal items that the user enters in interactive sessions. Output is shown in nonbold, fixed-space font.
[ ]	Brackets enclose optional portions of a command or directive line.
...	Ellipses indicate that a preceding element can be repeated.

Reader Comments

If you have comments about the technical accuracy, content, or organization of this document, please tell us. Be sure to include the title and part number of the document with your comments.

You can contact us in any of the following ways:

- Send electronic mail to the following address:

  `techpubs@sgi.com`

- Send a facsimile to the attention of "Technical Publications" at fax number +1 650 932 0801.

- Use the Suggestion Box form on the Technical Publications Library World Wide Web page:

  `http://techpubs.sgi.com/library/`

- Call the Technical Publications Group, through the Technical Assistance Center, using one of the following numbers:

  For Silicon Graphics IRIX based operating systems: 1 800 800 4SGI

  For UNICOS or UNICOS/mk based operating systems or CRAY Origin2000 systems: 1 800 950 2729 (toll free from the United States and Canada) or +1 651 683 5600

- Send mail to the following address:

  Technical Publications
  Silicon Graphics, Inc.
  1600 Amphitheatre Pkwy.
  Mountain View, California 94043–1351

We value your comments and will respond to them promptly.

Introduction [1]

This manual describes the MIPSpro 7 Fortran 90 compiler. This compiler runs under the IRIX operating system, version 6.2 and later, on Silicon Graphics and Cray computer systems.

The MIPSpro 7 Fortran 90 compiler was developed to support the Fortran standard adopted by the American National Standards Institute (ANSI) and the International Standards Organization (ISO). This standard, commonly referred to as *the Fortran 95 standard*, is ISO/IEC 1539–1:1997. Because the Fortran 95 standard is a superset of previous Fortran standards, the MIPSpro 7 Fortran 90 compiler compiles code written in accordance with previous Fortran standards.

The Fortran 95 standard is a revision to the Fortran 90 language standard. Because of the number and complexity of the features, the standards organizations are continuing to interpret the Fortran 95 standard for Silicon Graphics and for other vendors. To maintain conformance to the Fortran 95 standard, Silicon Graphics may need to change the behavior of certain MIPSpro 7 Fortran 90 features in future releases based upon the outcome of the outstanding interpretations to the standard.

1.1 The `f90`(1) Command

In the following example, the `f90`(1) command is used to invoke the compiler. The `-listing` option is specified to generate a source listing and a cross reference. File `pgm.f` is the input file. After compilation, you can run this program by entering the output file name as a command. In this example, the default output file name, `a.out`, is used. Figure 1 illustrates this example:

```
% f90 -listing pgm.f
% ./a.out
```

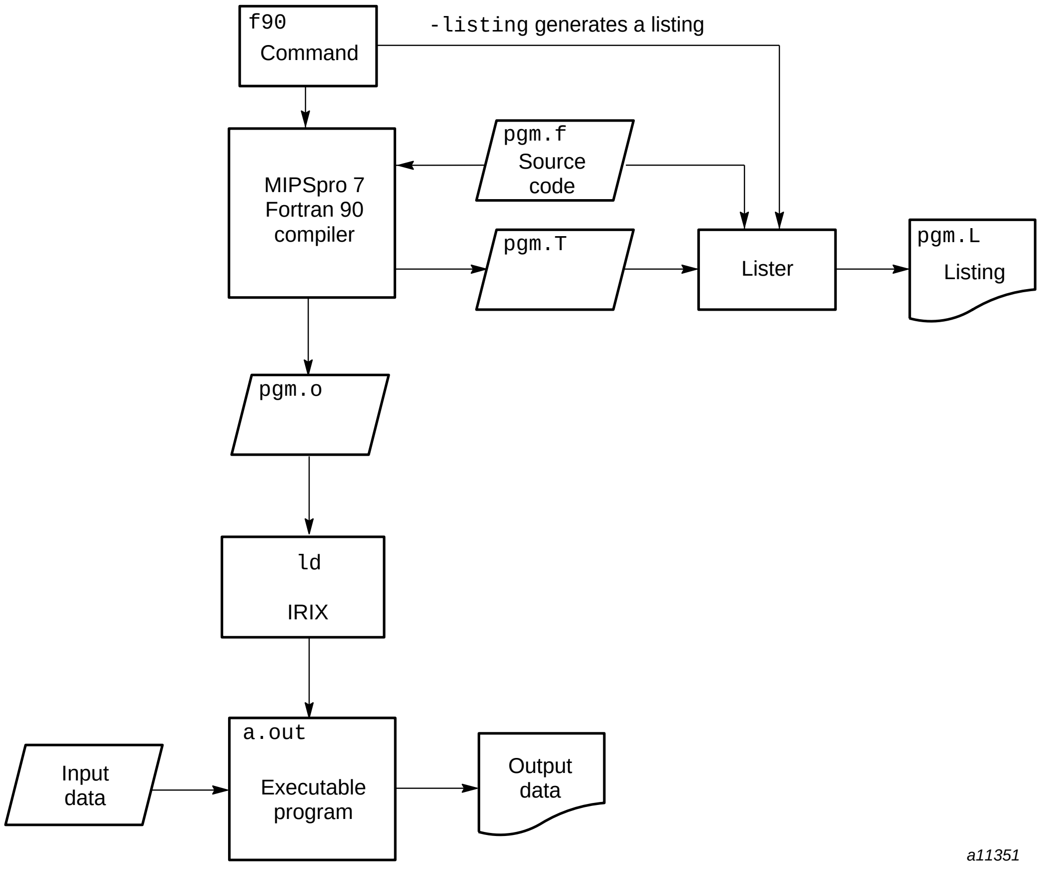

Figure 1. `f90`(1) command example

You can use the options on the `f90`(1) command line to modify the default actions; for example, you can disable the load step. For more information on `f90`(1) command line options, see Chapter 2, page 5.

1.2 The MIPSpro 7 Fortran 90 Programming Environment

The MIPSpro 7 Fortran 90 compiler is one of many products that form the IRIX programming environment. This environment allows you to develop, debug, and run Fortran codes on your computer system. It includes the following products:

- A loader. By default, the IRIX loader, `ld`(1), is invoked and your program is automatically loaded.
- A preprocessor. By default, files suffixed with `.F` or `.F90` are run through the Fortran source preprocessor prior to compilation. You can use the `-ftpp` or `-cpp` options on the `f90`(1) command line to invoke a preprocessor for files without the `.F` or `.F90` suffix.
- A lister. You can specify the `-listing` option on the `f90`(1) command line to obtain a source listing and a cross reference. You can also invoke a separate lister, `ftnlist`(1).
- The `ftnlint`(1) utility, which checks Fortran programs for possible errors.
- The compiler information file (CIF) tools, which include the `cifconv`(1) command and the libraries. For more information on these, see the *Compiler Information File (CIF) Reference Manual*.
- The libraries, which include functions optimized for use on IRIX systems. Information on the individual library routines can be found in the online man pages for each routine. In addition to online man pages, the *Application Programmer's Library Reference Manual*, contains printed copies of the library routine man pages and other library information.

  The intrinsic procedures are implemented within the math library (`libm`), within `libfortran`, and within the compiler itself. The *Intrinsic Procedures Reference Manual*, contains printed copies of the online man pages for all the intrinsic procedures.
- The performance tools contained in SpeedShop and in the ProDev ProMP suite. For more information on these products, see the *SpeedShop User's Guide* or the *ProDev ProMP User's Guide*.
- The archiving tool. An *archive library* is a file that contains one or more routines in object file format (*file*`.o`). When a program calls an object file that is not explicitly included in the program, the loader, `ld`(1), looks for that object file in an archive library. The loader then loads only that object file, not the whole library, and loads it with the calling program.

The archiver creates and maintains archive libraries. It allows you to copy new objects into the library, replace existing objects in the library, move objects within the library, and copy individual objects from the library into individual object files. For more information on the archive library, see the `ar`(1) man page.

- Object file tools, which allow you to disassemble object files into machine instructions, print information about archive files, and perform other tasks. For more information on these tools, see the following man pages: `dis`(1), `elfdump`(1), `file`(1), `nm`(1), `size`(1), and `strip`(1).
- `ftnchop`(1), `ftnmgen`(1), and `ftnsplit`(1). These commands invoke a program unit problem isolator, a Fortran makefile utility, and a split utility, respectively. For more information on these commands, see the man pages for each.
- Online documentation utilities. The `man`(1) command allows you to retrieve online man pages. Prose reference text, such as this manual, can be retrieved through the WWW browser supported at your site. Contact your support staff for specific information on retrieving information in this manner.
- Modules. The MIPSpro 7 Fortran 90 compiler can be installed with the modules utility. This utility allows you to access different versions of the compiler and runtime environment. For more information on using the modules utility, see the `modules`(1) man page or enter the following command:

  ```
  % relnotes modules
  ```
- The message system. This system lets you obtain more comprehensive explanations of messages generated by the compiler and tools in the MIPSpro 7 Fortran 90 compiling environment. When a message condition occurs, both a message number and a verbal summary of the problem is generated. If you need more information on the error condition described in the summary, you can enter the `explain`(1) command to retrieve a more detailed description.
- Environment variables. For more information, see the `pe_environ`(5) man page, which describes many environment variables that can be used when compiling Fortran programs.

Invoking MIPSpro 7 Fortran 90 [2]

This chapter describes the options for the `f90`(1) command. Section 2.66.11, page 61, describes CPU targeting.

The `f90`(1) command invokes the MIPSpro 7 Fortran 90 compiler. The following syntax boxes show the `f90`(1) command syntax:

```
f90 [-64 | -n32][-mipsn] file.suffix[90] [file.suffix[90]]...
```

```
f90 [-64 | -n32] [-alignn] [-ansi] [-apo] [-apokeep] [-apolist]
   [-auto_use module_name[,module_name] ...] [-C] [-check_bounds]
   [-c] [-chunk=integer] [-cif] [-coln] [-cord] [-cpp] [-cray_mp]
   [-Dvar[=def][,var[=def]]...] [-DEBUG:...] [-dn] [-default64] [-E]
   [-extend_source] [-fbfile] [-fixedform] [-flist] [-FLIST:]
   [-freeform] [-ftpp] [-fullwarn] [-Gnum] [-g[debug_lvl]] [-help]
   [-I[dir]] [-INLINE:...] [-ipa] [-IPA[:...]] [-in] [-ignore_suffix]
   [-KPIC] [-keep] [-Ldirectory] [-llibrary] [-LANG:...] [-LIST:...]
   [-LNO:...] [-listing] [-macro_expand] [-MDupdate[file]] [-mipsn]
   [-mp] [-mplist] [-MP:...] [-mp_schedtype=mode] [-noappend]
   [-nocpp] [-noextend_source] [-nostdinc] [-Olevel] [-OPT:...]
   [-oout_file] [-P] [-pfa] [-pfakeep] [-pfalist] [-rreal_spec]
   [-rprocessor] [-S] [-static] [-static_threadprivate] [-TARG:...]
   [-TENV:...] [-Uvar] [-u] [-version] [-Wl,opt[,arg][,opt[,arg]]...]
   [-w[arg]] [-woffnum] [-xdirlist] [-xgot] [--] file.suffix[90]
   [file.suffix[90]...]
```

In some cases, more than one option can have an effect on a single compiler feature. The following list shows some of the compiler features and the options that affect them:

- Listing control: `-flist`, `-FLIST:`, `-listing`, `-LIST:`.
- Source preprocessing: `-cpp`, `-D`*var*[=*def*][,*var*[=*def*]]..., `-E`, `-F`, `-ftpp`, `-macro_expand`, `-nocpp`, `-P`, `-U`*var*.
- Setting the compilation environment: `-n32`, `-64`, `-mips`*n*, `-r`*processor*, `-TARG:`, `-TENV:`.
- Optimization: `-apo`, `-LNO:`, `-OPT:`, `-O`*level*.

Note: The MIPSpro Auto-Parallelizing Option is invoked when you specify the `-apo` command line option. You must be licensed for the MIPSpro Auto-Parallelizing Option in order to be able to use this command line option.

Various environment variable settings can affect your compilation. For more information on the environment variables, see the `pe_environ`(5) man page.

Some `f90`(1) command options, for example, `-LNO:...`, `-LIST:...`, `-MP:...` , `-OPT:...`, `-TARG:...`, and `-TENV:...` accept several suboptions and allow you to specify a setting for each suboption. To specify multiple suboptions, either use colons to separate each suboption or specify multiple options on the command line. For example, the following command lines are equivalent:

```
f90 -LIST:notes=ON:options=OFF b.f
f90 -LIST:notes=ON -LIST:options=OFF b.f
```

Some arguments to suboptions of this type are specified with a setting that either enables or disables the feature. To enable a feature, specify the suboption either alone or with `=1`, `=ON`, or `=TRUE`. To disable a feature, specify the suboption with either `=0`, `=OFF`, or `=FALSE`. For example, the following command lines are equivalent:

```
f90 -LNO:auto_dist:blocking=OFF:oinvar=FALSE a.f
f90 -LNO:auto_dist=1:blocking=0:oinvar=OFF a.f
```

For brevity, this manual shows only the `ON` or `OFF` settings to suboptions, but the compiler also accepts 0, 1, `TRUE`, and `FALSE` as settings.

2.1 -64, -n32

Specifies the Application Binary Interface (ABI), either `-n32` or `-64`. Specifying `-n32` generates 32–bit objects. Specifying `-64` generates 64–bit objects.

Note: Certain predefined system defaults can greatly affect your compilation. These include system defaults for your ABI, Instruction Set Architecture (ISA), and processor type. To determine the default ABI for your system, look in file `/etc/compiler.defaults`. To determine your system's processor, use the `hinv`(1) command. The `-64` and `-n32` options can affect the Instruction Set Architecture (ISA) used during compilation. For more information on this interaction, see the `-mips`*n* option.

When `-n32` is specified, the total memory allocation for a program and individual arrays cannot exceed 2 gigabytes (2 GB, or 2,048 MB). When `-64` is specified, the compiler supports arrays that are larger than 2 GB.

As the following example shows, the arrays can be local, global, or dynamically created when compiling with the following command line:

```
f90 -64 -i8 -mips3 whale.f
```

```
MODULE DEFS
INTEGER, PARAMETER   :: ARRAY_SIZE = 4294967304_8     ! Z'100000008'
INTEGER              :: I(ARRAY_SIZE)
END MODULE


PROGRAM MAIN
USE DEFS
INTEGER, ALLOCATABLE :: J(:)
INTEGER              :: STATUS

ALLOCATE(J(ARRAY_SIZE), STAT=STATUS)

IF (STATUS == 0) THEN
  I(ARRAY_SIZE) = 7
  J(ARRAY_SIZE) = 8
  CALL SUB
END IF

END PROGRAM


SUBROUTINE SUB
USE DEFS
INTEGER :: K(ARRAY_SIZE)

K(ARRAY_SIZE) = 9;

END SUBROUTINE
```

Note: In the preceding example, you cannot specify an array with a size greater than 32 bits in an input or an output list of a READ, WRITE, or PRINT statement.

You must have enough swap space to support the working set size and you must have your shell limit datasize, stack size, and vmemoryuse variables set to values large enough to support the sizes of the arrays. For information on these settings, see the sh(1) man page.

The following example compiles and runs the preceding code after setting the stack size to a correct value:

```
$uname -a
IRIX64 cydrome 6.2 03131016 IP19
$f90 -64 -i8 -mips3 whale.f
$limit
cputime         unlimited
filesize        unlimited
datasize        unlimited
stacksize       65536 kbytes
coredumpsize    0 kbytes
memoryuse       524288 kbytes
descriptors     300
vmemoryuse      unlimited
threads         1024
$limit stacksize unlimited
$limit
cputime         unlimited
filesize        unlimited
datasize        unlimited
stacksize       524288 kbytes
coredumpsize    0 kbytes
memoryuse       754544 kbytes
descriptors     300
vmemoryuse      unlimited
threads         1024
```

2.2 `-align`*n*

Aligns data objects on specified boundaries. The `-align`*n* specifications are as follows:

Option	Action
`-align32`	Aligns objects 32 bits or larger on 32-bit boundaries.
`-align64`	Aligns objects 64 bits or larger on 64-bit boundaries. Default.

When an alignment is specified, objects smaller than the specification are aligned on boundaries that correspond to their sizes. For example, when `align64` is specified, 32-bit and larger objects are aligned on 32-bit boundaries; 16-bit and larger objects are aligned on 16-bit boundaries; and 8-bit and larger objects are aligned on 8-bit boundaries.

2.3 `-ansi`

Causes the compiler to generate messages when it encounters source code that does not conform to the Fortran standard. Specifying this option in conjunction with the `-fullwarn` option causes all messages, regardless of level, to be generated. For more information on the `-fullwarn` option, see Section 2.26, page 16.

2.4 `-apo`, `-apokeep`, `-apolist`

Controls the Auto-Parallelizing Option (APO), which automatically converts sequential code into parallel code by inserting parallel directives where it is safe and beneficial to do so. Specifying `-apo` invokes APO and sets the `-mp` option, which enables recognition of parallel directives inserted into your code.

The `-apolist` option produces a parallelization listing, *file*`.list`.

The `-apokeep` option specifies that *file*`.anl` and *file*`.m` should be retained after compilation and enables `-apolist`. For information on these files, see Section 9.1.2, page 215.

> **Note:** These options are ignored unless you are licensed for the MIPSpro Auto-Parallelizing Option. For more information on this product contact, your sales representative.
>
> The `-apo`, `-apokeep`, and `-apolist` options are equivalent to the `-pfa`, `-pfakeep`, and `-pfalist` options. The `-apo` options are preferred.

The outmoded forms of these options, `-apolist` and `-apo keep`, are still accepted, but the preferred format is the format without the extra space. For more information on APO, see Chapter 9, page 213.

If `-apokeep` is specified in conjunction with `-ipa` or `-IPA`, the default settings for IPA suboptions are used with the exception of the `inline=`*setting* suboption. For that suboption, the default becomes `OFF`. For more information on IPA, see the `ipa`(5) man page.

2.5 `-auto_use` *module_name*[`,`*module_name*] `...`

Directs the compiler to behave as if a `USE` *module_name* statement were entered in your Fortran source code for each *module_name*. The `USE` statements are entered in every program unit and interface body in the source file being compiled.

Note: Using this option can add to compile time in some situations.

2.6 `-c`

Disables the load step and writes the binary object file to *file*`.o`.

For example, the following command line produces file `more.o`:

```
% f90 -c more.f
```

2.7 `-C`, `-check_bounds`

Performs run–time array subscript range checking. Subscripts that are out of range cause fatal run–time errors. If you set the `F90_BOUNDS_CHECK_ABORT` environment variable to `YES`, the program aborts.

These options are equivalent to the `-DEBUG:subscript_check` option. For more information on this option, see the `debug_group`(5) man page.

2.8 `-chunk=`*integer*

When compiling a multitasked program, this option specifies the number of loop iterations per chunk. For scheduling purposes, the iterations of a loop are broken up into pieces. This option must be specified in conjunction with the `-mp` option.

Specify a nonzero, unsigned, positive integer for *integer*. There is no default value for *integer*.

2.9 `-cif`

Generates a compiler information file (CIF) for use by the programming tools. For more information on CIF, see the *Compiler Information File (CIF) Reference Manual*.

2.10 `-col`*n*

Specifies the line width for fixed-format source lines. Specify 72, 80, or 120 for *n*. By default, fixed-format lines are 72 characters wide. Specifying `-col120` implies `-extend_source` and recognizes lines up to 132 characters wide.

For more information on specifying line length, see the `-extend_source` and `-noextend_source` options.

2.11 `-cord`

Runs the procedure rearranger, `cord`(1), on the resulting file after loading. The rearrangement is done to reduce virtual memory paging and/or instruction cache misses.

For more information on procedure rearranging, see the `cord`(1), `pixie`(1), and `prof`(1) man pages.

2.12 `-cpp`

Runs a nondefault source preprocessor, `cpp`(1), on all input source files, regardless of suffix, before compiling. This preprocessor automatically expands macros outside of preprocessor statements.

The default is to run the Fortran preprocessor if the intput file ends in a `.F` or `.F90` suffix.

For more information on source preprocessing compiler options, see the following options: `[-D`*var*[=*def*]`[,`*var*[=*def*]]...], `-E`, `-ftpp`, `-macro_expand`, `-nocpp`, `-P`, and `-U`*var*.

For information on source preprocessing and the macros available, see Chapter 7, page 185.

2.13 `-cray_mp`

Specifies that all Autotasking directives described in Appendix C, page 253, should be recognized. These directives are also implemented in the CF90 compiler on UNICOS systems. The prefix for these directives is `!MIC$`.

Note: The Autotasking directives are outmoded. The preferred alternatives are the OpenMP Fortran API directives described in Chapter 4, page 89.

You must specify this option if you want the following directives to be recognized in your code: `DOALL`, `DOPARALLEL`, `ENDDO`, `[END]GUARD`, `[END]PARALLEL`, `[END]CASE`, and `NUMCPUS`. For more information on these directives, see Appendix C, page 253.

It is not necessary to specify this option in order for the following two directives to be recognized: `PERMUTATION` and `CNCALL`. These two directives are recognized even when `-cray_mp` is not specified.

This option can be specified on the command line along with `-apo` or `-pfa`, but it cannot be specified along with `-mp`.

2.14 `-d`*n*

Specifies the `KIND` specification used for objects declared `DOUBLE COMPLEX` and `DOUBLE PRECISION`, as follows:

Option	`KIND` value
`-d8`	Uses `REAL(KIND=8)` for objects declared as `DOUBLE PRECISION`. Uses `COMPLEX(KIND=8)` for objects declared `DOUBLE COMPLEX`. Default.
`-d16`	Uses `REAL(KIND=16)` for objects declared as `DOUBLE PRECISION`. Uses `COMPLEX(KIND=16)` for objects declared `DOUBLE COMPLEX`.

2.15 `-D`*var*[`=`*def*][`,`*var*[`=`*def*]]`...`

Defines variables used for source preprocessing as if they had been defined by a `#define` directive. If no *def* is specified, 1 is used. For information on undefining variables, see the `-U`*var* option.

For more information on source preprocessing compiler options, see the following options: `-cpp`, `-E`, `-ftpp`, `-macro_expand`, `-nocpp`, `-P`, and `-U`*var*.

For information on source preprocessing and the macros available, see Chapter 7, page 185.

2.16 -DEBUG:...

Controls the compiler's attempts to detect various errors (at compile time or run time) and controls how the errors are reported. For more information on the debugging options, see the `debug_group`(5) man page.

2.17 -default64

Sets the sizes of default integer, real, logical, and double precision objects to be the same as if the program were executing on a UNICOS system. This option causes the following options to go into effect: `-r8`, `-i8`, `-d16`, and `-64`.

Calling a routine in a specialized library, such as SCSL, requires that its 64-bit entry point be specified when 64-bit data are used. Similarly, its 32-bit entry point must be specified when 32-bit data are used.

2.18 -E

Run only the source preprocessor files, without considering suffixes, and writes the result to `stdout`. This option overrides the `-nocpp` option. The output file contains line directives. To generate an output file without line directives, see the `-P` option.

For more information on source preprocessing compiler options, see the following options: `-cpp`, `-D`*var*[=*def*][`,` *var*[=*def*]]..., `-ftpp`, `-macro_expand`, `-nocpp`, `-P`, and `-U`*var*.

For information on source preprocessing and the macros available, see Chapter 7, page 185.

2.19 -extend_source

Specifies a 132-character line length for fixed-format source lines. By default, fixed-format lines are 72 characters wide. For more information on controlling line length, see the `-col`*n* option

2.20 -fb*file*

Specifies the feedback file to be used. This file (with the suffix `.cfb`) can be produced by `prof`(1) with its `-feedback` option from one or more `.Counts` files generated by the execution of the instrumented program produced by `pixie`(1).

2.21 `-fixedform`

Treats all input source files, regardless of suffix, as if they were written in fixed source form. By default, only input files suffixed with `.f` or `.F` are assumed to be written in fixed source form.

2.22 `-flist`

Invokes all Fortran listing control options. Shows lowering, versioning, and tilling. The effect is the same as if all `-FLIST:...` options had been enabled.

2.23 `-FLIST:...`

Invokes the Fortran listing control group, which controls production of the compiler's internal program representation back into Fortran code, after IPA inlining and loop-nest transformations. This is used primarily as a diagnostic tool, and the generated Fortran code may not always compile.

The following sections describe the individual `-FLIST:...` options.

2.23.1 `-FLIST:=`*setting*

Enables or disables the listing. *setting* can be either `ON` or `OFF`. The default is `OFF`.

This option is enabled when any other `-FLIST:...` options are enabled, but it can also be used to enable a listing when no other options are enabled.

2.23.2 `-FLIST:ansi_format=`*setting*

Sets ANSI format. *setting* can be either `ON` or `OFF`. When set to `ON`, the compiler uses a space (instead of tab) for indentation and a maximum of 72 characters per line. The default is `OFF`.

2.23.3 -FLIST:emit_pfetch=*setting*

Writes prefetch information, as comments, in the transformed source file. *setting* can be either `ON` or `OFF`. The default is `OFF`.

In the listing, `PREFETCH` identifies a prefetch and includes the variable reference (with an offset in bytes), an indication of read/write, a stride for each dimension, and a number in the range from 1 (low) to 3 (high), which reflects the confidence in the prefetch analysis. `prefetch` identifies the reference(s) being prefetched by the `PREFETCH` descriptor. The comments occur after a read/write to a variable and note the identifier of the `PREFETCH`-spec for each level of the cache.

2.23.4 -FLIST:emit_omp=*setting*

Controls whether or not code written to listings and intermediate files is written using OpenMP Fortran API directives. *setting* can be either `ON` or `OFF`. The default is `ON`.

When `ON` is in effect, which is the default, all generated files are written using OpenMP directives. When `OFF` is in effect, all generated files are written using the outmoded MIPS multiprocessing directives.

2.23.5 -FLIST:ftn_file=*file*

Writes the program to *file*. By default, the program is written to *file*`.w2f.f`.

2.23.6 -FLIST:linelength=*n*

Sets the maximum line length to *n* characters.

2.23.7 -FLIST:show=*setting*

Writes the input and output filenames to `stderr`. *setting* can be either `ON` or `OFF`. The default is `ON`.

2.24 -freeform

Treats all input source files, regardless of suffix, as if they were written in free source form. By default, only input files suffixed with `.f90` or `.F90` are assumed to be written in free source form.

2.25 `-ftpp`

Runs the Fortran source preprocessor on input Fortran source files that are suffixed with `.f` or `.f90` before compiling. By default, only files suffixed with `.F` or `.F90` are run through the Fortran source preprocessor.

The Fortran source preprocessor does not automatically expand macros outside of preprocessor statements, so you need to specify `-macro_expand` if you want macros expanded.

If `-ftpp` and `-P` are specified, the preprocessed source code is placed in *file*`.i`, and *file*`.i` does not contain # lines.

For more information on source preprocessing compiler options, see the following options: `-cpp`, `-D`*var*[=*def*][`,`*var*[=*def*]]..., `-E`, `-macro_expand`, `-nocpp`, `-P`, and `-U`*var*.

For information on source preprocessing and the macros available, see Chapter 7, page 185.

2.26 `-fullwarn`

Requests that the compiler generate comment-level messages. These messages are suppressed by default. This option can be useful during software development.

2.27 `-G`*num*

Specifies the maximum size, in bytes, of a data item that is to be accessed from the Global Pointer (GP). *num* must be a decimal number.

If *num* is 0, no data is accessed. The default value for *num* is 8 bytes. Data stored relative to the GP can be accessed by the program quickly, but this space is limited. Large programs can overflow the space accessed by the GP at load time.

If the loader gives the `Bad -G num value` error message, recompile the program with `-G0`. Use the same value for this option or for compiling all files that comprise a program executable or DSO.

2.28 `-g`*debug_lvl*

Generates debugging information and establishes a debugging level. Specify one of the following:

Option	Support
`-g0`	No debugging information produced. Default.
`-g2`, `-g`	Information for symbolic debugging is produced, and optimization is disabled.
`-g3`	Information for symbolic debugging of fully optimized code is produced. The debugging information produced may be inaccurate. This option can be used in conjunction with the `-O`, `-O1`, `-O2`, and `-O3` options.

2.29 `-help`

Lists all available options. The compiler is not invoked.

To list all suboptions within an option group, specify `-LIST:all_options=ON`. This shows, for example, all the suboptions to the `-TENV:`, `-OPT:`, and `-LNO:` options. For more information on the `LIST:` option, see Section 2.41, page 22.

2.30 `-i`*n*

Specifies the length of default integer constants, default integer variables, and logical quantities. Specify one of the following:

Option	Action
`-i4`	Specifies 32-bit (4-byte) objects. Default.
`-i8`	Specifies 64-bit (8-byte) objects. Also see the `-default64` option.

2.31 `-I`*dir*

Specifies a directory to be searched for the following types of files:

- Files named in `INCLUDE` lines in the Fortran source file that do not begin with a slash (`/`) character

- Files named in `#include` source preprocessing directives that do not begin with a slash (`/`) character
- Files specified on `USE` statements

Files are searched in the following order: first, in the directory that contains the input file; second, in the directories specified by *dir*; and third, in the standard directory, `/usr/include`.

2.32 `-ignore_suffix`

Compiles all files as if they were Fortran source files. By default, the `f90`(1) command determines the type of processing necessary for an input file based in its suffix. Files that end in `.c`, for example, are compiled by `cc`(1). When `-ignore_suffix` is specified, the compiler processes all files named as if they were Fortran source files, regardless of suffix.

2.33 `-INLINE:…`

Specifies actions for the standalone inliner. These options control the application of subprogram inlining within one file when interprocedural analysis (IPA) is not enabled.

If you have included inlining directives in your source code, the `-INLINE` option must be specified in order for those directives to be recognized.

For more information on the individual options in this group, see `ipa`(5).

2.34 `-ipa`

Invokes interprocedural analysis (IPA). Specifying this option is identical to specifing `-IPA` or `-IPA:`. Default settings for the individual IPA suboptions are used.

2.35 `-IPA`[`:…`]

Controls the application of interprocedural analysis (IPA) and optimization. This includes inlining, common block array padding, constant propagation, dead function elimination, alias analysis, and other features. Specify `-IPA` with no arguments to invoke the interprocedural analysis phase with default options.

If you have included IPA directives in your source code, the `-IPA` option must be specified in order for those directives to be recognized.

If you compile and load in distinct steps, you must use at least `-IPA` for the compile step, and you must specify `-IPA` and the individual options in the group for the load step. For more information on the individual options in this group, see the `ipa`(5) man page.

2.36 -keep

Writes all intermediate compilation files. *file*`.s` contains the generated assembly language code. *file*`.i` contains the preprocessed source code.

These files are retained after compilation is finished.

If IPA is in effect and you want to retain *file*`.s`, you must specify `-IPA:keeplight=OFF` in addition to `-keep`.

2.37 -KPIC

Generates position-independent code (PIC), which is necessary for programs loaded with dynamic shared libraries. Enabled by default.

2.38 -l*library*

Searches the library named `lib`*library*`.a` or `lib`*library*`.so`. Libraries are searched in the order given on the command line.

If you are using another compiler, for example the C compiler, to load Fortran object files, you need to explicitly specify to the C compiler that the Fortran libraries be loaded.

The following table shows the Fortran libraries that the `f90`(1) command loads by default.

`-l` option	Link library	Content
`-lfortran`	`/usr/lib*/libfortran.so`	Intrinsic procedure, I/O, multiprocessing, IRIX interface, and indexed sequential access method library for shared loading and compiling.
`-lm`	`/usr/lib*/libm.so`	Mathematics library.

Example 1. In the following example, the `cc`(1) command loads Fortran object files. The `-l` option loads the Fortran library files:

```
cc -o myprog main.o rest.o -lfortran -lm
```

See the `ld`(1) man page for information on specifying the `-l` option.

Example 2. You may need to specify libraries when you use IRIX system packages that are not part of a particular language. Most of the man pages for these packages list the required libraries. For example, the `getwd`(3C) subroutine requires the BSD compatibility library `libbsd.a`. Specify this library as follows:

```
% f90 main.o more.o rest.o -lbsd
```

Example 3. To load the Silicon Graphics/Cray Scientific Library (SCSL), specify one of the following command lines:

```
% f90 -lscs sci.f
```

or

```
% f90 -lscs_mp mpsci.f
```

The `-lscs_mp` option used in the preceding command line loads the multiprocessed version of SCSL, which is supported on Origin series systems.

Example 4. To specify a library created with the archiver, type in the path name of the library as follows:

```
% f90 main.o more.o rest.o libfft.a
```

Note: The loader searches libraries in the order you specify. Therefore, if you have a library named `libfft.a` that uses data or procedures from `-lfourier`, you **must** specify `libfft.a` first.

2.39 `-L`*directory*

Changes the library search algorithm for the loader. For *directory*, specify the path to a directory that should be searched before using the default system libraries. You can specify multiple `-L` options on the command line. The library search algorithm searches these directories in left to right order.

2.40 `-LANG:...`

Controls the language option group. The following sections describe the suboptions available in this group.

2.40.1 `-LANG:heap_allocation_threshold=`*size*

Determines heap or stack allocation. If the size of an automatic array or compiler temporary variable exceeds *size* bytes, it is allocated on the heap instead of the stack. If *size* is `-1`, objects are always put on the stack. If *size* is `0`, objects are always put on the heap. The default is `-1`, which allows for maximum performance and for compatibility with previous releases.

2.40.2 `-LANG:IEEE_minus_zero=`*setting*

Controls whether or not a minus sign (`-`) is written for negative zero. Specify either `ON` or `OFF` for *setting*. The default is `OFF`, which suppresses the minus sign. The minus sign is suppressed by default to prevent problems from hardware instructions and optimizations that can return a `-0.0` result from a `0.0` value.

2.40.3 `-LANG:recursive=`*setting*

Invokes the language option control group to control recursion support. *setting* can be either `ON` or `OFF`. The default is `OFF`.

In either mode, the compiler supports a recursive, stack-based calling sequence. The difference lies in the optimization of statically allocated local variables, as follows:

- With `-LANG:recursive=ON`, the compiler assumes that a statically allocated local variable could be referenced or modified by a recursive procedure call. Therefore, such a variable must be stored into memory before making a call and reloaded afterwards.
- With `-LANG:recursive=OFF`, the compiler can safely assume that a statically allocated local variable is not referenced or modified by a procedure call. This setting enables the compiler to optimize more aggressively.

2.41 -LIST:...

Writes an assembler listing file to *file*`.l`. If the `-S` option is also in effect, the content of this listing is also written to the assembly language file (*file*`.s`).

Note: For information on how to obtain a source listing and cross reference, see the `-listing` option.

The following sections describe the individual `-LIST:` options.

Note: If `-LIST:` is not specified, all the suboptions described in the following sections are set to `OFF`.

2.41.1 -LIST:=*setting*

Writes or suppresses the listing file. Specify `ON` or `OFF` for *setting*.

If one or more `-LIST` options are enabled, the listing file is written. By default, the listing file contains a list of compiler options in effect during compilation.

2.41.2 -LIST:all_options=*setting*

Writes or suppresses the list of all supported options in the listing file. Specify `ON` or `OFF` for *setting*. The default is `OFF`.

2.41.3 -LIST:notes=*setting*

Writes or suppresses notes regarding various optimization phases in the assembly listing file (*file*`.s`). Must be specified in conjunction with `-S`. Specify `ON` or `OFF` for *setting*. The default is `ON`.

2.41.4 `-LIST:options=`*setting*

Writes or suppresses a listing of the compiler options in effect during compilation in the listing file. Specify `ON` or `OFF` for *setting*. The default is `OFF`.

2.41.5 `-LIST:symbols=`*setting*

Writes or suppresses a listing of the internal compiler symbol tables used in the compilation in the listing file. Specify `ON` or `OFF` for *setting*. The default is `OFF`.

2.42 `-listing`

Writes a source code listing and a cross reference listing to *file*`.L`.

2.43 `-LNO:…`

Specifies options and transformations performed on loop nests by the Loop Nest Optimizer. The `-LNO` options are enabled only if `-O3` is also specified on the `f90`(1) command line.

The arguments to `-LNO` are divided into the following groups:

- General options
- Transformation options
- Cache memory management options
- Translation Lookaside Buffer (TLB) options
- Prefetch options

For information on the LNO options that are in effect during a compilation, specify `-LIST:all_options=ON`.

The following sections describe the individual LNO options.

2.43.1 General Options

The following sections describe the general options.

2.43.1.1 `-LNO:auto_dist=`*setting* (Origin Series Only)

Distributes local arrays and arrays in common blocks that are accessed in parallel. Specify `ON` or `OFF` for *setting*. The default is `OFF`.

When `-LNO:auto_dist=ON`, the compiler distributes local and `COMMON` arrays that are accessed in parallel based on access patterns inside the routines that contain definitions of arrays (as opposed to array declarations). Access patterns of arrays used as dummy arguments are ignored. This optimization works with either automatic parallelism or parallelism expressed through directives. This optimization is always safe, does not affect the layout of arrays in virtual space. and does not incur addressing overhead.

Example:

```
      PROGRAM FRED
      REAL A(1000,100)
      COMMON A
!$OMP PARALLEL DO PRIVATE (I,J)
      DO I=1,N
        DO J=1,N
          A(J,I) = 0.0
        END DO
      END DO
      END
```

In the preceding code fragment, every processor accesses a block of iterations of parallel loop `I`. This implies that every processor will zero a block of columns of array `A`. When this option is enabled, the compiler distributes the array using the `!$SGI DISTRIBUTE A(*,BLOCK)` directive so that each processor accesses data local to its own memory. The compiler might not pick the best distribution. In particular, if arrays are accessed differently in different subroutines, the distribution is that which suites the majority. This option is useful for programs that are not written with data distribution in mind. For more information on the `DISTRIBUTE` directive, see Section 5.2.1, page 150.

2.43.1.2 `-LNO:gather_scatter=`*n*

Performs gather-scatter optimizations. Specify 0, 1, or 2 for *n*. The default is 1.

`gather_scatter=0` disables all gather-scatter optimization. `gather_scatter=1` performs gather-scatter optimizations on non-nested `IF` statements. `gather_scatter=2` performs multilevel gather-scatter optimizations.

The following code fragment shows gatter-scatter optimization:

```
SUBROUTINE SUB(N)
COMMON/BLK/A(1000),B(1000),C(1000),INDEX(1000)
DO J = 1,N
  IF(A(J) .EQ. B(J)) THEN
      C(J) = SQRT(A(J))
  END IF
END DO
END
```

The compiler transforms this as follows:

```
  INC = 0
  DO J = 1, N
    ITEMP(INC + 1) = J
    IF(A0(J) .EQ. B0(J)) THEN
      INC = INC + 1
    ENDIF
  END DO
  DO IND_0 = 0, INC_0 + -1
    J_TMP = ITEMP(IND_0 + 1)
    C0(J_TMP) = SQRT(A0(J_TMP))
  END DO
```

2.43.1.3 `-LNO:ignore_pragmas=`*setting*

Specifies that the command line options override directives in the source file. Specify either `ON` or `OFF` for *setting*. The default is `ignore_pragmas=OFF`.

By default, directives within a file override command line options.

2.43.1.4 `-LNO:oinvar=`*setting*

Controls outer loop hoisting. *Hoisting* is the process by which invariant statements or expressions are taken out of a loop. The compiler looks for expressions that vary in the inner loop but are invariant in an outer loop. The compiler precomputes all the invariant expressions and stores them in a temporary array. All references to the expression in the inner loop are replaced by loads from the array. Specify `ON` or `OFF` for *setting*. The default is `oinvar=ON`.

2.43.1.5 `-LNO:opt=`*n*

Controls the LNO optimization level. Specify either 0 or 1 for *n*. The default is 1.

`opt=0` disables nearly all loop nest optimization. `opt=1` performs full LNO transformations.

2.43.1.6 `-LNO:outer=`*setting*

Enables or disables outer loop fusion. Specify `ON` or `OFF` for *setting*. The default is `outer=ON`.

For more information on controling loop fusion, see the `-LNO:fusion` option.

2.43.1.7 `-LNO:parallel_overhead=`*num_cycles*

Overrides internal compiler estimates concerning the efficiency to be gained by executing certain loops in parallel rather than serially. *num_cycles* specifies the number of processor cycles. Specify an integer for *num_cycles*. The default is 2600.

When the `-apo` or `-pfa` options are in effect, loops in a program are parallelized automatically. The compiler tests each `DO` loop to ensure that there is enough work in the loop to make it worth executing in parallel. Generally, the testing performed by the compiler evaluates each loop as follows:

```
IF ((work_per_processor(N, P) + parallel_overhead) < total_work_in_loop(N)) THEN
   perform parallel execution
   ELSE
      perform serial execution
   END IF
```

The *work_per_processor*, *parallel_overhead*, and *total_work_in_loop* are compiler-generated estimates, expressed in machine cycles, as follows:

- The *work_per_processor* depends on the loop's trip count, *N* (a value that is known just before the loop is executed), and the number of processors, *P*, upon which the loop is to be executed.
- The *total_work_in_loop* depends only on *N*, the loop's trip count.
- The *parallel_overhead* represents the costs to initiate execution of the loop by all the threads and to synchronize them at the end. This value is 2600 cycles by default.

The `-LNO:parallel_overhead=`*num_cycles* option changes *parallel_overhead* to *num_cycles* cycles.

As *parallel_overhead* increases, it becomes less likely that loops will run in parallel. Increasing the value of *parallel_overhead* is useful when the parallelized loop actually runs more slowly than the serial version, but the *work_per_processor* is underestimated, causing the loop to run in parallel and suffer a slowdown.

Conversely, as *parallel_overhead* decreases, it becomes more likely for the loop to run in parallel. Decreasing *parallel_overhead* is useful if the loop runs faster in parallel than in serial, but the *work_per_processor* determination overestimates the actual execution time and causes the slower serial version of the loop to be executed.

2.43.1.8 `-LNO:pure=`*n*

Specifies the extent to which the compiler should consider the effect of a `PURE` procedure or a `!DIR$ NOSIDEEFFECTS` directive when performing parallel analysis. Specify `0`, `1`, or `2` for *n*, as follows:

*n*Value	Description
`0`	Directs the compiler to ignore the fact that a `PURE` procedure or a procedure preceded by a `!DIR$ NOSIDEEFFECTS` directive does not modify global data or its arguments.
`1`	Directs the compiler to consider the fact that `PURE` procedures and procedures preceded by a `!DIR$ NOSIDEEFFECTS` directive do not modify global data or procedure arguments when performing parallel analysis. Default.
`2`	Asserts to the compiler that that `PURE` procedures and procedures preceded by a `!DIR$ NOSIDEEFFECTS` directive do not modify global data, do not modify procedure dummy arguments, and do not access global data. This setting asserts that the only non-local data items referenced by the procedure are the dummy arguments to the procedure. This is an extension of the Fortran standard meaning of `PURE` and of the meaning of `!DIR$ NOSIDEEFFECTS`. At this setting, more

aggressive parallelization can occur if procedures are known not to access global data.

2.43.1.9 `-LNO:vintr=`*setting*

Specifies that vectorizable versions of the math intrinsic functions should be used. Vector versions of routines return multiple results per call, reducing the number of calls made in a loop and, thus, the call over head. Specify `ON` or `OFF` for *setting*. The default is `vintr=ON`.

For information on the math intrinsic functions, see the `math`(3M) man page.

2.43.2 Transformation Options

The loop transformation options described in the following sections allow you to control cache blocking, loop fission, loop fusion, loop unrolling, and loop interchange.

2.43.2.1 `-LNO:blocking=`*setting*

Specifies whether cache blocking is performed.

Specify `blocking=OFF` to disable cache blocking. Cache blocking is performed to improve reuse of data in cache. Specify `ON` or `OFF` for *setting*. The default is `blocking=ON`.

For more information on blocking, see the *MIPSpro Compiling and Performance Tuning Guide*.

2.43.2.2 `-LNO:blocking_size=`*n1*`[,`*n2*`]`

Specifies a code blocking size that the compiler must use when performing any blocking. Specify a value for *n2* when using a 2-level cache. For *n1* or *n2*, enter an integer number that represents the number of iterations.

2.43.2.3 `-LNO:fission=`*n*

Controls loop fission. Specify 0, 1, or 2 for *n*. The default is 1.

Loop fission is an optimization process by which a loop is divided into smaller, independent loops. This can improve register use for large inner loops. It also enables other optimizations, such as loop interchange and blocking, to execute more efficiently. Consider the following loop:

```
DO I ...
   DO J1 ...
      ...
   ENDDO
   DO J2 ...
      ...
   ENDDO
ENDDO
```

With loop fission, the preceding loop is transformed into the following two loops:

```
DO I1 ...
   DO J1 ...
      ...
   ENDDO
ENDDO
DO I2 ...
   DO J2 ...
      ...
   ENDDO
ENDDO
```

`fission=0` disables loop fission. `fission=1` performs normal fission as necessary. `fission=2` specifies that fission be tried before fusion.

If `-LNO:fission=`*n* and `-LNO:fusion=`*n* are both set to 1 or to 2, fusion is performed.

2.43.2.4 `-LNO:fusion=`*n*

Controls loop fusion. *Loop fusion* is an optimization process by which two small loops are transformed into one larger loop. Loop fusion can lower the number of memory references and improve cache behavior. It also enables other optimizations, such as loop interchange and cache blocking, to execute more efficiently. Specify 0, 1, or 2 for *n*. The default is 1. The loops to be fused need not have identical iteration counts, but the iteration counts should be approximately the same.

Consider the following loop:

```
DO I = 1,N
   DO J = 1,N
      A(I,J) = B(I,J) + B(I,J-1) + B(I,J+1)
   END DO
END DO
DO I = 1,N
   DO J = 1,N
      B(I,J) = A(I,J) + A(I,J-1) + A(I,J+1)
   END DO
END DO
```

With loop fusion, the preceding loops are transformed into the following loop:

```
DO I=1,N
   A(I,1) = B(I,0) + B(I,1) + B(I,2)
   DO J = 2,N
      A(I,J) = B(I,J) + B(I,J-1) + B(I,J+1)
      B(I,J-1) = A(I,J-2) + A(I,J-1) + A(I,J)
   END DO
   B(I,N) = A(I,N-1) + A(I,N) + A(I,N+1)
END DO
```

`fusion=0` disables loop fusion. `fusion=1` performs standard outer loop fusion. `fusion=2` specifies that outer loops should be fused, even if it means partial fusion. The compiler attempts fusion before fission. The compiler performs partial fusion if not all levels can be fused in the multiple-level fusion.

If `-LNO:fission=`*n* and `-LNO:fusion=`*n* are both set to 1 or to 2, fusion is performed. For information on controling outer loop fusion, see the `-LNO:outer` option.

The `fusion=` options affect the singly nested loops produced by the compiler.

2.43.2.5 `-LNO:fusion_peeling_limit=`*n*

Sets the limit for the number of iterations allowed to be peeled, where $n \geq 0$. By default, `fusion_peeling_limit=5`.

Loops that are candidates for loop fusion must have identical iteration counts. *Loop peeling* is an optimization that the compiler may need to perform on loops prior to loop fusion. For example, consider the following loops:

```
DO I = 1,N     ! loop 1
  . . .
END DO

DO I = 1,N-1     ! loop 2
  . . .
END DO
```

In the preceding example, the iteration counts of `loop 1` and `loop 2` differ. The compiler removes (peels) one iteration from `loop 1`; fuses `loop 1` and `loop 2`; and executes the peeled iteration from `loop 1` separately from the resulting fused loop. In this example, one iteration was peeled. The default maximum number of iterations that can be peeled is five iterations. This option allows you to specify a different maximum number of iterations that the compiler can peel.

2.43.2.6 `-LNO:interchange=`*setting*

Specifies whether loop interchange optimizations are performed.

Loop nests such as the following benefit from loop interchange optimizations:

```
DO I ...
   DO J ...
      DO K ...
         A(J,K) = A(J, K) + B(I,K)
      END DO
   END DO
END DO
```

In the preceding loop, each iteration of loop `K` requires two loads and one store. Also, if the loop bounds are large, every memory reference results in a cache miss.

With `-LNO:interchange=ON`, the loop is transformed into the following loop:

```
DO K ...
   DO J ...
      DO I ...
         A(J,K) = A(J,K) + B(I,K)
      END DO
   END DO
END DO
```

In the new loop, note that `A(J,K)` is a loop invariant entity; only one load is needed per iteration. The new loop is also more efficient with regard to cache management.

Specifying `-LNO:interchange=OFF` disables loop interchange optimizations. Specify `ON` or `OFF` for *setting*. The default is `interchange=ON`.

2.43.2.7 `-LNO:ou=`*n*, `ou_max=`*n*, and `ou_prod_max=`*n*

Specifies aspects of loop unrolling. When a loop is *unrolled*, the compiler makes copies of the loop body and executes them in sequence. The compiler performs some loop unrolling by default, but this option let you override default system assumptions.

Specifying `ou=`*n* indicates that all outer loops for which unrolling is legal should be unrolled *n* times; the result is that the compiler creates *n* copies of the loop. Specify an integer for *n*. The compiler unrolls loops by this amount (if specified) or not at all.

Specifying `ou_max=`*n* indicates that the compiler can unroll as many as *n* copies per loop, but no more.

Specifying `ou_prod_max=`*n* indicates that the product of unrolling of the various outer loops in a given loop nest is not to exceed *n*. The default is 16.

Example. The following loop is compiled with `-LNO:ou=2`:

```
DO I = 1,N
   DO J = 1,N
      A(J,I) = A(J,I) + B(J)
   END DO
END DO
```

After unrolling, the loop is as follows:

```
DO I = 1,N-1,2
   DO J = 1,N
      A(J,I) = A(J,I) + B(J)
      A(J,I+1) = A(J,I+1) + B(J)
   END DO
END DO
DO I = I,N        ! This nest computes remaining iterations.
   DO J = 1,N     ! This is the wind down loop.
      A(J,I) = A(J,I) + B(J)
   END DO
END DO
```

The advantage of unrolling, in the example, is that there is no need to load `B(J)` `N` times but instead `N/2` times.

2.43.2.8 `-LNO:ou_deep=`*setting*

Specifies that for loops with a nesting depth of 3 or more, the compiler should outer unroll the wind-down loops that result from outer unrolling loops further out. This results in a large executable file, but it generates much faster code whenever wind-down loop execution costs are important. The default is `ou_deep=ON`.

2.43.2.9 `-LNO:ou_further=`*n*

Specifies whether the compiler performs outer loop unrolling on wind-down loops. When unrolling a loop with *n* iterations *u* times, the compiler must generate a wind–down loop to handle cases in which *n* is not a multiple of *u*. The *wind-down loop* handles the extra iterations at the end. The wind-down loop will have at most *u*-1 iterations. When the unrolling factor, *u*, is large, it may be beneficial to unroll the wind-down loop itself. When this option is set to *n*, the compiler unrolls a wind-down loop only if the original loop was unrolled by at least a factor of *n*. Specify an integer for *n*.

You can disable additional wind-down unrolling by specifying `-LNO:ou_further=999999`. Unrolling is enabled as much as is sensible by specifying `-LNO:ou_further=3`.

2.43.3 Cache Memory Management Options

LNO does several transformations, such as blocking and loop interchange, to improve the cache behavior of programs. When performing these transformations, LNO assumes that the target platform has certain cache characteristics. The following sections describe suboptions that allow you to change the default cache characteristics, thereby giving finer control over the optimizations that LNO performs.

The cache memory management options allow you to tune up to four aspects of the memory hierarchy on your system. For example, these four levels could include level 1 cache, level 2 cache, the TLB, and main memory.

The numbering in these arguments starts with the cache level closest to the processor and works outward.

2.43.3.1 `-LNO:assoc1=`*n*, `assoc2=`*n*, `assoc3=`*n*, `assoc4=`*n*

Specifies cache set associativity. For example, main memory is a fully associative cache for disk. Set *n* to any sufficiently large number, such as 128. Specifying *n*=0 indicates that there is no cache at that level.

2.43.3.2 `-LNO:cmp1=`*n*, `cmp2=`*n*, `cmp3=`*n*, `cmp4=`*n* and `dmp1=`*n*, `dmp2=`*n*, `dmp3=`*n*, `dmp4=`*n*

Specifies, in processor cycles, the time for a clean or dirty miss to the next outer level of the memory hierarchy. This number is approximate because it depends upon a clean or dirty line, read or write miss, and so on. Specifying *n*=0 indicates that there is no cache at that level.

2.43.3.3 `-LNO:cs1=`*n*, `cs2=n`, `cs3=`*n*, `cs4=`*n*

Specifies the cache size. The value *n* can be 0, or it can be a positive integer followed by one of the following letters: `k`, `K`, `m`, or `M`. This specifies the cache size in kilobytes or megabytes. Specifying *n*=0 indicates that there is no cache at that level.

`cs1` refers to the primary cache. `cs2` refers to the secondary cache. `cs3` refers to memory. `cs4` refers to disk. The default cache size for each type of cache depends on your system. You can specify `-LIST:all_options=ON` to direct the compiler to generate a listing that includes the default cache sizes used during compilation. In addition, you can enter the following command to see the secondary cache size(s) on your system:

```
hinv -c memory | grep Secondary
```

2.43.3.4 `-LNO:is_mem1=`*setting*, `is_mem2=`*setting*, `is_mem3=`*setting*, `is_mem4=`*setting*

Specifies that certain memory hierarchies should be modeled as memory, not cache. Specify `ON` or `OFF` for *setting*. The default is `OFF` for each option.

If an `is_mem`*k*`=`*setting* setting is specified, the corresponding `assoc`*n*`=`*n* specification is ignored. Blocking can be attempted for this memory hierarchy level, and blocking appropriate for memory, rather than cache, is applied. No prefetching is performed, and any prefetching options are ignored. Any `cmp`*n*`=`*n* and `dmp`*n*`=`*n* options on the command line are ignored.

2.43.3.5 `-LNO:local_pad_size=`*n*

Specifies the amount by which to pad local array dimensions. By default, the compiler automatically chooses the amount of padding to improve cache behavior for local array accesses. The unit for *n* is in elements of the original arrays.

2.43.3.6 `-LNO:ls1=`*n*, `ls2=`*n*, `ls3=`*n*, `ls4=`*n*

Specifies the line size, in bytes. This is the number of bytes, specified in the form of an integer number, *n*, that are moved from the memory hierarchy level further out to this level on a miss. Specifying *n*=0 indicates that there is no cache at that level.

2.43.4 Translation Lookaside Buffer (TLB) Options

The following options control the TLB. The TLB is a cache for the page table. Blocking for the TLB can improve cache performance. The following sections describe options that control how the loop nest optimizer models the TLB when performing transformations. The TLB hardware is assumed to be fully associative.

2.43.4.1 `-LNO:ps1=`*n*, `ps2=`*n*, `ps3=`*n*, `ps4=`*n*

Specifies the number of bytes in a page, where *n* is an integer in the range 4000 ≤ *n* ≤ 256000. The default *n* depends on your system hardware, and you can obtain this information through the `getpagesize`(2) system call. For more information on this system call, see the `getpagesize`(2) man page.

2.43.4.2 `-LNO:tlb1=`*n*, `tlb2=`*n*, `tlb3=`*n*, `tlb4=`*n*

Specifies the number of entries in the TLB for this cache level, where *n* is an integer in the range 40 ≤ *n* ≤ 100. The default *n* depends on your system hardware.

2.43.4.3 `-LNO:tlbcmp1=`*n*, `tlbcmp2=`*n*, `tlbcmp3=`*n*, `tlbcmp4=`*n* and `tlbdmp1=`*n*, `tlbdmp2=`*n*, `tlbdmp3=`*n*, `tlbdmp4=`*n*

Specifies the number of processor cycles it takes to service a clean or dirty TLB miss, where *n* is an integer in the range 40 ≤ *n* ≤ 200. The default *n* depends on your system hardware.

2.43.5 Prefetch Options

The following options control use of the prefetch operation. When an LNO prefetch option is enabled, the compiler examines the source code for memory references that can cause cache misses. It then inserts prefetches into the generated code so that the prefetches are performed ahead of the corresponding memory references.

The `-mips4` and `-r10000` (or `-r12000`) options must be in effect in order for the LNO prefetch options to be recognized.

2.43.5.1 `-LNO:pf`*k*`=`*setting*

Selectively disables and enables prefetching for cache level *k*, where $1 \leq k \leq 4$. Specify `ON` or `OFF` for *setting*.

When `-r10000` or `-r12000` is in effect, `pf1=ON` and `pf2=ON` by default. At any other `-r`*n* setting, `OFF` is in effect for all cache levels.

2.43.5.2 `-LNO:prefetch=`*n*

Specifies levels of prefetching.

`prefetch=0` disables all prefetching. This is the default when `-r4000`, `-r5000`, or `-r8000` is in effect.

`prefetch=1` enables conservative prefetching. This is the default when `-r10000` or `-r12000` is in effect.

`prefetch=2` enables aggressive prefetching.

2.43.5.3 `-LNO:prefetch_ahead=`*n*

Prefetches the specified number of cache lines ahead of the reference. The default is 2.

2.43.5.4 `-LNO:prefetch_manual=`*setting*

Specifies whether manual prefetches (through directives) should be respected or ignored. Specify `ON` or `OFF` for *setting*.

`prefetch_manual=OFF` ignores manual prefetches. This is the default when `-r4000`, `-r5000`, or `-r8000` is in effect.

`prefetch_manual=ON` respects manual prefetches. This is the default when `-r10000` or `-r12000` is in effect.

2.44 `-macro_expand`

Enables macro expansion in preprocessed Fortran source files throughout each file.

When `-macro_expand` is specified, macro expansion occurs throughout the source file. When `-macro_expand` is not specified, macro expansion is limited to preprocessor (#) directives in files processed by the Fortran preprocessor.

For more information on source preprocessing compiler options, see the following options: `-cpp`, [`-D`*var*[=*def*][`,`*var*[=*def*]]...], `-E`, `-ftpp`, `-nocpp`, `-P`, and `-U`*var*.

For information on source preprocessing and the macros available, see Chapter 7, page 185.

2.45 `-MDupdate`[*file*]

Updates makefile dependencies in *file*. The file can be included by `smake`(1) and `pmake`(1) to get dependencies. Files named on `INCLUDE` statements and modules named on `USE` statements are updated.

When *file* is not specified, the lines updated are those that begin with the name of the output file, followed by a colon (`:`), and end with a distinctive `make`(1) comment.

When *file* is specified, *file* is updated during compilation to contain header, library, and run-time `make`(1) dependencies for the output file.

For example, assume that file `foo.f90` contains the following two lines:

```
INCLUDE "bar.h"
USE mod
```

The updated file will contain a line similar to the following:

```
foo.o : bar.h MOD.mod
```

2.46 `-mips`*n*

Specifies the Instruction Set Architecture (ISA). Specify `-mips3` to specify the MIPS III instruction set. Specify `-mips4` to specify the MIPS IV instruction set. For information on the default setting for your system, see file `/etc/compiler.defaults`.

The `-mips`*n* option interacts with the `-64` and `-n32` options.

2.47 -mp

Generates multiprocessing code for the files being compiled. This option causes the compiler to recognize all multiprocessing directives and enables all `-MP:...` options.

If you have specified more than one type of multiprocessing directive for an individual loop, you need to disable one or more sets of directives by using the `-MP` option in conjunction with the `-mp` option. Only one set of multiprocessing directives can be recognized for a specific loop. Specifying `-mp` sets all the `-MP` options to `ON`. To disable one or more sets of directives, specify one or more `-MP` options in conjunction with `-mp`.

The following list describes the sets of multiprocessing directives and indicates the command line options needed to selectively disable one or more sets of directives:

- The OpenMP Fortran API multiprocessing directives described in Chapter 4, page 89, and the Silicon Graphics directives that are extensions to OpenMP described in Chapter 5, page 143. These directives begin with the `!$OMP` and `!$SGI` prefixes. To disable these directives, but allow other multiprocessing directive to be recognized, specify `-mp` and `-MP:open_mp=OFF`.
- The Silicon Graphics multiprocessing directives described in Appendix D, page 267. These directives begin with the `!$` and `!$PAR` prefix. These directives are outmoded. To disable these directives, but allow other directives to be recognized, specify `-mp` and `-MP:old_mp=OFF`.
- The Origin series distributed shared memory directives described in Chapter 5, page 143, that begin with a `!$` prefix. These directives are outmoded. To disable these directives, but allow other multiprocessing directives to be recognized, specify `-mp` and `-MP:dsm=OFF`.

At load time, you can specify both object files produced with the `-mp` option and object files produced without it. If any or all of the files are compiled with `-mp`, the executable must be loaded with `-mp` so that the correct libraries are used.

Example 1: Multiprocessor executable. The following command line compiles and loads the Fortran program `foo.f`:

```
% f90 -mp foo.f
```

Example 2: Multiprocessor and optimizer. In the following example, the Fortran routines in the file `snark.f` are compiled with multiprocessing code generation enabled. The optimizer is also used.

```
% f90 -c -mp -O2 snark.f
```

A standard `snark.o` binary is produced, which must be loaded.

```
% f90 -mp -o boojum snark.o bellman.o
```

In this example, the `-mp` option signals the loader to use the Fortran multiprocessing library. The `bellman.o` file did not have to be compiled with the `-mp` option.

After loading, the resulting executable can be run like any executable file. Creating multiple execution threads, running and synchronizing threads, and task termination are all handled automatically.

When an executable file is loaded with `-mp`, the Fortran initialization routines determine how many parallel threads of execution to create. This determination occurs each time the task starts; the number of threads is not compiled into the code. The default is to use either 8 or the number of processors that are on the machine, whichever is less. You can override the default by setting the `OMP_NUM_THREADS` environment variable to a value that is less than or equal to the number of physical processors. If it is set, Fortran tasks use the specified number of execution threads. For more information on the `OMP_NUM_THREADS` environment variable, see `pe_environ`(5).

2.48 -MP:…

Specifies individual multiprocessing options that provide fine control over certain optimizations.

Specifying `-mp` enables all the `-MP:`… options. The `-mp` option must be specified in conjunction with any `-MP:` options in order for the `-MP:` options to be honored.

The following sections describe the `-MP:` options.

2.48.1 -MP:check_reshape=*setting*

Enables or disables run time consistency checks across procedure boundaries when passing reshaped arrays (or portions thereof) as actual arguments. Specify `ON` or `OFF` for *setting*. The default is `check_reshape=OFF`.

2.48.2 `-MP:clone=`*setting*

Enables or disables autocloning. Specify `ON` or `OFF` for *setting*. The compiler automatically duplicates procedures that are called with reshaped arrays as actual arguments for the incoming distribution. If you have explicitly specified the distribution on all relevant dummy arguments, you can disable autocloning. The consistency checking of the distribution between actual and dummy arguments is not affected by this option and is always enabled. The default is `clone=ON`.

For more information on regular and reshaped distribution, see Chapter 5, page 143.

2.48.3 `-MP:dsm=`*setting* (Origin Series Systems Only)

Enables or disables recognition of the Origin series distributed shared memory directives described in Chapter 5, page 143. These directives begin with a `!$` prefix and are outmoded.

Specify `ON` or `OFF` for *setting*. When the `-mp` option is also in effect, the default is `dsm=ON`. When the `-mp` option is not in effect, the default is `dsm=OFF`.

> **Note:** The Origin series distributed shared memory directives that begin with the `!$` prefix are outmoded. Silicon Graphics and Cray Research encourage you to write new codes using the Silicon Graphics directives that are extensions to OpenMP Fortran API. The OpenMP extension directives begin with the `!$SGI` prefix and are otherwise identical to the Origin series distributed shared memory directives.

The effects of this option when used in conjunction with `-mp` are as follows:

Options specified	Directives recognized
`-MP:dsm=ON` and `-mp`	OpenMP multiprocessing directives described in Chapter 4, page 89, and the Silicon Graphics extension directives to OpenMP described in Chapter 5, page 143. Multiprocessing directives described in Appendix D, page 267. Origin series distributed shared memory multiprocessing directives described in Chapter 5, page 143, that begin with the `!$` prefix.

`-MP:dsm=OFF` and `-mp`	OpenMP multiprocessing directives described in Chapter 4, page 89, and the Silicon Graphics extension directives to OpenMP described in Chapter 5, page 143. Multiprocessing directives described in Appendix D, page 267.

When the `-mp` option is specified on the `f90`(1) command line, the compiler silently generates bookkeeping information in the `rii_files` directory. This information is used to implement data distribution directives, as well as perform consistency checks of these directives across multiple source files. To disable the processing of the data distribution directives and not generate the `rii_files`, compile the program with the `-MP:dsm=off` option.

2.48.4 `-MP:old_mp=`*setting*

Enables or disables recognition of the Silicon Graphics multiprocessing directives described in Appendix D, page 267, and the Origin series distributed shared memory directives described in Chapter 5, page 143, that begin with a `!$` prefix. These directives are the loop-level multiprocessing directives (including those for Origin series systems) and the PCF directives. These directives begin with a `!$` or `!$PAR` prefix.

Specify `ON` or `OFF` for *setting*. The default is `ON`.

Note: The Silicon Graphics multiprocessing directives are outmoded. Their preferred alternatives are the OpenMP Fortran API directives described in Chapter 4, page 89.

The effects of this option when used in conjunction with `-mp` are as follows:

Options specified	Directives recognized
`-MP:old_mp=ON` and `-mp`	OpenMP multiprocessing directives described in Chapter 4, page 89, and the Silicon Graphics extension directives to OpenMP described in Chapter 5, page 143. Multiprocessing directives described in Appendix D, page 267 Origin series distributed shared memory multiprocessing directives described in Chapter 5, page 143, that begin with the `!$` prefix.

`-MP:old_mp=OFF` and `-mp`	OpenMP multiprocessing directives described in Chapter 4, page 89, and the Silicon Graphics extension directives to OpenMP described in Chapter 5, page 143.

2.48.5 **`-MP:open_mp=`***setting*

Enables or disables recognition of the OpenMP Fortran API multiprocessing directives described in Chapter 4, page 89, and the Silicon Graphics extensions to OpenMP described in Chapter 5, page 143. These directives begin with a `!$OMP` or a `!$SGI` prefix.

Specify `ON` or `OFF` for *setting*. The default is `ON`.

The effects of this option when used in conjunction with `-mp` are as follows:

Options specified	Directives recognized
`-MP:open_mp=ON` and `-mp`	OpenMP multiprocessing directives described in Chapter 4, page 89, and the Silicon Graphics extension directives to OpenMP described in Chapter 5, page 143.
	Multiprocessing directives described in Appendix D, page 267.
	Origin series distributed shared memory multiprocessing directives described in Chapter 5, page 143, that begin with a `!$` prefix.
`-MP:open_mp=OFF` and `-mp`	Multiprocessing directives described in Appendix D, page 267.
	Origin series distributed shared memory multiprocessing directives described in Chapter 5, page 143, that begin with a `!$` prefix.

2.49 `-mplist`

Generates *file*`.w2f.f`.

Note: Because of data conflicts, do not specify the `-mplist` or `-FLIST` options when `-apokeep` or `-pfakeep` are specified. For more information on `-FLIST`, see Section 2.23, page 14. For more information on `-apokeep` and `-pfakeep`, see Section 2.4, page 9.

2.50 `-mp_schedtype=`*mode*

Specifies a default mode for scheduling work among the participating tasks in loops. This option must be specified in conjunction with `-mp`.

Specifying this option has the same effect as putting a `!$MP_SCHEDTYPE=`*mode* directive at the beginning of the file. Specify one of the following for *mode*:

mode	Action
`DYNAMIC`	Breaks the iterations into pieces, the size of which is specified by the `-chunk=`*integer* option. As each process executes a piece, it enters a critical section and obtains the next available piece. For more information, see the `-chunk=`*integer* option.
`GSS`	Schedules pieces according to the sizes of the pieces awaiting execution.
`INTERLEAVE`	Breaks the iterations into pieces, the size of which is specified by the `-chunk=`*integer* option. Execution of the pieces is interleaved among the processes. For more information, see the `-chunk=`*integer* option.
`RUNTIME`	Schedules pieces according to information contained in the `MP_SCHEDTYPE` environment variable.
`SIMPLE`	Divides the iterations among processes by dividing them into contiguous pieces and assigning one piece to each process. Default.

For more information on environment variables, these modes, and their effects, see the `pe_environ`(5) man page.

2.51 -noappend

Prevents the compiler from appending a trailing underscore character (_) on external names.

2.52 -nocpp

Disables the source preprocessor.

For more information on source preprocessing compiler options, see the following options: `-cpp`, `-D`*var*[=*def*][`,`*var*[=*def*]]..., `-E`, `-ftpp`, `-macro_expand`, `-P`, and `-U`*var*.

For information on source preprocessing and the macros available, see Chapter 7, page 185.

2.53 -noextend_source

Restricts Fortran source code lines to columns 1 through 72. See the `-col`*n* and `-extend_source` options for more information on controlling line length.

2.54 -nostdinc

Directs the system to skip the standard directory, `/usr/include`, when searching for `#include` files and files named on Fortran `INCLUDE` statements.

2.55 -o*out_file*

Writes the executable file to *out_file* rather than to `a.out`. By default, the executable output file is written to `a.out`.

For example, the following command line loads object module `myprog.o` and produces an executable object named `myprog`:

```
% f90 -o myprog myprog.o
```

2.56 `-O`*level*

Specifies the basic optimization level, as follows:

Option	Action
`-O0`	No optimization. Default.
`-O1`	Local optimization.
`-O2, -O`	Extensive optimization. Optimizations performed at this level are almost always beneficial. The execution time is shortened, but compile time may be lengthened.
`-O3`	Aggressive optimization. Optimizations performed at this level may generate results that differ from those obtained when `-O2` is specified.
`-Ofast`[=*ipxx*]	Enables `-O3` and `-ipa`. This option enables optimizations selected to maximize performance for the target platform *ipxx* processor type. To determine a platform *ipxx* designation, use the `hinv`(1) command. The optimizations performed may differ from release to release and among the supported platforms. The optimizations always enable the full instruction set of the target platform (for example, `-mips4` for an R10000). Although the optimizations are generally safe, they may affect floating-point accuracy due to operator reassociation. Typical optimizations selected include those performed at `-O3`. See the `-TARG:platform=`*ipxx* option for more information on the *ipxx* argument. The default is an R10000 POWER CHALLENGE, IP25.

2.57 `-OPT:…`

Controls miscellaneous optimizations. These options override defaults based on the main optimization level.

For information on inlining, see the `-INLINE:`... option. For information on loop nest optimization, see the `-LNO:`... option. For information on interprocedural optimization, see the `-IPA:`... option.

The following sections describe the various general optimization options.

2.57.1 -OPT:alias=*name*

Specifies the pointer aliasing model to be used. By specifying one of the following for *name*, the compiler is able to make assumptions throughout the compilation:

name	Assumption
`parm` or `no_parm`	`parm` asserts that Fortran arguments do not alias to any other variable. Default.
	`no_parm` asserts that Fortran arguments can alias to any other variable.
`cray_pointer` or `no_cray_pointer`	`cray_pointer` asserts that a pointee's storage is never overlaid on another variable's storage. The pointee is stored in memory before a call to an external procedure and is read out of memory as its next reference. It is also stored before a `RETURN` or `END` statement of a subprogram.
	`no_cray_pointer` asserts that a pointee's storage can overlay on another variable's storage. Default.

2.57.2 -OPT:cis=*setting*

Converts SIN/COS pairs with the same argument to a single call that calculates both values at once. Specify `ON` or `OFF` for *setting*. The default is `cis=ON`.

2.57.3 -OPT:cray_ivdep=*setting*

Instructs the compiler to ignore all dependencies when an `IVDEP` directive is encountered. Specify `ON` or `OFF` for *setting*. The default is `OFF`.

For more information on the `IVDEP` directive, see Section 6.6, page 177.

2.57.4 -OPT:div_split=*setting*

Enables or disables the calculation of x/y as $x \times (1.0/y)$. Specify `ON` or `OFF` for *setting*. The default is `div_split=OFF`.

This is enabled by the `-OPT:IEEE_arithmetic=3` option. Also see the `-OPT:recip` option. This option should be used with caution because it produces less accurate results.

2.57.5 -OPT:fast_bit_intrinsics=*setting*

`fast_bit_intrinsics=ON` turns off the check for the bit count being within range for Fortran bit intrinsics (for example, `BTEST` and `ISHFT`). Specify `ON` or `OFF` for *setting*. The default is `fast_bit_intrinsics=OFF`.

2.57.6 -OPT:fast_complex=*setting*

`fast_complex=ON` enables fast calculations for values declared as type complex. When set to `ON`, complex absolute value (norm) and complex division calculations use fast algorithms that can cause overflow for an operand (divisor, in the case of division) that has an absolute value that is larger than the square root of the largest representable floating-point number (or underflow for a value that is smaller than the square root of the smallest representable floating point number).

Specify `ON` or `OFF` for *setting*. The default is `fast_complex=OFF`. `fast_complex=ON` is enabled if `-OPT:roundoff=3` is in effect.

2.57.7 -OPT:fast_exp=*setting*

`fast_exp=ON` optimizes exponentiation by replacing the run-time call for exponentiation by multiplication and/or square root operations for certain compile-time constant exponents (integers and halves). This can produce results that are rounded differently than the run-time routine. `fast_exp=ON` is in effect unless `-OPT:roundoff=1` is in effect.

Specify `ON` or `OFF` for *setting*. The default is `fast_exp=ON`.

2.57.8 -OPT:fast_nint=*setting*

`fast_nint=ON` uses hardware features to implement `NINT` and `ANINT` (both single- and double-precision versions). Specify `ON` or `OFF` for *setting*. The default is `fast_nint=OFF`, but `fast_nint=ON` is enabled by default if `-OPT:roundoff=3` is in effect. `fast_nint=ON` is also enabled when `fast_trunc=ON` is in effect.

When `fast_nint=ON` is in effect, rounding is performed according to the IEEE standard rather than the Fortran standard. For example, the Fortran standard requires that `NINT(1.5)=2` and `NINT(2.5)=3`. The IEEE standard, however, rounds both of these to 2.

2.57.9 `-OPT:fast_sqrt=`*setting*

`fast_sqrt=ON` calculates square roots using the identity `sqrt(x)=x*rsqrt(x)`, where `rsqrt` is the reciprocal square root operation. Specify `ON` or `OFF` for *setting*. The default is `OFF`.

The `-mips4` and `-r8000` options must be in effect in order for `-OPT:fast_sqrt` to be recognized.

Warning: This option results in `sqrt(0.0)` producing a NaN result. Use it only when zero `sqrt` operands are not valid.

2.57.10 `-OPT:fast_trunc=`*setting*

`fast_trunc=ON` inlines the `NINT`, `ANINT`, `AINT`, and `AMOD` Fortran intrinsics, both single- and double-precision versions. Specify `ON` or `OFF` for *setting*. The default is `fast_trunc=OFF`. `fast_trunc=ON` is enabled automatically if `-OPT:roundoff=1` (or greater) is in effect.

Although fully compliant with the Fortran standard, `fast_trunc=ON` reduces the valid argument range somewhat.

If `fast_trunc=ON` is in effect, `fast_nint=ON` is also enabled.

2.57.11 `-OPT:fold_reassociate=`*setting*

`fold_reassociate=ON` allows optimizations involving reassociation of floating-point quantities. Specify `ON` or `OFF` for *setting*. The default is `fold_reassociate=OFF`. `fold_reassociate=ON` is enabled automatically when `-O3` is in effect or when `-OPT:roundoff=2` or greater is in effect.

2.57.12 -OPT:fold_unsafe_relops=*setting*

`fold_unsafe_relops=ON` folds relational operators in the presence of possible integer overflow. Specify `ON` or `OFF` for *setting*. The default is `fold_unsafe_relops=ON`.

2.57.13 -OPT:fold_unsigned_relops=*setting*

`fold_unsigned_relops=ON` folds unsigned relational operators in the presence of possible integer overflow. Specify `ON` or `OFF` for *setting*. The default is `fold_unsigned_relops=OFF`.

2.57.14 -OPT:got_call_conversion=*setting*

`got_call_conversion=ON` loads function addresses to be moved out of loops. The load is set up with the proper relocation so that the address is resolved at program start-up time. Specify `ON` or `OFF` for *setting*. `got_call_conversion=OFF` is the default when `-O2` or lower is in effect. `got_call_conversion=ON` when `-O3` is in effect.

Note: This option should be disabled when compiling shared objects that contain function addresses that may be preempted by `rld`(1). For more information, see the `dso`(5) man page.

2.57.15 -OPT:IEEE_arithmetic=*n*

Specifies the level of conformance to ANSI/IEEE 754-1985, the IEEE Standard for Binary Floating-point Arithmetic, which describes a standard for NaN and inf operands, arithmetic roundoff, and overflow. Specify one of the following for *n*:

n	Description
1	Inhibits optimizations that produce less accurate results than required by ANSI/IEEE 754-1985. This is the default.
2	Allows compiler optimizations that can produce less accurate inexact results (but accurate exact results) on the target hardware. That is, expressions that would have produced a NaN or an inf may produce different answers, but otherwise answers are the same as those obtained when `IEEE_arithmetic=1` is in effect. Examples: `0*X` may be changed to 0, and `X/X` may be changed to 1 even though this is inaccurate when `X` is +inf, -inf, or NaN.

`3`	Performs arbitrary, mathematically valid transformations, even if they can produce inaccurate results for operations specified in ANSI/IEEE 754-1985. These transformations can cause overflow or underflow for a valid operand range. An example is the conversion of `x/y` to `x*recip(y)` for MIPS IV targets. Also see `-OPT:roundoff=`*n*.

2.57.16 `-OPT:IEEE_comparisons=`*setting*

Forces all comparisons to yield results that conform to ANSI/IEEE 754-1985, the IEEE Standard for Binary Floating-point Arithmetic, which describes a standard for `NaN` and `inf` operands. Specify `ON` or `OFF` for *setting*. The default is `IEEE_comparisons=OFF`.

`IEEE_comparisons=OFF` produces non-IEEE results for comparisons. For example, `x=x` is treated as `TRUE` without executing a test.

Note: This option has been deprecated and will be removed in a future release. The preferred alternative is `-OPT:IEEE_NaN_inf=`*setting*.

2.57.17 `-OPT:IEEE_NaN_inf=`*setting*

Forces all operations that might have NaN or inf operands to yield results that conform to ANSI/IEEE 754-1985, the IEEE Standard for Binary Floating-point Arithmetic, which specifies the standard for NaN and inf operands. Specify `ON` or `OFF` for *setting*. The default is `IEEE_NaN_inf=OFF`.

`IEEE_NaN_inf=OFF` produces non-IEEE results for various operations. For example, `x=x` is treated as `TRUE` without executing a test and `x/x` is simplified to 1 without dividing. Turning this option on can suppress many such common optimizations and hurt performance.

2.57.18 `-OPT:inline_intrinsics=`*setting*

`inline_intrinsics=OFF` turns all Fortran intrinsics that have a library function into a call to that function. Specify `ON` or `OFF` for *setting*. The default is `inline_intrinsics=ON`.

2.57.19 `-OPT:liberal_ivdep=`*setting*

Specifies that the compiler should use UNICOS semantics when a `!DIR$ IVDEP` directive is encountered. The compiler ignores all lexically backward loop iteration dependencies. Specify `ON` or `OFF` for *setting*. The default is `OFF`, which directs the compiler to use IRIX semantics when a `!DIR$ IVDEP` directive is encountered.

For more information on the `!DIR$ IVDEP` directive, see Section 6.6, page 177.

2.57.20 **`-OPT:Olimit=`*n***

Specifies that any routine bigger than *n* should not be optimized.

You should use this option if you receive a message indicating that a different `Olimit` value is needed for optimizing your program; the value itself is based on internal compiler calculations. If `-O2` or greater is in effect and a routine is so big that the compile speed can be slow, the compiler generates a message indicating the `Olimit` value that is needed to optimize. You can recompile with that value of *n*, or you can recompile with `-OPT:Olimit=0` and avoid having any `Olimit` cutoff.

2.57.21 **`-OPT:pad_common=`*setting***

`pad_common=ON` reorganizes common blocks to improve the cache behavior of accesses to members of the common block. This may involve adding padding between members and/or breaking a common block into a collection of common blocks. Specify `ON` or `OFF` for *setting*. The default is `pad_common=OFF`.

This option should not be used unless the common block definitions (including `EQUIVALENCE`) are consistent among all sources comprising a program. In addition, `pad_common=ON` should not be specified if common blocks are initialized with `DATA` statements. If specified, `pad_common=ON` must be used for all source files in the program.

`pad_common=ON` is supported for Fortran only. It should not be used if a common block is referenced from C code.

2.57.22 **`-OPT:procedure_reorder=`*setting***

`procedure_reorder=ON` must be specified in conjunction with the `ld`(1) command's `-LD_LAYOUT:reorder_file=`*feedback_file* option to enable linker cording. *Linker cording* is the linker's ability to optimize the layout of functions based upon a feedback file; this minimizes page faults and cache misses. The default is `OFF`.

For more information on the `-LD_LAYOUT` option, see the `ld`(1) man page. For an example that shows reordering of code regions, see the *MIPSpro Compiling and Performance Tuning Guide*.

2.57.23 `-OPT:recip=`*setting*

The `-OPT:recip=`*setting* option causes your program's executable code to conform more closely to the IEEE floating-point standard than the default mode. When specified, many identity optimizations are disabled, executable code is slower, and a scaled complex divide mechanism is enabled that increases the range of complex values that can be handled without producing an underflow.

The `-OPT:recip=`*setting* option causes the compiler to optimize expressions such as `X.NE.X` to false and `X/X` to 1, where `X` is a floating-point value. With `-OPT:recip=`*setting* in effect, these and other similar arithmetic identity optimizations are not performed.

`recip=ON` specifies that faster, but potentially less accurate, reciprocal operations should be performed. Specify `ON` or `OFF` for *setting*. The default is `recip=OFF`. `-r8000` must be in effect in order for `-OPT:recip=ON` to have an effect. If `-O3` or `-OPT:IEEE_arithmetic=2` or above are in effect, `recip=ON` is enabled automatically.

2.57.24 `-OPT:reorg_common=`*setting*

`reorg_common=ON` reorganizes common blocks to improve the cache behavior of accesses to members of the common block. The reorganization is performed only if the compiler detects that it is safe to do so. Specify `ON` or `OFF` for *setting*.

This option produces consistent results for programs that conform to the Fortran standard; for example, programs that do not overindex arrays in common blocks. The optimizations performed are safe even if common blocks are declared differently in different subroutines or if elements in the common block are equivalenced.

`reorg_common=ON` is enabled by default when `-O3` is in effect and when all files that reference the common block are compiled at `-O3`. `reorg_common=OFF` is set when the file that contains the common block is compiled at `-O2` (or below).

2.57.25 **`-OPT:roundoff=`*n***

Specifies the level of acceptable departure from source language floating-point round-off, and overflow semantics. Specify 0, 1, 2, or 3 for *n*. Program performance is best at `roundoff=3`.

`roundoff=0` is the default when optimization levels `-O0`, `-O1`, and `-O2` are in effect. This inhibits optimizations that might affect the floating-point behavior.

`roundoff=1` allows simple transformations that might cause limited round-off or overflow differences. Compounding such transformations could have more extensive effects.

`roundoff=2` is the default level when `-O3` is in effect. This level allows more extensive transformations, such as the reordering of reduction loops.

`roundoff=3` enables any mathematically valid transformation.

To obtain best performance in conjunction with software pipelining, specify `roundoff=2` or `roundoff=3`. This is because reassociation is required for many transformations to break recurrences in loops. Also see the descriptions for `-OPT:IEEE_arithmetic`, `-OPT:fast_complex`, `-OPT:fast_trunc`, and `-OPT:fast_nint`.

2.57.26 **`-OPT:rsqrt=`*setting***

`rsqrt=ON` specifies that faster, but potentially less accurate, reciprocal square root operations may be performed. Specify `ON` or `OFF` for *setting*. The default is `rsqrt=OFF`.

If `-OPT:IEEE_arithmetic=2` or above or `-O3` are in effect, `rsqrt=ON` is enabled.

2.57.27 **`-OPT:space=`*setting***

`space=ON` specifies that code space is to be given priority in tradeoffs with execution time in optimization choices. For instance, this forces all exits from a function to go through a single exit block. Specify `ON` or `OFF` for *setting*. The default is `space=OFF`.

This option can affect loop unrolling size. For more information on this, see Section 2.57.31, page 54.

2.57.28 **-OPT:speculative_ptr_deref=*setting***

This option allows speculative loads of memory locations that differ by a small offset from some referenced memory location.

This option is enabled by default at `-O2` and `-O3`. However, the legal offset ranges are different at each level. At `-O2`, the range is 32 (-16 .. +16). At `-O3`, the range is 128 (-64 .. +64).

This optimization can result in an exception if the speculated location is on a different page than that of the referenced memory location. The chances of this happening with these legal offset ranges is very remote.

2.57.29 **-OPT:swp=*setting***

`swp=ON` enables software pipelining. *Software pipelining* is a compiler code generation technique in which operations from various loop iterations are overlapped in order to exploit instruction-level parallelism, increase the instruction issue rate, and better hide memory and instruction latency. As an optimization technique, software pipelining is similar to *bottom loading*, but it includes additional, and more efficient, scheduling optimizations.

Specify `ON` or `OFF` for *setting*. `swp=ON` is enabled when `-O3` is in effect. The default is `swp=OFF`.

2.57.30 **-OPT:unroll_analysis=*setting***

`unroll_analysis=ON` analyzes resource usage and recurrences in bodies of innermost loops that do not qualify for being fully unrolled. Such loops are unrolled only to the extent for which there is a potential benefit in doing so. A loop could be unrolled, for example, to decrease the shortest possible schedule length per iteration. Specify `ON` or `OFF` for *setting*. The default is `unroll_analysis=ON`.

`unroll_analysis=ON` can have the negative effect of unrolling loops less than the upper limit dictated by the `-OPT:unroll_times_max` and `-OPT:unroll_size` specifications.

2.57.31 **-OPT:unroll_size=*n***

Specifies the maximum size (in instructions) of an unrolled loop. Specify an integer for *n*. When `-OPT:space=OFF` is in effect, the default is `unroll_size=80`. When `-OPT:space=ON` is in effect, the default is `unroll_size=20`. For more information, see Section 2.57.27, page 53.

This option indirectly determines which loops can be fully unrolled. Also see the `-OPT:unroll_times_max` option.

2.57.32 `-OPT:unroll_times_max=`*n*

Specifies the maximum number of times a loop will be unrolled if it is not going to be fully unrolled. Specify an integer for *n*.

The default value of *n* depends on the target processor. The default is `8` when `-r8000`, `-r10000` or `-r12000` are in effect, and the default is `4` in all other cases. Also see the `-OPT:unroll_size` option.

2.57.33 `-OPT:wrap_around_unsafe_opt=`*setting*

Allows you to prevent the compiler from performing potentially unsafe optimizations involving induction variable replacement and linear function replacement. These optimizations are performed by default when `-O2` or `-O3` are specified. Specify `ON` or `OFF` for *setting*.

Setting `wrap_around_unsafe_opt=OFF` disables both the induction variable replacement and linear function test replacement optimizations. These optimizations are safe when loop induction variables do not overflow or wrap around in memory. These optimizations are unsafe when incorrect code is generated due to multiple induction variables in loops having combined initial values that overflow or wrap around in memory. Using this option can degrade performance. It is provided as a diagnostic tool.

2.58 `-P`

Runs only the source preprocessor and puts the results for each source file (that is, for *file*`.f[90]`, *file*`.F[90]`, and/or *file*`.s`) in a corresponding *file*`.i`. The *file*`.i` that is generated does not contain # lines.

For more information on source preprocessing compiler options, see the following options: `-cpp`, [`-D`*var*[=*def*][`,`*var*[=*def*]]...], `-E`, `-ftpp`, `-macro_expand`, `-nocpp`, and `-U`*var*.

For information on source preprocessing and the macros available, see Chapter 7, page 185.

2.59 `-pad_char_literals`

Blank pads all character literal constants that are shorter than the size of the default integer type and that are passed as actual arguments. The padding extends the length to the size of the default integer type.

2.60 `-pfa`, `-pfakeep`, `-pfalist`

The `-pfa`, `-pfakeep`, and `-pfalist` options control the Auto-Parallelizing Option (APO). These options have been superseded by the `-apo`, `-apokeep`, and `-apolist` options.

For more information on the `-apo`, `-apokeep`, and `-apolist` options, see Section 2.4, page 9. For more information on APO, see Chapter 9, page 213.

Note: These options are ignored unless you are licensed for the MIPSpro Auto-Parallelizing Option. For more information on this product contact your sales representative.

If `-pfakeep` is specified in conjunction with `-ipa` or `-IPA`, the default settings for IPA suboptions are used with the exception of the `inline=`*setting* suboption. For that suboption, the default becomes `OFF`. For more information on IPA, see the `ipa`(5) man page.

2.61 `-r`*processor*

Specifies the code scheduler. The `-r` option accepts `4000`, `5000`, `8000`, `10000`, and `12000` as arguments, as follows:

Option	Action
`-r4000`	Schedules code for the R4000 processor.
`-r5000`	Schedules code for the R5000 processor.
`-r8000`	Schedules code for the R8000 processor.
`-r10000`	Schedules code for the R10000 processor.
`-r12000`	Schedules code for the R12000 processor.

Note that these options can also be specified with a `k` substituted for `000`, as follows: `-r8k`, `-r10k`, and so on.

This option adds one of the following to the head of the library search path, where *processor* is as you specified:

- `-L/usr/lib32/mips3/`*processor*
- `-L/usr/lib32/mips4/`*processor*
- `-L/usr/lib64/mips3/`*processor*
- `-L/usr/lib64/mips4/`*processor*

The actual library search path that is added depends on the ABI that is specified or implied. For information on specifying an ABI, see the `-64` and `-n32` options described in Section 2.1, page 6.

2.62 `-r`*real_spec*

Specifies the default kind specification for real values, as follows:

Option	Kind value
`-r4`	Uses `REAL(KIND=4)` and `COMPLEX(KIND=4)` for real and complex variables, respectively. Default.
`-r8`	Uses `REAL(KIND=8)` and `COMPLEX(KIND=8)` for real and complex variables, respectively. You can specify `-r8` when porting programs from UNICOS systems.

2.63 `-S`

Generates an assembly file, *file*`.s`, rather than an object file (*file*`.o`).

2.64 `-static`

Statically allocates all local variables. Statically allocated local variables are initialized to zero and exist for the life of the program. This option can be useful when porting programs from older systems in which all variables are statically allocated.

When compiling with the `-static` option, global data is allocated as part of the compiled object (*file*`.o`) file. The total size of any *file*`.o` cannot exceed 2 GB, but the total size of a program loaded from multiple `.o` files can exceed 2 GB. An individual common block cannot exceed 2 GB, but you can declare multiple common blocks each having that size.

For more information on compiling with large files, see the `-64` and `-n32` options described in Section 2.1, page 6.

If a parallel loop in a multiprocessed program calls an external routine, that external routine cannot be compiled with the `-static` option. You can mix static and multiprocessed object files in the same executable, but a static routine cannot be called from within a parallel region.

2.65 `-static_threadprivate`

Makes all static variables private to each thread. This option can be specified in conjunction with the `-static` option, which statically allocates all local variables.

2.66 `-TARG:…`

Cross compiling is compiling a program on one system and executing it on another. To cross compile, you can either use the `-TARG:` command line options to control the target architecture and machine for which code is generated or you can set the `COMPILER_DEFAULTS_PATH` environment variable to specify the file that contains the default processor information needed to generate executable code for the target system.

The following sections describe cross compiling using both the `-TARG:` options and the `COMPILER_DEFAULTS_PATH` environment variable.

2.66.1 `-TARG:dismiss_mem_faults=`*setting*

Forces the kernel to dismiss any memory faults, such as SIGSEGV or SIGBUS, that occur during execution of the program (not just the code being compiled). This option allows optimizations that might cause extra faults and can slow down execution if extra faults occur. It also prevents recognition of legitimate faults. *setting* can be `ON` or `OFF`. The default is `OFF`.

2.66.2 `-TARG:exec_max=`*letter*

Specifies the maximum set of IEEE-754 floating-point exceptions for which traps may be enabled at run time for the program (not just the code being compiled). The default is `IUOZV`. The default can be affected by the `-TENV:X` option.

This option allows optimizations that might cause extra exceptions, and it may prevent recognition of legitimate faults. It does not affect explicit setting of exception enable flags by the program and should be avoided if the program does this.

Specify zero or more of the following for *letter* to specify exceptions: `I` specifies inexact; `U` specifies underflow; `O` specifies overflow; `Z` specifies divide by zero; and `V` specifies invalid operations.

For related information, see the `-TARG:exc_min` option description. For information on the `-TENV:X` option, see Section 2.67.5, page 63.

2.66.3 **`-TARG:exec_min=`*letter***

Specifies the minimum set of IEEE-754 floating-point exceptions for which traps must be enabled at runtime for the program (not just the code being compiled). The default is none.

This option does not affect explicit setting of exception enable flags by the program and should be avoided if the program does this.

Specify zero or more of the following for *letter* to specify exceptions: `I` specifies inexact; `U` specifies underflow; `O` specifies overflow; `Z` specifies divide by zero; and `V` specifies invalid operations.

For related information, see the `-TARG:exc_max` option description. The `-TARG:exc_max` and `-TARG:exc_min` options specified for the various files that comprise a program must be consistent; for example, none of the `-TARG:exc_min` values may require exceptions disabled by `-TARG:exc_max` values.

2.66.4 **`-TARG:fp_precise=`*setting***

Forces the target processor into precise floating-point mode at execution time. Using this option to compile any component source files of a program invokes this feature in the resulting program. Specify `ON` or `OFF` for *setting*. The default is `OFF`.

This option is only meaningful when `-r8000` is in effect. It can cause significant performance degradation for programs with heavy floating-point usage. For more information on floating-point mode, see the `fpmode`(1) man page.

2.66.5 **`-TARG:isa=`*instruction_set***

Identifies the target instruction set architecture for compilation, such as the set of instructions that are generated. For *instruction_set* specify `mips3` or `mips4`. Specify `-TARG:isa=mips3` for code that must run on R4000 processors. This option is equivalent to specifying `-mips3` or `-mips4`. For information on defaults, and for information on the `-mips`*n* option, see Section 2.46, page 37.

2.66.6 **`-TARG:madd=`*setting***

Enables or prevents transformations from using multiply and add instructions. Specify `ON` or `OFF` for *setting*. The default is `ON`. This option is ignored unless `-mips4` is in effect.

These instructions perform a multiply/add with a single round off. They are more accurate than the usual discrete operations, and they may cause results not to match baselines from other targets. Use this option to determine whether observed differences are due to multiply/add instructions.

2.66.7 **`-TARG:platform=`*ipxx***

Specifies the target platform for compilation, choosing various internal parameters (such as cache sizes) appropriately. Supported values are as follows: `ip19`, `ip20`, `ip21`, `ip22_4k`, `ip22_5k`, `ip24`, `ip25`, `ip26`, `ip27`, `ip28`, `ip30`, `ip32_5k`, and `ip32_10k`. The appropriate selection for your platform can be determined by entering the following command:

```
hinv -c processor
```

The first line of output identifies the proper IP number. If a processor suffix (for example, `_4k`) is required, the next line identifies the processor (for example, R4000).

2.66.8 **`-TARG:processor=`*processor***

Selects the processor for which to schedule code. The chosen processor must support the instruction set architecture (ISA) that is specified (or implied by the ABI). Specify one of the following for *processor*: `r4000`, `r5000`, `r8000`, `r10000`, or `-r12000`.

2.66.9 `-TARG:r4krev22=`*setting*

Generates code to work around bugs in the R4000 rev 2.2 chip. This currently means simulating 64-bit variable shifts in the software. Specify `ON` or `OFF` for *setting*. The default is `OFF`.

2.66.10 `-TARG:sync=`*setting*

Enables or disables use of `SYNC` instructions. Specify `ON` or `OFF` for *setting*. The default is `ON`.

2.66.11 CPU Targeting (Cross Compiling) Using the `compiler.defaults` File

The MIPSpro 7 Fortran 90 compiler retrieves default information for the Application Binary Interface (ABI), instruction set architecture (ISA), and processor type, optimization, and IEEE arithmetic computations from `/etc/compiler.defaults`.

To compile for a different system, set the `COMPILER_DEFAULTS_PATH` environment variable to a path or to a colon-separated list of paths designating where the compiler is to look for the `compiler.defaults` file. For more information on this environment variable, see the `pe_environ`(5) man page.

The target `compiler.defaults` file must contain a `-DEFAULT:` *option* specifier that specifies the default information in the following format:

```
-DEFAULT:[abi=n32|64] [:isa=mips3|mips4]
[:proc=r4000|r5000|r8000|r10000|r12000] [:opt=0|1|2|3]
[:arith=1|2|3]
```

Note that command line settings override any settings in the system-supplied `compiler.defaults` file or in the `compiler.defaults` file that you create.

2.67 `-TENV:`…

Specifies the target environment option group. The *target environment* is the system upon which the executable code will be run. These options control the target environment assumed and/or produced by the compiler.

The following sections describe the `-TENV:`… options.

2.67.1 `-TENV:align_aggregate=`*bytes*

Controls alignment of allocated aggregates (that is, arrays and derived types). The value specified for *bytes* specifies that any aggregate object at least that large is to be given at least that alignment. By default, or if *bytes* is not specified, aggregates are aligned to the integer register size, which, for example, is 8 bytes for 64-bit programs and 4 bytes for 32-bit programs.

If `align_aggregate=0` is specified, the minimum alignment consistent with the ABI is used. Otherwise, the value specified must be 1, 2, 4, 8, or 16.

2.67.2 `-TENV:check_div=`*n*

Inserts checks for divide by zero operations and overflow conditions on integer divide operations. Specify 0, 1, 2, or 3 for *n*. The default is `check_div=1`.

`check_div=0` inhibits checking. `check_div=1` checks for division by zero. `check_div=2` checks for overflow. `check_div=3` checks for both division by zero and overflow.

Note: This option is deprecated. It will be removed in a future release. The preferred alternative is to specify `DEBUG:div_check`. For more information on the `DEBUG:div_check` option, see the `debug_group`(5) man page.

2.67.3 `-TENV:large_GOT=`*setting*

Generates code to accommodate a larger Global Offset Table (GOT) than is standard. Specify `ON` or `OFF` for *setting*. The default is `large_GOT=OFF`.

You can set this option to `ON` if you get a GOT or GP overflow message. For more information about the GOT, see the *What should I do about a GOT overflow?* question in the FAQ section of the `dso`(5) man page.

Note: If you specify both `-TENV:large_GOT=ON` and `-TENV:small_GOT=ON` on your command line, a message is issued and the `-TENV:small_GOT=ON` directive is recognized.

2.67.4 `-TENV:small_GOT=`*setting*

Assumes that the GOT for shared code is smaller than 64 KB, that is, assumessmall offsets for references to it. Specify `ON` or `OFF` for *setting*. The default is `small_GOT=ON`.

For more information on controlling the GOT, see the `-TENV:large_GOT` option.

2.67.5 `-TENV:X=`*n*

Specifies the level of enabled exceptions that will be assumed for purposes of performing speculative code motion; exceptions considered here are the floating point exceptions defined in the ANSI/IEEE 754-1985, the IEEE Standard for Binary Floating-point Arithmetic (inexact, overflow, underflow, divide-by-zero, and invalid operation) and memory traps (SIGSEGV and SIGBUS).

The default is `X=2` when `-O3` is in effect. The default is `X=1` when other `-O` optimization levels are in effect. Specify 0, 1, 2, 3, or 4 for *n*. The default is `X=1`.

Generally, an instruction is not speculated (moved above a branch by the optimizer) unless any exceptions it might cause are disabled by this option. `X=0` inhibits speculative code motion.

`X=1` specifies that safe speculative code motion be performed and disables all underflow and inexact exceptions according to ANSI/IEEE 754-1985.

`X=2` disables all exceptions described in ANSI/IEEE 754-1985, except divide by zero.

`X=3` disables all exceptions described in ANSI/IEEE 754-1985, including divide by zero.

`X=4` disables or ignores memory exceptions.

At levels higher than the `X=1` default level, various hardware exceptions, which are normally useful for debugging, or which are trapped and repaired by the hardware, may be disabled or ignored. This can hide obscure bugs. The program should not explicitly manipulate the IEEE floating-point trap-enable flags in the hardware if this option is used.

2.68 `-u`

Makes the default type of a variable undefined, rather than using default Fortran rules.

2.69 `-U`*var*

Undefines a variable for the source preprocessor. See the [`-D`*var*[=*def*][`,`*var*[=*def*]]...] option for information on defining variables.

For more information on source preprocessing compiler options, see the following options: `-cpp`, [`-D`*var*[=*def*][`,`*var*[=*def*]]...], `-E`, `-ftpp`, `-macro_expand`, `-nocpp`, and `-P`.

For information on source preprocessing and the macros available, see Chapter 7, page 185.

2.70 `-version`

Writes compiler release version information to `stdout`. No input file needs to be specified when this option is used.

2.71 `-w`[*arg*]

Specifies messages. This option can take one of the following forms:

Option	Action
`-w`	Suppresses warning messages.
`-w2`	Shows warning messages. Default.

2.72 `-Wl,`*opt*[`,`*arg*][`,`*opt*[`,`*arg*]]`...`

Specifies options to be passed directly to the loader. For *opt*, specify any of the options that the loader, `ld`(1), accepts. For *arg*, specify an argument, if necessary, to *opt*. For information on possible values for *opt* and *arg*, see the `ld`(1) man page.

Example. The following command line passes the loader options `-B static` and `-nostdlib` to `ld`(1):

```
f90 -Wl,-B,static,-nostdlib herfile.f
```

2.73 `-woff`*num*

Specifies message numbers to suppress. Examples:

- Specifying `-woff2026` suppresses message number 2026.
- Specifying `-woff2026-2352` suppresses messages 2026 through 2352.
- Specifying `-woff2026-2352,2400-2500` suppresses messages 2026 through 2353 and messages 2400 through 2500.

In the message level indicator, the message numbers appear after the dash in the message itself. For example, in the following message prefix, the message number is `197`:

```
f90-197 mfef90:...
```

You cannot suppress messages issued at the ERROR level.

2.74 `-x`*dirlist*

Disables specified directives or specified classes of directives. If specifying a multiword directive, either enclose the directive name in quotation marks or remove the spaces between the words in the directive's name.

For *dirlist*, enter one of the following:

dirlist	Directives disabled
`all` or `mipspro`	All directives.
`conditional_omp`	Directives prefixed with `!$`.
`dir`	Directives with a `!DIR` or `CDIR` prefix.
`mic`	Directives with a `!MIC` or `CMIC` prefix.

directive

One or more directives. If specifying more than one, separate them with commas, as follows:
`-x ORDERED,"ASSERT NOARGUMENTALIASING"`.

2.75 --

Separates options and file names. This option, which consists of two dashes, signifies the end of the options. After this symbol, you can specify the files to be processed.

2.76 *file.suffix*[90][*file.suffix*[90]...]

File or files to be processed, where *suffix* is either an uppercase `F` or a lowercase `f` for source files. Files ending in `.i`, `.o`, and `.s` are also accepted. The Fortran source files are compiled, and an executable object file is produced.

The default name of the executable object file is `a.out`. For example, the following command line produces `a.out`:

```
% f90 myprog.f
```

By default, several files are created during processing. The MIPSpro 7 Fortran 90 compiler can add a suffix to the *file* portion of the file name and write the files it creates to your working directory.

The following is a file summary:

File	Content
`a.out`	Executable output file.
file`.a`	Object file archive.
file`.B`	Intermediate file written by the front end of the compiler. To retain this file, specify the `-keep` option. For more information on the `-keep` option, see Section 2.36, page 19.
file`.cfb`	Feedback file for use with performance tools.
file`.f` or *file*`.F`	Input Fortran source file in fixed source form. If *file* ends in `.F`, the source preprocessor is

	invoked. For more information on preprocessing, see Chapter 7, page 185.
file`.f90` or *file*`.F90`	Input Fortran source file in free source form. If *file* ends in `.F90`, the source preprocessor is invoked. For more information on preprocessing, see Chapter 7, page 185.
file`.i`	File generated by the source preprocessor. To retain this file, specify the `-P` option. For more information on the `-P` option, see Section 2.58, page 55.
file`.l`	Assembler listing file. To retain this, specify the `-LIST` option. For more information on the `-LIST` option, see Section 2.41, page 22.
file`.list`	APO listing file. To retain this, specify the `-apolist` option. For more information on the `-apolist` option, see Section 2.4, page 9.
file`.L`	Listing file containing a cross reference and a source listing. To retain this file, specify the `-listing` option. For more information on the `-listing` option, see Section 2.42, page 23.
file`.mod`	Module file.
file`.o`	Object file. The implementation of alternate returns is not compatible between the MIPSpro Fortran 77 and the MIPSpro 7 Fortran 90 compilers. You cannot specify *file*`.o` files from both the MIPSpro Fortran 77 and the MIPSpro 7 Fortran 90 compilers, as input files, on the `f90`(1) command line if the files use alternate returns.
file`.s`	Assembly language file. To retain this file, specify the `-S` option. For information on the `-S` option, see Section 2.63, page 57.
file`.so`	Dynamic Shared Object (DSO) library.
`f90sigfpe.h`	Floating-point exception-handling include file.

General Directives [3]

A *directive* is a line inserted into Fortran source code that specifies actions to be performed by the compiler. Directive lines are not Fortran statements.

Many MIPSpro 7 Fortran 90 compiler features are implemented as either command line options or directives. The features implemented as command line options are set at compile time and applied to all files in the compilation. The features implemented through directives are set within your Fortran source code, and they apply to portions of your source code.

This chapter introduces the MIPSpro 7 Fortran 90 directive set and describes the general directives.

The sections in this chapter are as follows:

- Section 3.1, page 69, describes using directives.
- Section 3.2, page 72, describes the loop nest optimization (LNO) directives.
- Section 3.3, page 82, describes the argument aliasing directives.
- Section 3.4, page 83, describes the symbol storage directives.
- Section 3.5, page 86, describes the inlining and IPA directives.

3.1 Using Directives

All directives are of the following form:

prefix directive

prefix Each directive begins with a prefix. The prefix needed for each directive is shown in the directive's description. The following directive prefixes are used by the MIPSpro 7 Fortran 90 compiler:

- `!*$*` and `C*$*`. These prefixes are used by the loop-nest directives described in this chapter.
- `!$OMP` and `C$OMP`. These prefixes are used by the OpenMP Fortran API multiprocessing directives described in Chapter 4, page 89.
- `!$SGI` and `C$SGI`. These prefixes are used by multiprocessing directives that are Silicon Graphics extensions to the OpenMP Fortran API described in Chapter 5, page 143.
- `!DIR$` and `CDIR$`. These prefixes are used by the Autotasking directives described in Chapter 6, page 171.
- `!$PAR` and `C$PAR`. These prefixes are used by the PCF multitasking directives described in Appendix D, page 267.
- `!$` and `C$`. These prefixes are used by the outmoded multitasking directives described in Appendix D, page 267, and Chapter 5, page 143.
- `!MIC$` and `CMIC$`. These prefixes are used by the Autotasking directives described in Appendix C, page 253.

The prefix used also depends on which Fortran source form you are using, as follows:

- If you are using fixed source form, begin a directive line with the characters `C`*prefix* or `!`*prefix*. The `!` or `C` character must appear in column 1. Beginning the directive with a `!` or `C` character ensures that compilers other than the MIPSpro 7 Fortran 90 compiler will treat compiler directive lines as comment lines.
- If you are using free source form, begin a directive line with the characters `!`*prefix*, followed by a space, and then one or more directives. The `!`*prefix* need not start in column 1, but it must be the first text on a line.

Because both fixed source form and free source form accept directives that start with the exclamation point (`!`), that is the initial character used in all directive syntax descriptions in this manual.

directive	This is the specific directive's syntax. The syntax usually consists of the directive name. Some directives accept arguments. A directive's arguments, if any, are shown in the description for the directive itself.

The following sections describe the general format for directives and explain how directives are continued across source code lines.

Note: The multiprocessing directives supported in previous MIPSpro 7 Fortran 90 releases are outmoded, and so are the `!$PAR`, `C$PAR`, `!$`, and `C$` directive prefixes. This technology is outmoded, but it is still supported for older codes that require this functionality. Silicon Graphics and Cray Research encourage you to modify your code using the OpenMP directives described in Chapter 4, page 89.

3.1.1 Directives and Command Line Options

Some compiler features can be activated on the command line and through compiler directives. The difference is that a command line setting applies to all files in the compilation, but a directive applies to only a program unit or to another specific part of a source file.

Generally, and by default, directives override command line options. There are exceptions to this rule, however. The exceptions, if any, are noted in the introductory text to each directive group.

3.1.2 Directive Range

The range of a particular directive depends on the directive itself, as follows:

- If a directive appears within a program unit, it applies only to that program unit. Within a program unit, many directives apply only to the loops that they immediately precede.
- If a directive appears outside a program unit (for example, prior to program code in a file) it applies to the entire file.

The descriptions for the individual directives indicate the range of the directive.

3.1.3 Directive Continuation and Other Considerations

It is sometimes necessary to continue a directive across one or more source code lines. The continuation character used and its placement within the directive line depends on the type of directive you are using. The introductory text for each directive group indicates the continuation character that is appropriate for that group.

For all directives in this chapter, the prefix for a directive line that is a continuation line is `!*$*&`.

Do not use source preprocessor (#) directives within multiline compiler directives.

3.2 LNO Directives

The loop nest optimization (LNO) directives control loop nest optimizations. By default, directives override command line options. To reverse this, and have command line options override the LNO directives, specify `-LNO:ignore_pragmas`. For information on the `-LNO:ignore_pragmas` option, see Section 2.43.1.3, page 25.

To continue a directive, the continuation line must begin with `!*$*&`.

The following directives control loop nest optimizations:

- `AGGRESSIVEINNERLOOPFISSION`
- `BLOCKABLE`
- `BLOCKINGSIZE`, `NOBLOCKING`
- `FISSION`, `FISSIONABLE`, `NOFISSION`
- `FUSE`, `FUSEABLE`, `NOFUSION`
- `INTERCHANGE`, `NOINTERCHANGE`
- `PREFETCH`
- `PREFETCH_MANUAL`
- `PREFETCH_REF`
- `PREFETCH_REF_DISABLE`
- `UNROLL`

The following sections describe the LNO directives.

3.2.1 Request Loop Fission for Inner Loops: `AGGRESSIVEINNERLOOPFISSION` Directive

The `AGGRESSIVEINNERLOOPFISSION` directive specifies that the following loop should be split into as many loops as possible. In a loop nest, this directive must precede an inner loop.

The format of this directive is as follows:

```
!*$* AGGRESSIVEINNERLOOPFISSION
```

3.2.2 Permit Cache Blocking: `BLOCKABLE` Directive

The `BLOCKABLE` directive specifies that it is legal to cache block the subsequent loops. For more information on controlling cache blocking, see the `-LNO:blocking` option in Section 2.43.2.1, page 28, and the `-LNO:blocking_size` option in Section 2.43.2.2, page 28.

The format of this directive is as follows:

```
!*$* BLOCKABLE (do_variable,do_variable [,do_variable]...)
```

do_variable	Specify the *do_variable* names of two or more loops. The loops identified by the *do_variable* names must be adjacent and nested within each other, although they need not be perfectly nested.

This directive informs the compiler that these loops can be involved in a blocking situation with each other, even if the compiler would consider such a transformation illegal. The loops must also be interchangeable and unrollable. This directive does not instruct the compiler on which of these transformations to apply.

3.2.3 Declare Cache Blocking: `BLOCKINGSIZE` and `NOBLOCKING` Directives

The `BLOCKINGSIZE` and `NOBLOCKING` directives assert that the loop following the directive either is (or is not) involved in a cache blocking for the primary or secondary cache.

The formats of these directives are as follows:

```
!*$* BLOCKINGSIZE(n1[,n2])

!*$* NOBLOCKING
```

n1,*n2* An integer number that indicates the block size. If the loop is involved in a blocking, it will have a block size of *n1* for the primary cache and *n2* for the secondary cache. The compiler attempts to include this loop within such a block, but it cannot guarantee this.

If *n1* or *n2* are 0, the loop is not blocked, but the entire loop is inside the block.

Example:

```
      SUBROUTINE AMAT(X,Y,Z,N,M,MM)
      REAL(KIND=8) X(100,100), Y(100,100), Z(100,100)
      DO K = 1, N
!*$* BLOCKING SIZE (20)
         DO J = 1, M
!*$* BLOCKING SIZE (20)
            DO I = 1, MM
               Z(I,K) = Z(I,K) + X(I,J)*Y(J,K)
            END DO
         END DO
      END DO
      END
```

For the preceding code, the compiler makes 20 X 20 blocks when blocking, but it could block the loop nest such that loop K is not included in the tile. If it did not, add a `BLOCKINGSIZE(0)` directive just before loop K to specify that the compiler should generate a loop such as the following:

```
      SUBROUTINE AMAT(X,Y,Z,N,M,MM)
      REAL(KIND=8) X(100,100), Y(100,100), Z(100,100)
      DO JJ = 1, M, 20
         DO II = 1, MM, 20
            DO K = 1, N
               DO J = JJ, MIN(M, JJ+19)
                  DO I = II, MIN(MM, II+19)
                     Z(I,K) = Z(I,K) + X(I,J)*Y(J,K)
                  END DO
               END DO
            END DO
         END DO
      END DO
      END
```

Note that an `INTERCHANGE` directive can be applied to the same loop nest as a `BLOCKINGSIZE` directive. The `BLOCKINGSIZE` directive applies to the loop it directly precedes; it moves with that loop when an interchange is applied.

The `NOBLOCKING` directive prevents the compiler from involving the subsequent loop in a cache blocking situation.

3.2.4 Control Loop Fission for Outer Loops: `FISSION`, `FISSIONABLE`, and `NOFISSION` Directives

The fission control directives specify whether the compiler should perform loop fission on the loops that immediately follow these directives.

The formats of these directives are as follows:

```
!*$* FISSION[(level)]

!*$* FISSIONABLE

!*$* NOFISSION
```

level	Specify an integer number that indicates the number of loop levels that should undergo loop fission.

The `FISSION` directive specifies that loop fission should be attempted. The compiler performs a validity test on the subsequent loops unless you have also specified a `FISSIONABLE` directive. The `NOFISSION` directive specifies that the following loop should not undergo fission, but its inner loops, if any, may undergo fission.

These directives do not cause statements to be reordered.

3.2.5 Control Loop Fusion for Outer Loops: `FUSE`, `FUSEABLE`, and `NOFUSION` Directives

The fusion control directives specify whether the compiler should perform loop fusion on the loops that immediately follow these directives.

The formats of these directives are as follows:

```
!*$* FUSE[(n,[level])]

!*$* FUSEABLE

!*$* NOFUSION
```

n	Specify an integer number that indicates the number of subsequent loops that should undergo loop fusion. The default is 2.
level	Specify an integer that indicates how deeply the loops should be fused. The level of loop fusion is determined by the maximum perfectly nested loop levels of the fused loops, although partial fusion is allowed.

Loop iterations may be peeled as needed during loop fusion. The limit of this peeling is 5, or the number specified by the `-LNO:fusion_peeling_limit` command line option.

The `FUSE` directive specifies that loop fusion should be attempted. The compiler performs a validity test on the subsequent loops unless you have also specified a `FUSEABLE` directive. When the `FUSEABLE` directive is specified, the fusion is done for loops with identical iteration counts. The `NOFUSION` directive specifies that the following loop should not be fused with any other loop. For more information on the `-LNO:fusion_peeling_limit` command line option, see Section 2.43.2.5, page 30.

Example. Consider the following code:

```
DO I = 1,N
  DO J = 1,N
    ...
  END DO
END DO
DO I = 1,N
  DO J = 1,N
    ...
  END DO
END DO
```

Fusing the loops with a *level* of 1 results in the following loop nest:

```
DO I = 1,N
  DO J = 1,N
    ...
  END DO
  DO J = 1,N
    ...
  END DO
END DO
```

Fusing the loops with a *level* of 2 results in the following loop nest:

```
DO I = 1,N
  DO J = 1,N
    ...
    ...
  END DO
END DO
```

3.2.6 Control Loop Interchange: `INTERCHANGE` and `NOINTERCHANGE` Directives

The loop interchange control directives specify whether or not the order of the following two or more loops should be interchanged. These directives apply to the loops that they immediately precede.

The formats of these directives are as follows:

```
!*$* INTERCHANGE (do_variable1,do_variable2 [,do_variable3]...)

!*$* NOINTERCHANGE
```

do_variable	Specifies two or more *do_variable* names. The *do_variable* names can be specified in any order, and the compiler reorders the loops. The loops must be perfectly nested. If the loops are not perfectly nested, you may receive unexpected results.

The compiler reorders the loops such that the loop with *do_variable1* is outermost, then loop *do_variable2*, then loop *do_variable3*.

The `NOINTERCHANGE` directive inhibits loop interchange on the loop that immediately follows the directive.

3.2.7 Control Prefetching for a Program Unit: `PREFETCH` Directive

The `PREFETCH` directive controls the MIPS IV prefetch instruction. Using this directive can increase performance in program units that are likely to encounter cache misses during execution. This directive applies only to the program unit in which it appears.

When the directive is specified, the compiler estimates the memory references that will be cache misses, inserts prefetches for the misses, and schedules the prefetches ahead of their corresponding references. You can specify different levels of prefetching aggressiveness for the primary and secondary cache.

The format of this directive is as follows:

```
!*$* PREFETCH (primary_cache [,secondary_cache])
```

primary_cache, *secondary_cache*	For each of these, specify 0, 1, or 2. The number specified indicates the level of prefetching requested for the primary and secondary cache levels, respectively. A 0 disables all prefetching. 1 requests conservative prefetching. 2 requests aggressive prefetching. By default, *primary_cache* and *secondary_cache* are both set to 1 when the `-r10000` command line option is in effect, and they are set to 0 for all other processor settings.

This directive is recognized only if the `-mips4` and `-r10000` command line options are in effect.

3.2.8 Control Prefetching in a Subprogram: `PREFETCH_MANUAL` Directive

The `PREFETCH_MANUAL` directive specifies whether the `PREFETCH_REF` and the `PREFETCH_REF_DISABLE` directives, which perform manual prefetches, should be respected or ignored within a subprogram. This directive applies only to the program unit in which it appears.

The format of this directive is as follows:

```
!*$* PREFETCH_MANUAL (n)
```

n	Specify either 0 or 1 for *n*. 0 indicates that the compiler should ignore all prefetch directive. 1 indicates that all prefetch directives should be recognized. By default, all prefetch directives are recognized.

This directive is recognized only if the `-mips4` and `-r10000` command line options are in effect. For more information on the `-mips4` option, see Section 2.46, page 37. For more information on the `-r10000` option, see Section 2.61, page 56.

3.2.9 Request Prefetching for an Array: `PREFETCH_REF` Directive

The `PREFETCH_REF` directive requests prefetching for a specific memory reference. This directive applies only to the loop nest that includes references to *array*, and the directive must immediately precede the loop nest.

When this directive is specified, all references to *array* in the subsequent loop nest are ignored by the automatic prefetcher (if enabled).

The format of this directive is as follows:

```
!*$* PREFETCH_REF=array [,stride=stride[,stride]] [,level=level[,level]]
  [,kind=rw] [,size=size]
```

array	For *array*, specify identification information for the array. For example: `A(I,J)`.
stride	Specify prefetching for every *stride* iterations of the loop. The default is 1.
level	Specify the level in the memory hierarchy to prefetch, either 1 or 2. The default is 2. 1 specifies a prefetch from secondary cache to primary cache. 2 specifies a prefetch from memory to primary cache.
rw	Specify `rd` or `wr`. `rd` indicates that the location is read. `wr` indicates that the location is written. The default is `wr`.
size	Specify the size, in KB, of *array*. Must be a constant. If *size* is specified, the automatic prefetcher (if enabled) reduces the effective cache size by that amount in its calculations. The

compiler tries to issue one prefetch per *stride* iterations, but this cannot be guaranteed.

This directive generates a single prefetch instruction to a specified memory reference. It searches for array references that match the supplied reference in the current loop nest and takes the following actions:

- If the reference is found, the reference is scheduled relative to the prefetch node, based on the miss latency for the specified level of the cache.
- If no such reference is found, the prefetch is generated at the start of the loop body.

This directive is recognized only if the `-mips4` and `-r10000` command line options are in effect. For more information on the `-mips4` option, see Section 2.46, page 37. For more information on the `-r10000` option, see Section 2.61, page 56

3.2.10 Disable Prefetching for a Specific Array: `PREFETCH_REF_DISABLE` Directive

The `PREFETCH_REF_DISABLE` directive disables prefetching for all references to an array. This directive applies to all array references within the program unit.

The format of this directive is as follows:

```
!*$* PREFETCH_REF_DISABLE=array [, size=size]
```

array For *array*, specify identification information for the array. For example: `A(I,J)`.

If the automatic prefetcher is enabled, it ignores *array*.

size Specifies the size, in Kbytes, of *array*. Must be a constant.

The *size* is used for *volume analysis*. Volume analysis is performed as part of prefetching analysis. In volume analysis, the compiler tries to determine the amount of data referenced by each loop or loop nest. This information is used when determining whether or not to prefetch memory references.

This directive is recognized only if the `-mips4` and `-r10000` command line options are in effect.

3.2.11 Request Loop Unrolling: `UNROLL` Directive

The `UNROLL` directive specifies loop unrolling. This directive applies to the loop that immediately follows the directive.

Inner loop unrolling occurs automatically when `-O2` or `-O3` are in effect. Non-inner loop unrolling (and jam) occurs when `-O3` is in effect.

The format of this directive is as follows:

```
!*$* UNROLL (n)
```

n	Specifies the number of copies of the loop body to be generated, as follows: • When this directive precedes an inner loop, the compiler generates *n* – 1 copies of the loop body. This is standard loop unrolling. • When this directive precedes an outer loop, the compiler performs an *unroll and jam* operation on the loop. The value of *n* must be at least 2 in order for unrolling to occur. If *n* = 1, no unrolling is performed.

Even with this directive specified, unrolling is not performed if the compiler determines that unrolling would be unsafe. To specify that the compiler unroll the loop regardless of its analysis, you must also specify a `BLOCKABLE` directive. For information on the `BLOCKABLE` directive, see Section 3.2.2, page 73.

Example. Assume that `-O3` is specified and that the outer loop of the following nest will be unrolled by two:

```
!*$* UNROLL (2)
      DO I = 1, 10
        DO J = 1,100
              A(J,I) = B(J,I) + 1
        END DO
      END DO
```

With outer loop unrolling, the compiler produces the following nest, in which the two bodies of the inner loop are adjacent to each other:

```
      DO I = 1, 10, 2
        DO J = 1,100
```

```
            A(J,I) = B(J,I) + 1
      END DO
      DO J = 1,100
            A(J,I+1) = B(J,I+1) + 1
      END DO
    END DO
```

The compiler then *jams*, or *fuses*, the inner two loop bodies together, producing the following nest:

```
    DO I = 1, 10, 2
      DO J = 1,100
            A(J,I)   = B(J,I) + 1
            A(J,I+1) = B(J,I+1) + 1
      END DO
    END DO
```

3.3 Argument Aliasing Directives (ASSERT ARGUMENTALIASING and ASSERT NOARGUMENTALIASING)

The ASSERT ARGUMENTALIASING and ASSERT NOARGUMENTALIASING directives allow the compiler to make assumptions about procedure dummy arguments when performing optimizations.

It is possible to call a procedure and specify the same variable or array element in two or more positions of the actual argument list. Within the procedure, two or more dummy argument names, which appear to refer to different memory locations, actually refer to the same location. This practice violates the Fortran standard. You can use the ASSERT ARGUMENTALIASING directive to force the compiler to be more conservative.

By default, ASSERT NOARGUMENTALIASING is in effect.

The formats for these directives are as follows:

```
!*$* ASSERT ARGUMENTALIASING

!*$* ASSERT NOARGUMENTALIASING
```

If these directives appear prior to Fortran source code in a file, they are applied to all program units in the file. If they appear in a program unit, they are applied to that program unit only. If one of these directives is encountered, it remains in effect until reset by the opposing directive.

3.4 Symbol Storage Directives

The following directives control symbol storage:

- `ALIGN_SYMBOL`
- `FILL_SYMBOL`
- `FLUSH`
- `SECTION_GP`
- `SECTION_NON_GP`

3.4.1 Control Symbol Alignment and Padding: `ALIGN_SYMBOL` and `FILL_SYMBOL` Directives

The `ALIGN_SYMBOL` and `FILL_SYMBOL` directives control the way symbols are stored.

The `ALIGN_SYMBOL` directive aligns the start of *symbol* at a specified alignment boundary.

The `FILL_SYMBOL` directive pads *symbol* with additional storage so that the symbol is assured not to overlap (even partially) with any other data item within the storage of the specified size. The additional padding required is divided between each end of the specified variable. For example, a `FILL_SYMBOL(X,L1CACHELINE)` directive guarantees that `X` does not suffer from false sharing for the primary cache line.

The formats for these directives are as follows:

```
!*$* ALIGN_SYMBOL (symbol [, storage])

!*$* FILL_SYMBOL (symbol [, storage])
```

symbol	Specify the name of a symbol. *symbol* can be a common block variable or a module name. *symbol* cannot be a component of a derived type, an array element, a common block, or blank common.

storage Specify the storage size. Specify one of the following values for *storage*:

storage	Action
`L1CACHELINE`	Specifies the machine-specific first-level cache line size, typically 32 bytes.
`L2CACHELINE`	Specifies the machine-specific secondary cache line size, typically 128 bytes.
`PAGE`	Specifies a machine-specific page. Typically 16 KB.
power-of-two	An integer value that is a power of 2. This is measured in bytes.

For common block variables, these directives are required at each declaration of the common block. Because the directives modify the allocated storage and its alignment for the named *symbol*, inconsistent directives can lead to undefined results.

The `ALIGN_SYMBOL` directive has no effect on fixed-size local symbols, such as simple scalars or arrays of known size (for example symbols declared as `REAL(N)` or `REAL(A(3))`). The directive continues to be effective for automatic arrays (stack-allocated arrays of dynamically determined size).

You cannot specify an `ALIGN_SYMBOL` directive and a `FILL_SYMBOL` directive for the same *symbol*.

Example:

```
! X IS A COMMON BLOCK VARIABLE
      COMMON X!
      INTEGER(KIND=4) X
!*$* ALIGN_SYMBOL (X, 32)

!   X WILL START AT A 32-BYTE BOUNDARY.
!   WARNING: THE LAYOUT OF THE COMMON BLOCK WILL BE AFFECTED

!*$* ALIGN_SYMBOL (X, 2)
!   ERROR: CANNOT REQUEST AN ALIGNMENT LOWER THAN THE NATURAL
!   ALIGNMENT OF THE SYMBOL.
```

```
      REAL(KIND=8) Y
!   Y IS A COMMON BLOCK OR LOCAL VARIABLE
!*$* FILL_SYMBOL (Y, L2CACHELINE)


!   ALLOCATE EXTRA STORAGE BOTH BEFORE AND AFTER Y SO THAT
!   Y IS WITHIN AN L2CACHELINE (128 BYTES) ALL BY ITSELF.
!   THIS CAN BE USEFUL TO AVOID FALSE-SHARING BETWEEN MULTIPLE
!   PROCESSORS FOR THE CACHE LINE CONTAINING Y.
```

3.4.2 Declare a Synchronization Point: FLUSH Directive

The FLUSH directive identifies synchronization points at which thread-visible variables are written back to memory. This directive must appear at the precise point in the code at which the synchronization is required.

Note: This directive has the same effect as the FLUSH directive described in the OpenMP Fortran API. For more information on the OpenMP FLUSH directive, see Section 4.6.5, page 109.

Thread-visible variables include the following data items:

- Globally visible variables (common blocks and modules).
- Local variables that do not have the SAVE attribute but have had their address taken and saved or have had their address passed to another subprogram.
- Local variables that do not have the SAVE attribute that are declared shared in a parallel region within the subprogram.
- Dummy arguments.
- All pointer dereferences.

This directive has the following format:

```
!*$* FLUSH [(var[, var] ...)]
```

var Variables to be flushed.

3.4.3 Specify Global Pointer Use: `SECTION_GP` and `SECTION_NON_GP` Directives

The MIPSpro 7 Fortran 90 compiler can reference global data by using the global pointer and an offset value. Using the global pointer (gp) is more efficient than constructing the address at each occurence, but because the offset size is limited to 16 bits, only a limited set of elements can be referenced using the global pointer.

The compiler places global data in gp-relative or non-gp-relative sections, but you can use the `SECTION_GP` and `SECTION_NON_GP` directives to specify the variables to go within the gp-relative section and the variables that need to be addressed explicitly.

The formats for these directives are as follows:

```
!*$* SECTION_GP (symbol [, symbol] ...)

!*$* SECTION_NON_GP (symbol [, symbol] ...)
```

symbol Enter one or more symbols. Separate multiple symbols with commas. Valid symbols are common block names, variables specified on `SAVE` statements, and module names. If a module name is specified, all storage in the module is affected. If a common block name is specified, it must be of the following form: `/`*name*`/`.

3.5 Inlining and IPA Directives (`INLINE`, `NOINLINE`, `IPA`, and `NOIPA`)

The following are the inlining and interprocedural analysis (IPA) directives:

- `INLINE`, `NOINLINE`
- `IPA`, `NOIPA`

Note: Neither inlining nor IPA are enabled by default. By default, the directives in this section, if present in your source code, are ignored. To enable the directives and turn on inlining and IPA, specify the `-INLINE:` option or the `-IPA:` option on your `f90`(1) command line. For more information on the command line interaction with these features, see Chapter 2, page 5, or see one of the following man pages: `f90`(1) or `ipa`(5).

Inlining is the process of replacing a procedure reference with a copy of the procedure's code. This eliminates procedure call overhead and exposes the relationships between the procedure code, the return value, and the surrounding code. The `INLINE` and `NOINLINE` directives allow you to specify procedures that should be inlined.

Interprocedural analysis (IPA) is a MIPSpro compiler feature that includes inlining, common block array padding, constant propagation, dead procedure elimination, dead variable elimination, and global name optimizations. For detailed information on the IPA feature, see the `ipa`(5) man page. The `IPA` and `NOIPA` directives allow you to control IPA.

The formats of these directives are as follows:

```
!*$* INLINE location [(name [,name] ...)]

!*$* NOINLINE location [(name [,name] ...)]
```

```
!*$* IPA location [(name [,name] ...)]

!*$* NOIPA location [(name [,name] ...)]
```

location Specify one of the following for *location*:

location	Action
`HERE`	Specifies that routines named on the subsequent source code line should be inlined or should undergo IPA. Default.
`ROUTINE`	Specifies that the named function should be inlined or should undergo IPA everywhere it appears within the current routine.
`GLOBAL`	Specifies that the named function should be inlined or should undergo IPA throughout the source file.

name	For the inlining directives, each *name* specification represents one or more routines to be inlined. If no routines are named, all routines in the program are inlined. For the IPA directives, each *name* specification represents one or more routines to undergo IPA. If no routines are named, all routines in the program undergo IPA.

Example. Consider the following code fragment:

```
      DO I = 1,N
!*$* INLINE (BETA) HERE
         CALL BETA(I,1)
      ENDDO
      CALL BETA(N,2)
```

Using the specifier `ROUTINE` rather than `HERE` in this example would inline both calls to `BETA`. Note that `-INLINE:=ON` must be specified on the `f90`(1) command line when this code is compiled in order for the inlining directive to be recognized.

OpenMP Fortran API Multiprocessing Directives [4]

This chapter describes the multiprocessing directives that the MIPSpro 7 Fortran 90 compiler supports. These directives are based on the OpenMP Fortran application program interface (API) standard. Programs that use these directives are portable and can be compiled by other compilers that support the OpenMP standard.

To enable recognition of the OpenMP directives, specify `-mp` on the `f90`(1) command line. The `-mp` option must be specified in order for the compiler to honor any `-MP:...` options that may also be specified on the command line. The `-MP:open_mp=ON` option is on by default and must be in effect during compilation.

The following example command line can compile program `ompprg.f`, which contains OpenMP Fortran API directives:

```
f90 -mp ompprg.f
```

In addition to directives, the OpenMP Fortran API describes several library routines and environment variables. Information on these other utilities can be found in the following locations:

Programming Utility	Information Location
Command line information	For information on the `-mp` option, see Section 2.47, page 38. For information on the `-MP:` option, see Section 2.48.5, page 42.
Library routines	`omp_lock`(3), `omp_nested`(3), and `omp_threads`(3) man pages

Environment variables	`pe_environ`(5) man page

Note: If individual loops in your program contain both OpenMP directives and extensions (prefixed with `!$OMP` or `!$SGI`) **and** any of the outmoded multiprocessing directives described in Appendix D, page 267, and Chapter 5, page 143, (prefixed with `!$` or `!$PAR`), you must specify the set of directives that the compiler should use. To direct the compiler to ignore the OpenMP directives, compile with `-MP:open_mp=OFF`. To direct the compiler to ignore the outmoded multiprocessing directives, compile with `-MP:old_mp=OFF`. To direct the compiler to ignore the outmoded Origin series distributed shared memory directives, specify `-MP:dsm=OFF`. For more information on the `-mp` option, see Section 2.47, page 38. For more information on the `-MP:` option, see Section 2.48.5, page 42.

The sections in this chapter are as follows:

- Section 4.1, page 91, describes using directives and the directive format.
- Section 4.2, page 92, describes conditional compilation.
- Section 4.3, page 94, describes the parallel region construct.
- Section 4.4, page 96, describes work-sharing constructs.
- Section 4.5, page 102, describes the combined parallel work-sharing constructs.
- Section 4.6, page 105, describes synchronization constructs.
- Section 4.7, page 111, describes the data environment, which includes directives and clauses that affect the data environment.
- Section 4.8, page 121, describes directive binding.
- Section 4.9, page 123, describes directive nesting.
- Section 4.10, page 126, Section 4.11, page 135, and Section 4.12, page 137, describe optimization.

Note: The Silicon Graphics multiprocessing directives, including the Origin series distributed shared memory directives, are outmoded. Their preferred alternatives are the OpenMP Fortran API directives described in this chapter.

4.1 Using Directives

All multiprocessing directives are case-insensitive and are of the following form:

prefix directive [*clause*[[`,`] *clause*]`...`]

prefix	Each directive begins with a prefix, and the prefixes you can use depend on your source form, as follows: • If you are using fixed source form, the following prefixes can be used: `!$OMP`, `C$OMP`, or `*$OMP`. Prefixes must start in column one and appear as a single word with no intervening white space. Fortran fixed form line length, case sensitivity, white space, continuation, and column rules apply to the directive line. • If you are using free source form, the following prefix can be used: `!$OMP`. A prefix can appear in any column as long as it is preceded only by white space. It must appear as a single word with no intervening white space. Fortran free form line length, case sensitivity, white space, and continuation rules apply to the directive line.
directive	The name of the directive.
clause	One or more directive clauses. Clauses can appear in any order after the directive name and can be repeated as needed, subject to the restrictions listed in the description of each clause.

Directives cannot be embedded within continued statements, and statements cannot be embedded within directives. Comments cannot appear on the same line as a directive.

In fixed source form, initial directive lines must have a space or zero in column six, and continuation directive lines must have a character other than a space or a zero in column six.

In free source form, initial directive lines must have a space after the prefix. Continued directive lines must have an ampersand as the last nonblank character on the line. Continuation directive lines can have an ampersand after the directive prefix with optional white space before and after the ampersand.

Example 1 (fixed source form). The following formats for specifying directives are equivalent (the first line represents the position of the first 9 columns):

```
C23456789
!$OMP PARALLEL DO SHARED(A,B,C)

C$OMP PARALLEL DO
C$OMP+SHARED(A,B,C)

C$OMP PARALLELDOSHARED(A,B,C)
```

Example 2 (free source form). The following formats for specifying directives are equivalent (the first line represents the position of the first 9 columns):

```
!23456789
       !$OMP PARALLEL DO &
                 !$OMP SHARED(A,B,C)

!$OMP PARALLEL &
       !$OMP&DO SHARED(A,B,C)

      !$OMP PARALLEL DO SHARED(A,B,C)
```

Note: In order to simplify the presentation, the remainder of this chapter uses the `!$OMP` prefix in all syntax descriptions and examples.

4.2 Conditional Compilation

Fortran statements can be compiled conditionally as long as they are preceded by one of the following conditional compilation prefixes: `!$`, `C$`, or `*$`. The prefix must be followed by a Fortran statement on the same line. During compilation, the prefix is replaced by two spaces, and the rest of the line is treated as a normal Fortran statement.

Your program must be compiled with the `-mp` option in order for the compiler to honor statements preceded by conditional compilation prefixes; without the `mp` command line option, statements preceded by conditional compilation prefixes are treated as comments. For more information on the `-mp` option, see Section 2.47, page 38.

The `!$` prefix is accepted when compiling either fixed source form files or free source form files. The `C$` and `*$` prefixes are accepted only when compiling fixed source form. The source form you are using also dictates the following:

- In fixed source form, the prefixes must start in column one and appear as a single word with no intervening white space. Fortran fixed form line length, case sensitivity, white space, continuation, and column rules apply to the line. Initial lines must have a space or zero in column six, and continuation lines must have a character other than a space or zero in column six.

  Example. The following forms for specifying conditional compilation are equivalent:

  ```
  C23456789
  !$ 10 IAM = OMP_GET_THREAD_NUM() +
  !$   &          INDEX

  #ifdef _OPENMP
     10 IAM = OMP_GET_THREAD_NUM() +
       &          INDEX
  #endif
  ```

- In free source form, the !$ prefix can appear in any column as long as it is preceded only by white space. It must appear as a single word with no intervening white space. Fortran free source form line length, case sensitivity, white space, and continuation rules apply to the line. Initial lines must have a space after the prefix. Continued lines must have an ampersand as the last nonblank character on the line. Continuation lines can have an ampersand after the prefix, with optional white space before and after the ampersand.

In addition to the conditional compilation prefixes, a preprocessor macro, _OPENMP, can be used for conditional compilation. For more information on source preprocessing and conditional compilation, see Chapter 7, page 185.

Example. The following example illustrates the use of the conditional compilation prefix. Assuming Fortran fixed source form, the following statement is invalid when using OpenMP constructs:

```
C234567890
!$  X(I) = X(I) + XLOCAL
```

With OpenMP compilation, the conditional compilation prefix !$ is treated as two spaces. As a result, the statement infringes on the statement label field. To be valid, the statement should begin after column six, like any other fixed source form statement:

```
C234567890
!$    X(I) = X(I) + XLOCAL
```

In other words, conditionally compiled statements need to meet all applicable language rules when the prefix is replaced with two spaces.

4.3 Parallel Region Constructs (`PARALLEL` and `END PARALLEL` Directives)

The `PARALLEL` and `END PARALLEL` directives define a *parallel region*. A parallel region is a block of code that is to be executed by multiple threads in parallel. This is the fundamental OpenMP parallel construct that starts parallel execution. These directives have the following format:

```
!$OMP PARALLEL [clause[[,] clause]...]

block

!$OMP END PARALLEL
```

clause — *clause* can be one or more of the following:

- `PRIVATE(`*var*`[, `*var*`] ...)`
- `SHARED(`*var*`[, `*var*`] ...)`
- `DEFAULT(PRIVATE | SHARED | NONE)`
- `FIRSTPRIVATE(`*var*`[, `*var*`] ...)`
- `REDUCTION ({`*operator*`|`*intrinsic*`}:`*var*`[, `*var*`] ...)`
- `IF(`*scalar_logical_expression*`)`
- `COPYIN(`*var*`[, `*var*`] ...)`

The `IF` clause is described in this section. For information on the `PRIVATE`, `SHARED`, `DEFAULT`, `FIRSTPRIVATE`, `REDUCTION`, and `COPYIN` clauses, see Section 4.7.2, page 113.

block — *block* denotes a structured block of Fortran statements. You cannot branch into or out of the block. The code contained within the dynamic extent of the parallel region is executed on each thread.

The `END PARALLEL` directive denotes the end of the parallel region. There is an implied barrier at this point. Only the master thread of the team continues execution past the end of a parallel region.

When a thread encounters a parallel region, it creates a team of threads, and it becomes the master of the team. The master thread is a member of the team and it has a thread number of 0 within the team. The number of threads in the team is controlled by environment variables and/or library calls.

The number of physical processors actually hosting the threads at any given time depends on the number of CPUs available and the system load. Once created, the number of threads in the team remains constant for the duration of that parallel region, but it can be changed either explicitly by the user or automatically by the run-time system from one parallel region to another. The OMP_SET_DYNAMIC(3) library routine and the OMP_DYNAMIC environment variable can be used to enable and disable the automatic adjustment of the number of threads. For more information on environment variables that affect OpenMP directives, see the pe_environ(5) man page.

OpenMP: The OpenMP Fortran API does not specify the number of physical processors that can host the threads at any given time.

If a thread in a team executing a parallel region encounters another parallel region, it creates a new team, and it becomes the master of that new team. By default, nested parallel regions are serialized; that is, they are executed by a team composed of one thread. This default behavior can be changed by using either the OMP_SET_NESTED(3) library routine or the OMP_NESTED environment variable. For more information on environment variables that affect OpenMP directives, see the pe_environ(5) man page.

If an IF clause is present, the enclosed code region is executed in parallel only if the *scalar_logical_expression* evaluates to .TRUE.. Otherwise, the parallel region is serialized. The expression must be a scalar Fortran logical expression.

The following restrictions apply to parallel regions:

- The PARALLEL/END PARALLEL directive pair must appear in the same routine in the executable section of the code.
- The code contained by these two directives must be a structured block. You cannot branch into or out of a parallel region.
- Only a single IF clause can appear on the directive.

Example. The PARALLEL directive can be used for exploiting coarse-grained parallelism. In the following example, each thread in the parallel region decides what part of the global array X to work on based on the thread number:

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,NPOINTS)
      IAM = OMP_GET_THREAD_NUM()
      NP =  OMP_GET_NUM_THREADS()
      IPOINTS = NPOINTS/NP
      CALL SUBDOMAIN(X,IAM,IPOINTS)
!$OMP END PARALLEL
```

Note: `ALLOCATABLE` or `POINTER` arrays can be privatized by using a `PRIVATE` clause on a `PARALLEL` directive or by using a worksharing construct (a `DO`, `SINGLE`, or `SECTIONS` directive).

However, `ALLOCATABLE` and `POINTER` arrays are not allowed within `FIRSTPRIVATE` or `LASTPRIVATE` clauses. Assumed-size and assumed-shape arrays are not allowed within `PRIVATE`, `FIRSTPRIVATE`, or `LASTPRIVATE` clauses.

4.4 Work-sharing Constructs

A work-sharing construct divides the execution of the enclosed code region among the members of the team that encounter it. A work-sharing construct must be enclosed within a parallel region in order for the directive to execute in parallel. The work-sharing directives do not launch new threads, and there is no implied barrier on entry to a work-sharing construct.

The following restrictions apply to the work-sharing directives:

- Work-sharing constructs and `BARRIER` directives must be encountered by all threads in a team or by none at all.
- Work-sharing constructs and `BARRIER` directives must be encountered in the same order by all threads in a team.

The following sections describe the work-sharing directives:

- Section 4.4.1, page 96, describes the `DO` and `END DO` directives.
- Section 4.4.2, page 99, describes the `SECTIONS`, `SECTION`, and `END SECTIONS` directives.
- Section 4.4.3, page 101, describes the `SINGLE` and `END SINGLE` directives.

4.4.1 Specify Parallel Execution: `DO` and `END DO` Directives

The `DO` directive specifies that the iterations of the immediately following `DO` loop must be divided among the threads in the parallel region. If there is no enclosing parallel region, the `DO` loop is executed serially.

The loop that follows a `DO` directive cannot be a `DO WHILE` or a `DO` loop without loop control.

The format of this directive is as follows:

```
!$OMP DO [clause[[,] clause]...]

do_loop

[!$OMP END DO [NOWAIT]]
```

clause	*clause* can be one of the following: • `PRIVATE(`*var*`[, `*var*`] ...)` • `FIRSTPRIVATE(`*var*`[, `*var*`] ...)` • `LASTPRIVATE(`*var*`[, `*var*`] ...)` • `REDUCTION({`*operator*`|`*intrinsic*`}:`*var*`[, `*var*`] ...)` • `SCHEDULE(`*type*`[,`*chunk*`])` • `ORDERED` The `SCHEDULE` and `ORDERED` clauses are described in this section. The `PRIVATE`, `FIRSTPRIVATE`, `LASTPRIVATE`, and `REDUCTION` clauses are described in Section 4.7.2, page 113.
do_loop	A `DO` loop.

If ordered sections are contained in the dynamic extent of the `DO` directive, the `ORDERED` clause must be present. The code enclosed within an ordered section is executed in the order in which it would be executed in a sequential execution of the loop. For more information on ordered sections, see the `ORDERED` directive in Section 4.6.6, page 110.

The `SCHEDULE` clause specifies how iterations of the `DO` loop are divided among the threads of the team. Within the `SCHEDULE(`*type*`[,`*chunk*`])` clause syntax, *type* can be one of the following:

type	Effect
`STATIC`	When `SCHEDULE(STATIC,`*chunk*`)` is specified, 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.
	When no *chunk* is specified, the iterations are divided among threads in contiguous pieces, and one piece is assigned to each thread. Default.
`DYNAMIC`	When `SCHEDULE(DYNAMIC,`*chunk*`)` is specified, the iterations are broken into pieces of a size specified by *chunk*. As each thread finishes its iterations, it dynamically obtains the next set of iterations.
	When no *chunk* is specified, it defaults to 1.
`GUIDED`	When `SCHEDULE(GUIDED,`*chunk*`)` is specified, each of the iterations are handed out in pieces of exponentially decreasing size. *chunk* specifies the minimum number of iterations to dispatch each time, except when there are less than *chunk* number of iterations, at which point the rest are dispatched.
	When no *chunk* is specified, it defaults to 1.
`RUNTIME`	When `SCHEDULE(RUNTIME)` is specified, the decision regarding scheduling is deferred until run time and you cannot specify a *chunk*.
	The schedule type and chunk size can be chosen at run time by setting the `OMP_SCHEDULE` environment variable. If this environment variable is not set, the resulting schedule is `STATIC`.
	For more information on the `OMP_SCHEDULE` environment variable, see the `pe_environ`(5) man page.

OpenMP: The OpenMP Fortran API does not define a default scheduling mechanism. You should not rely on a particular implementation of a schedule type for correct execution because it is possible to have variations in the implementations of the same schedule type across different compilers.

If an `END DO` directive is not specified, it is assumed at the end of the `DO` loop. If `NOWAIT` is specified on the `END DO` directive, threads do not synchronize at the end of the parallel loop. Threads that finish early proceed straight to the instructions following the loop without waiting for the other members of the team to finish the `DO` directive.

Example. If there are multiple independent loops within a parallel region, you can use the `NOWAIT` clause to avoid the implied `BARRIER` at the end of the `DO` directive, as follows:

```
!$OMP PARALLEL
!$OMP DO
      DO I=2,N
        B(I) = (A(I) + A(I-1)) / 2.0
      ENDDO
!$OMP END DO NOWAIT
!$OMP DO
      DO I=1,M
        Y(I) = SQRT(Z(I))
      ENDDO
!$OMP END DO NOWAIT
!$OMP END PARALLEL
```

Parallel `DO` loop control variables are block-level entities within the `DO` loop. If the loop control variable also appears in the `LASTPRIVATE` variable list of the parallel `DO`, it is copied out to a variable of the same name in the enclosing `PARALLEL` region. The variable in the enclosing `PARALLEL` region must be `SHARED` if it is specified on the `LASTPRIVATE` variable list of a `DO` directive.

The following restrictions apply to the `DO` directives:

- You cannot branch out of a `DO` loop associated with a `DO` directive.
- The values of the loop control parameters of the `DO` loop associated with a `DO` directive must be the same for all the threads in the team.
- The `DO` loop iteration variable must be of type integer.
- If used, the `END DO` directive must appear immediately after the end of the loop.
- Only a single `SCHEDULE` clause can appear on a `DO` directive.
- Only a single `ORDERED` clause can appear on a `DO` directive.

4.4.2 Mark Code for Specific Threads: `SECTION`, `SECTIONS` and `END SECTIONS` Directives

The `SECTIONS` directive specifies that the enclosed sections of code are to be divided among threads in the team. It is a noniterative work-sharing construct. Each section is executed once by a thread in the team.

The format of this directive is as follows:

```
!$OMP SECTIONS [clause[[,] clause]...]

[!$OMP SECTION]

block

[!$OMP SECTION

block]

. . .

!$OMP END SECTIONS [NOWAIT]
```

clause The *clause* can be one of the following:

- `PRIVATE(`*var*[, *var*] ...`)`
- `FIRSTPRIVATE(`*var*[, *var*] ...`)`
- `LASTPRIVATE(`*var*[, *var*] ...`)`
- `REDUCTION({` *operator*`|`*intrinsic*`}:`*var*[, *var*] ...`)`

The `PRIVATE`, `FIRSTPRIVATE`, `LASTPRIVATE`, and `REDUCTION` clauses are described in Section 4.7.2, page 113.

block Denotes a structured block of Fortran statements. You cannot branch into or out of the block.

Each section must be preceded by a `SECTION` directive, though the `SECTION` directive is optional for the first section. The `SECTION` directives must appear within the lexical extent of the `SECTIONS/END SECTIONS` directive pair. The last section ends at the `END SECTIONS` directive. Threads that complete execution of their sections wait at a barrier at the `END SECTIONS` directive unless a `NOWAIT` is specified.

The following restrictions apply to the `SECTIONS` directive:

- The code enclosed in a `SECTIONS/END SECTIONS` directive pair must be a structured block. In addition, each constituent section must also be a structured block. You cannot branch into or out of the constituent section blocks.
- You cannot have a `SECTION` directive outside the lexical extent of the `SECTIONS/END SECTIONS` directive pair.

4.4.3 Request Single–thread Execution: `SINGLE` and `END SINGLE` Directives

The `SINGLE` directive specifies that the enclosed code is to be executed by only one thread in the team. Threads in the team that are not executing the `SINGLE` directive wait at the `END SINGLE` directive unless `NOWAIT` is specified.

The format of this directive is as follows:

```
!$OMP SINGLE [clause[[,] clause]...]

 block

!$OMP END SINGLE [NOWAIT]
```

clause The *clause* can be one of the following:

- `PRIVATE(`*var*`[, ` *var*`] ...)`
- `FIRSTPRIVATE(`*var*`[, ` *var*`] ...)`

The `PRIVATE` and `FIRSTPRIVATE` clauses are described in Section 4.7.2, page 113.

block Denotes a structured block of Fortran statements. You cannot branch into or out of the block.

Example. In the following code fragment, the first thread that encounters the `SINGLE` directive executes subroutines `OUTPUT` and `INPUT`. You must not make any assumptions as to which thread will execute the `SINGLE` section. All other threads will skip the `SINGLE` section and stop at the barrier at the `END SINGLE` construct. If other threads can proceed without waiting for the thread executing the `SINGLE` section, a `NOWAIT` clause can be specified on the `END SINGLE` directive.

```
!$OMP PARALLEL DEFAULT(SHARED)
      CALL WORK(X)
!$OMP BARRIER
!$OMP SINGLE
      CALL OUTPUT(X)
      CALL INPUT(Y)
!$OMP END SINGLE
      CALL WORK(Y)
!$OMP END PARALLEL
```

4.5 Combined Parallel Work-sharing Constructs

The combined parallel work-sharing constructs are shortcuts for specifying a parallel region that contains only one work-sharing construct. The semantics of these directives are identical to that of explicitly specifying a `PARALLEL` directive followed by a single work-sharing construct.

The following sections describe the combined parallel work-sharing directives:

- Section 4.5.1, page 102, describes the `PARALLEL DO` and `END PARALLEL DO` directives.
- Section 4.5.2, page 103, describes the `PARALLEL SECTIONS` and `END PARALLEL SECTIONS` directives.

4.5.1 Declare a Parallel Region: `PARALLEL DO` and `END PARALLEL DO` Directives

The `PARALLEL DO` directive provides a shortcut form for specifying a parallel region that contains a single `DO` directive.

The format of this directive is as follows:

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

do_loop

[!$OMP END PARALLEL DO]
```

clause *clause* can be one or more of the clauses accepted by the `PARALLEL` directive or the `DO` directive. These clauses are as follows:

- `PRIVATE(`*var*`[, `*var*`] ...)`
- `FIRSTPRIVATE(`*var*`[, `*var*`] ...)`
- `LASTPRIVATE(`*var*`[, `*var*`] ...)`
- `REDUCTION({`*operator*`|`*intrinsic*`}:`*var*`[, `*var*`] ...)`
- `SCHEDULE(`*type*`[,`*chunk*`])`
- `ORDERED`
- `SHARED(`*var*`[, `*var*`] ...)`

- DEFAULT(PRIVATE | SHARED | NONE)
- IF(*scalar_logical_expression*)
- COPYIN(*var*[, *var*] ...)

The SCHEDULE and ORDERED clauses are described in Section 4.4.1, page 96. The IF clause is described in Section 4.3, page 94. The SHARED, DEFAULT, COPYIN, PRIVATE, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses are described in Section 4.7.2, page 113.

For information on the PARALLEL directive, see Section 4.3, page 94. For information on the DO directive, see Section 4.4.1, page 96.

do_loop	A DO loop.

If the END PARALLEL DO directive is not specified, the PARALLEL DO is assumed to end with the DO loop that immediately follows the PARALLEL DO directive. If used, the END PARALLEL DO directive must appear immediately after the end of the DO loop.

The semantics are identical to explicitly specifying a PARALLEL directive immediately followed by a DO directive.

Example. The following example shows how to parallelize a simple loop:

```
!$OMP PARALLEL DO
      DO I=1,N
        B(I) = (A(I) + A(I-1)) / 2.0
      ENDDO
!$OMP END PARALLEL DO
```

In the preceding code, the loop iteration variable is private by default, so it is not necessary to declare it explicitly. The END PARALLEL DO directive is optional.

4.5.2 Declare Sections within a Parallel Region: PARALLEL SECTIONS and END PARALLEL SECTIONS Directives

The PARALLEL SECTIONS directive provides a shortcut form for specifying a parallel region that contains a single SECTIONS directive. The semantics are identical to explicitly specifying a PARALLEL directive immediately followed by a SECTIONS directive.

The format of this directive is as follows:

```
!$OMP PARALLEL SECTIONS [clause[[,] clause]...]

[!$OMP SECTION ]

block

[!$OMP SECTION

block]

. . .

!$OMP END PARALLEL SECTIONS
```

clause — *clause* can be one or more of the clauses accepted by the `PARALLEL` directive or the `SECTIONS` directive. These clauses are as follows:

- `PRIVATE(`*var*`[,` *var*`] ...)`
- `FIRSTPRIVATE(`*var*`[,` *var*`] ...)`
- `LASTPRIVATE(`*var*`[,` *var*`] ...)`
- `REDUCTION({` *operator*`|`*intrinsic*`}:`*var*`[,` *var*`] ...)`
- `SHARED(`*var*`[,` *var*`] ...)`
- `DEFAULT(PRIVATE | SHARED | NONE)`
- `IF(`*scalar_logical_expression*`)`
- `COPYIN(`*var*`[,` *var*`] ...)`

The `IF` clause is described in Section 4.3, page 94. The `SHARED`, `DEFAULT`, `FIRSTPRIVATE`, `REDUCTION`, `COPYIN`, `PRIVATE`, `FIRSTPRIVATE`, `LASTPRIVATE`, and `REDUCTION` clauses are described in Section 4.7.2, page 113.

For more information on the `PARALLEL` directive, see Section 4.3, page 94. For more information on the `SECTIONS` directive, see Section 4.4.2, page 99.

block	Denotes a structured block of Fortran statements. You cannot branch into or out of the block.

The last section ends at the `END PARALLEL SECTIONS` directive.

Example. In the following code fragment, subroutines `XAXIS`, `YAXIS`, and `ZAXIS` can be executed concurrently. The first `SECTION` directive is optional. All the `SECTION` directives need to appear in the lexical extent of the `PARALLEL SECTIONS/END PARALLEL SECTIONS` construct.

```
!$OMP PARALLEL SECTIONS
!$OMP SECTION
      CALL XAXIS
!$OMP SECTION
      CALL YAXIS
!$OMP SECTION
      CALL ZAXIS
!$OMP END PARALLEL SECTIONS
```

4.6 Synchronization Constructs

The following sections describe the synchronization constructs:

- Section 4.6.1, page 105, describes the `MASTER` and `END MASTER` directives.
- Section 4.6.2, page 106, describes the `CRITICAL` and `END CRITICAL` directives.
- Section 4.6.3, page 107, describes the `BARRIER` directive.
- Section 4.6.4, page 107, describes the `ATOMIC` directive.
- Section 4.6.5, page 109, describes the `FLUSH` directive.
- Section 4.6.6, page 110, describes the `ORDERED` and `END ORDERED` directives.

4.6.1 Request Execution by the Master Thread: `MASTER` and `END MASTER` Directives

The code enclosed within `MASTER` and `END MASTER` directives is executed by the master thread.

These directives have the following format:

```
!$OMP MASTER

block

!$OMP END MASTER
```

block Denotes a structured block of Fortran statements. You cannot branch into or out of the block.

The other threads in the team skip the enclosed section of code and continue execution. There is no implied barrier either on entry to or exit from the master section.

4.6.2 Request Execution by a Single Thread: `CRITICAL` and `END CRITICAL` Directives

The `CRITICAL` and `END CRITICAL` directives restrict access to the enclosed code to one thread at a time.

These directives have the following format:

```
!$OMP CRITICAL [(name)]

block

!$OMP END CRITICAL [(name)]
```

name Identifies the critical section.

If a *name* is specified on a `CRITICAL` directive, the same *name* must also be specified on the `END CRITICAL` directive. If no *name* appears on the `CRITICAL` directive, no *name* can appear on the `END CRITICAL` directive.

block Denotes a structured block of Fortran statements. You cannot branch into or out of the block.

A thread waits at the beginning of a critical section until no other thread in the team is executing a critical section with the same name. 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.

Example. The following code fragment includes several `CRITICAL` directives. The example illustrates a queuing model in which a task is dequeued and worked on. To guard against multiple threads dequeuing the same task, the dequeuing operation must be in a critical section. Because there are two independent queues in this example, each queue is protected by `CRITICAL` directives with different names, `XAXIS` and `YAXIS`, respectively.

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(X,Y)
!$OMP CRITICAL(XAXIS)
      CALL DEQUEUE(IX_NEXT, X)
!$OMP END CRITICAL(XAXIS)
      CALL WORK(IX_NEXT, X)
!$OMP CRITICAL(YAXIS)
      CALL DEQUEUE(IY_NEXT,Y)
!$OMP END CRITICAL(YAXIS)
      CALL WORK(IY_NEXT, Y)
!$OMP END PARALLEL
```

4.6.3 Synchronize All Threads in a Team: `BARRIER` Directive

The `BARRIER` directive synchronizes all the threads in a team. When it encounters a barrier, a thread waits until all other threads in that team have reached the same point.

This directive has the following format:

```
!$OMP BARRIER
```

4.6.4 Protect a Location from Multiple Updates: `ATOMIC` Directive

The `ATOMIC` directive ensures that a specific memory location is updated atomically, rather than exposing it to the possibility of multiple, simultaneous writing threads.

This directive has the following format:

```
!$OMP ATOMIC
```

This directive applies only to the immediately following statement, which must have one of the following forms:

```
x = x operator expr

x = expr operator x

x = intrinsic (x, expr)

x = intrinsic (expr, x)
```

In the preceding statements:

- *x* is a scalar variable of intrinsic type. All references to storage location *x* must have the same type and type parameters.
- *expr* 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.`, or `.NEQV.` .

Only the load and store of *x* are atomic; the evaluation of *expr* is not atomic. To avoid race conditions, all updates of the location in parallel must be protected with the `ATOMIC` directive, except those that are known to be free of race conditions.

Example 1. The following code fragment uses the `ATOMIC` directive:

```
!$OMP ATOMIC
      X(INDEX(I)) = Y(INDEX(I)) + B
```

Example 2. The following code fragment avoids race conditions by protecting all simultaneous updates of the location, by multiple threads, with the `ATOMIC` directive:

```
!$OMP PARALLEL DO DEFAULT(PRIVATE) SHARED(X,Y,INDEX,N)
      DO I=1,N
        CALL WORK(XLOCAL, YLOCAL)
!$OMP ATOMIC
        X(INDEX(I)) = X(INDEX(I)) + XLOCAL
        Y(I) = Y(I) + YLOCAL
      ENDDO
```

Note that the `ATOMIC` directive applies only to the Fortran statement that immediately follows it. As a result, `Y` is not updated atomically in the preceding code.

4.6.5 Read and Write Variables to Memory: FLUSH Directive

The FLUSH directive identifies synchronization points at which thread-visible variables are written back to memory. This directive must appear at the precise point in the code at which the synchronization is required.

Thread-visible variables include the following data items:

- Globally visible variables (common blocks and modules)
- Local variables that do not have the SAVE attribute but have had their address taken and saved or have had their address passed to another subprogram
- Local variables that do not have the SAVE attribute that are declared shared in a parallel region within the subprogram
- Dummy arguments
- All pointer dereferences

This directive has the following format:

```
!$OMP FLUSH [(var[, var] ...)]
```

var	Variables to be flushed.

An implicit FLUSH directive is assumed for the following directives:

- BARRIER
- CRITICAL and END CRITICAL
- END DO
- END PARALLEL
- END SECTIONS
- END SINGLE
- ORDERED and END ORDERED

The directive is not implied if a NOWAIT clause is present.

Example. The following example uses the FLUSH directive for point-to-point synchronization between pairs of threads:

```
!$OMP PARALLEL DEFAULT(PRIVATE) SHARED(ISYNC)
      IAM = OMP_GET_THREAD_NUM()
      ISYNC(IAM) = 0
!$OMP BARRIER
      CALL WORK()
!
!I AM DONE WITH MY WORK, SYNCHRONIZE WITH MY NEIGHBOR
!
      ISYNC(IAM) = 1
!$OMP FLUSH(ISYNC)
!
!WAIT TILL NEIGHBOR IS DONE
!
      DO WHILE (ISYNC(NEIGH) .EQ. 0)
!$OMP FLUSH(ISYNC)
      END DO
!$OMP END PARALLEL
```

4.6.6 Request Sequential Ordering: ORDERED and END ORDERED Directives

The code enclosed within ORDERED and END ORDERED directives is executed in the order in which it would be executed in a sequential execution of an enclosing parallel loop.

These directives have the following format:

```
!$OMP ORDERED

block

!$OMP END ORDERED
```

block Denotes a structured block of Fortran statements. You cannot branch into or out of the block.

An ORDERED directive can appear only in the dynamic extent of a DO or PARALLEL DO directive. This DO directive must have the ORDERED clause specified. For more information on the DO directive, see Section 4.4.1, page 96. For information on directive binding, see Section 4.8, page 121.

Only one thread is allowed in an ordered section at a time. Threads are allowed to enter in the order of the loop iterations. No thread can enter an ordered section until it is guaranteed that all previous iterations have completed or will never execute an ordered section. This sequentializes and orders code within ordered sections while allowing code outside the section to run in parallel. `ORDERED` sections that bind to different `DO` directives are independent of each other.

The following restrictions apply to the `ORDERED` directive:

- An `ORDERED` directive cannot bind to a `DO` directive that does not have the `ORDERED` clause specified.
- An iteration of a loop with a `DO` directive must not execute the same `ORDERED` directive more than once, and it must not execute more than one `ORDERED` directive.

Example. Ordered sections are useful for sequentially ordering the output from work that is done in parallel. Assuming that a reentrant I/O library exists, the following program prints out the indexes in sequential order:

```
!$OMP DO ORDERED SCHEDULE(DYNAMIC)
      DO I=LB,UB,ST
        CALL WORK(I)
      END DO

      SUBROUTINE WORK(K)
!$OMP ORDERED
      WRITE(*,*) K
!$OMP END ORDERED
      END
```

4.7 Data Environment Constructs

The following subsections present constructs for controlling the data environment during the execution of parallel constructs. Section 4.7.1, page 111, describes the `THREADPRIVATE` directive, which makes common blocks local to a thread. Section 4.7.2, page 113, describes directive clauses that affect the data environment.

4.7.1 Declare Common Blocks Private to a Thread: `THREADPRIVATE` Directive

The `THREADPRIVATE` directive makes named common blocks private to a thread but global within the thread. In other words, each thread executing a `THREADPRIVATE` directive receives its own private copy of the named common blocks, which are then available to it in any routine within the scope of an application.

This directive must appear in the declaration section of the routine after the declaration of the listed common blocks. Each thread gets its own copy of the common block, so data written to the common block by one thread is not directly visible to other threads. During serial portions and MASTER sections of the program, accesses are to the master thread's copy of the common block.

On entry to the first parallel region, data in the THREADPRIVATE common blocks should be assumed to be undefined unless a COPYIN clause is specified on the PARALLEL directive. When a common block that is initialized using DATA statements appears in a THREADPRIVATE directive, each thread's copy is initialized once prior to its first use. For subsequent parallel regions, the data in the THREADPRIVATE common blocks are guaranteed to persist only if the dynamic threads mechanism has been disabled and if the number of threads are the same for all the parallel regions.

For more information on dynamic threads, see the OMP_SET_DYNAMIC(3) library routine and the OMP_DYNAMIC environment variable on the pe_environ(5) man page.

The format of this directive is as follows:

```
!$OMP THREADPRIVATE(/cb/[,/cb/]...)
```

cb The name of the common block to be made private to a thread. Only named common blocks can be made thread private.

The following restrictions apply to the THREADPRIVATE directive:

- The THREADPRIVATE directive must appear after every declaration of a thread private common block.
- You cannot use a THREADPRIVATE common block or its constituent variables in any clause other than a COPYIN clause. As a result, they are not permitted in a PRIVATE, FIRSTPRIVATE, LASTPRIVATE, SHARED, or REDUCTION clause. They are not affected by the DEFAULT clause.

You can use the mp_shmem library routines for communicating between threads. For information on these routines, see the mp(3F) man page.

4.7.2 Data Scope Attribute Clauses

Several directives accept clauses that allow a user to control the scope attributes of variables for the duration of the construct. Not all of the clauses in this section are allowed on all directives, but the clauses that are valid on a particular directive are included with the description of the directive. Usually, if no data scope clauses are specified for a directive, the default scope for variables affected by the directive is `SHARED`. Exceptions to this are described in Section 4.7.3, page 120.

The following sections describe the data scope attribute clauses:

- Section 4.7.2.1, page 113, describes the `PRIVATE` clause.
- Section 4.7.2.2, page 114, describes the `SHARED` clause.
- Section 4.7.2.3, page 114, describes the `DEFAULT` clause.
- Section 4.7.2.4, page 115, describes the `FIRSTPRIVATE` clause.
- Section 4.7.2.5, page 116, describes the `LASTPRIVATE` clause.
- Section 4.7.2.6, page 117, describes the `REDUCTION` clause.
- Section 4.7.2.7, page 119, describes the `COPYIN` clause.

4.7.2.1 `PRIVATE` Clause

The `PRIVATE` clause declares variables to be private to each thread in a team.

This clause has the following format:

```
PRIVATE(var[, var] ...)
```

var	A named variable or named common block that is accessible in the scoping unit. Subobjects cannot be specified. If a named common block is specified, its name must appear between slashes.

The behavior of a variable declared in a `PRIVATE` clause is as follows:

- A new object of the same type is declared once for each thread in the team. The new object is no longer storage associated with the storage location of the original object.
- All references to the original object in the lexical extent of the directive construct are replaced with references to the private object.

- Variables defined as PRIVATE are undefined for each thread on entering the construct and the corresponding shared variable is undefined on exit from a parallel construct.
- Contents, allocation state, and association status of variables defined as PRIVATE are undefined when they are referenced outside the lexical extent (but inside the dynamic extent) of the construct, unless they are passed as actual arguments to called routines.

Example. The following example shows how to scope variables with the PRIVATE clause:

```
      INTEGER I,J
      I = 1
      J = 2
!$OMP PARALLEL PRIVATE(I) FIRSTPRIVATE(J)
      I = 3
      J = J+ 2
!$OMP END PARALLEL
      PRINT *, I, J
```

In the preceding code, the values of I and J are undefined on exit from the parallel region.

4.7.2.2 SHARED Clause

The SHARED clause makes variables shared among all the threads in a team. All threads within a team access the same storage area for SHARED data.

This clause has the following format:

```
SHARED(var[, var] ...)
```

var	A named variable or named common block that is accessible in the scoping unit. Subobjects cannot be specified. If a named common block is specified, its name must appear between slashes.

4.7.2.3 DEFAULT Clause

The DEFAULT clause allows the user to specify a PRIVATE, SHARED, or NONE default scope attribute for all variables in the lexical extent of any parallel region. Variables in THREADPRIVATE common blocks are not affected by this clause.

This clause has the following format:

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

The `PRIVATE`, `SHARED`, and `NONE` specifications have the following effects:

- Specifying `DEFAULT(PRIVATE)` makes all named objects in the lexical extent of the parallel region, including common block variables but excluding `THREADPRIVATE` variables, private to a thread as if each variable were listed explicitly in a `PRIVATE` clause.
- Specifying `DEFAULT(SHARED)` makes all named objects in the lexical extent of the parallel region shared among the threads in a team, as if each variable were listed explicitly in a `SHARED` clause. In the absence of an explicit `DEFAULT` clause, the default behavior is the same as if `DEFAULT(SHARED)` were specified.
- Specifying `DEFAULT(NONE)` declares that there is no implicit default as to whether variables are `PRIVATE` or `SHARED`. In this case, the `PRIVATE`, `SHARED`, `FIRSTPRIVATE`, `LASTPRIVATE`, or `REDUCTION` attribute of each variable used in the lexical extent of the parallel region must be specified.

Only one `DEFAULT` clause can be specified on a `PARALLEL` directive.

Variables can be exempted from a defined default using the `PRIVATE`, `SHARED`, `FIRSTPRIVATE`, `LASTPRIVATE`, and `REDUCTION` clauses. As a result, the following example is valid:

```
!$OMP PARALLEL DO DEFAULT(PRIVATE), FIRSTPRIVATE(I),SHARED(X),
!$OMP& SHARED(R) LASTPRIVATE(I)
```

4.7.2.4 `FIRSTPRIVATE` Clause

The `FIRSTPRIVATE` clause provides a superset of the functionality provided by the `PRIVATE` clause.

This clause has the following format:

`FIRSTPRIVATE(`*var*`[, `*var*`] ...)`

var	A named variable or named common block that is accessible in the scoping unit. Subobjects cannot be specified. If a named common block is specified, its name must appear between slashes.

Variables specified are subject to PRIVATE clause semantics described in Section 4.7.2.1, page 113. In addition, private copies of the variables are initialized from the original object existing before the construct.

4.7.2.5 LASTPRIVATE Clause

The LASTPRIVATE clause provides a superset of the functionality provided by the PRIVATE clause.

When the LASTPRIVATE clause appears on a DO directive, the thread that executes the sequentially last iteration updates the version of the object it had before the construct. When the LASTPRIVATE clause appears in a SECTIONS directive, the thread that executes the lexically last SECTION updates the version of the object it had before the construct. Subobjects that are not assigned a value by the last iteration of the DO or the lexically last SECTION of the SECTIONS directive are undefined after the construct.

This clause has the following format:

```
LASTPRIVATE(var[, var] ...)
```

var A named variable or named common block that is accessible in the scoping unit. Subobjects cannot be specified. If a named common block is specified, its name must appear between slashes.

Each *var* is subject to the PRIVATE clause semantics described in Section 4.7.2.1, page 113.

Example. Correct execution sometimes depends on the value that the last iteration of a loop assigns to a variable. Such programs must list all such variables as arguments to a LASTPRIVATE clause so that the values of the variables are the same as when the loop is executed sequentially.

```
!$OMP PARALLEL
!$OMP DO LASTPRIVATE(I)
      DO I=1,N
        A(I) = B(I) + C(I)
      ENDDO
!$OMP END PARALLEL
      CALL REVERSE(I)
```

In the preceding code fragment, the value of `I` at the end of the parallel region will equal `N+1`, as in the sequential case.

4.7.2.6 `REDUCTION` Clause

This clause performs a reduction on the variables specified, with the operator or the intrinsic specified.

This clause has the following format:

```
REDUCTION({operator|intrinsic}:var[, var] ...)
```

operator	Specify one of the following: `+`, `*`, `-`, `.AND.`, `.OR.`, `.EQV.`, or `.NEQV.`
intrinsic	Specify one of the following: `MAX`, `MIN`, `IAND`, `IOR`, or `IEOR`.
var	A named variable or named common block that is accessible in the scoping unit. Subobjects cannot be specified. Each *var* must be a named scalar variable of intrinsic type. Variables that appear in a `REDUCTION` clause must be `SHARED` in the enclosing context. A private copy of each *var* is created for each thread as if the `PRIVATE` clause had been used. The private copy is initialized according to the operator. For more information, see Table 1, page 118. If a named common block is specified, its name must appear between slashes.

At the end of the `REDUCTION`, the shared variable is updated to reflect the result of combining the original value of the (shared) reduction variable with the final value of each of the private copies using the operator specified. The reduction operators are all associative (except for subtraction), and the compiler can freely reassociate the computation of the final value (the partial results of a subtraction reduction are added to form the final value).

The value of the shared variable becomes undefined when the first thread reaches the containing clause, and it remains so until the reduction computation is complete. Normally, the computation is complete at the end of the `REDUCTION` construct; however, if the `REDUCTION` clause is used on a construct to which `NOWAIT` is also applied, the shared variable remains undefined until a barrier synchronization has been performed to ensure that all the threads have completed the `REDUCTION` clause.

The `REDUCTION` clause is intended to be used on a region or work-sharing construct in which the reduction variable is used only in reduction statements with one of the following forms:

```
x = x operator expr

x = expr operator x  (except for subtraction)

x = intrinsic (x,expr)

x = intrinsic (expr, x)
```

Some reductions can be expressed in other forms. For instance, a `MAX` reduction might be expressed as follows:

```
IF (x .LT. expr) x = expr
```

Alternatively, the reduction might be hidden inside a subroutine call. You must ensure that the operator specified in the `REDUCTION` clause matches the reduction operation.

The following table lists the operators and intrinsics that are valid and their canonical initialization values. The actual initialization value will be consistent with the data type of the reduction variable.

Table 1. Initialization values

Operator/Intrinsic	Initialization
`+`	0
`*`	1
`-`	0
`.AND.`	`.TRUE.`
`.OR.`	`.FALSE.`
`.EQV.`	`.TRUE.`
`.NEQV.`	`.FALSE.`
`MAX`	Smallest representable number
`MIN`	Largest representable number

Operator/Intrinsic	Initialization
IAND	All bits on
IOR	0
IEOR	0

Any number of reduction clauses can be specified on the directive, but a variable can appear only once in a REDUCTION clause for that directive.

Example 1. The following directive line shows use of the REDUCTION clause:

```
!$OMP DO REDUCTION(+: A, Y) REDUCTION(.OR.: AM)
```

Example 2. The following code fragment shows how to use the REDUCTION clause:

```
!$OMP PARALLEL DO DEFAULT(PRIVATE) REDUCTION(+: A,B)
      DO I=1,N
        CALL WORK(ALOCAL,BLOCAL)
        A = A + ALOCAL
        B = B + BLOCAL
      ENDDO
!$OMP END PARALLEL DO
```

4.7.2.7 COPYIN Clause

The COPYIN clause applies only to common blocks that are declared THREADPRIVATE. A COPYIN clause on a parallel region 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.

This clause has the following format:

```
COPYIN(var[, var] ...)
```

var	A named variable or named common block that is accessible in the scoping unit. Subobjects cannot be specified. If a named common block is specified, its name must appear between slashes.

It is not necessary to specify a whole common block to be copied in.

Example. In the following example, the common blocks BLK1 and FIELDS are specified as thread private, but only one of the variables in common block FIELDS is specified to be copied in:

```
      COMMON /BLK1/ SCRATCH
      COMMON /FIELDS/ XFIELD, YFIELD, ZFIELD
!$OMP THREADPRIVATE(/BLK1/, /FIELDS/)
!$OMP PARALLEL DEFAULT(PRIVATE) COPYIN(/BLK1/,ZFIELD)
```

4.7.3 Data Environment Rules

The following rules and restrictions apply with respect to data scope:

1. Sequential DO loop control variables in the lexical extent of a PARALLEL region that would otherwise be SHARED based on default rules are automatically made private on the PARALLEL directive. Sequential DO loop control variables with no enclosing PARALLEL region are not classified automatically. You must guarantee that these indexes are private if the containing procedures are called from a PARALLEL region.

   All implied DO loop control variables are automatically made private at the enclosing implied DO construct.

2. Variables that are made private in a parallel region cannot be made private again on an enclosed work-sharing directive. As a result, variables that appear in the PRIVATE, FIRSTPRIVATE, LASTPRIVATE, and REDUCTION clauses on a work-sharing directive have shared scope in the enclosing parallel region.

3. A variable that appears in a PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION clause must be definable.

4. Assumed-size and assumed-shape arrays cannot be specified as PRIVATE, FIRSTPRIVATE, or LASTPRIVATE. Array dummy arguments that are explicitly shaped (including variably dimensioned) can be declared in any scoping clause.

5. Fortran pointers and allocatable arrays can be declared as PRIVATE or SHARED but not as FIRSTPRIVATE or LASTPRIVATE.

   Within a parallel region, the initial status of a private pointer is undefined. Private pointers that become allocated during the execution of a parallel region should be explicitly deallocated by the program prior to the end of the parallel region to avoid memory leaks.

The association status of a SHARED pointer becomes undefined upon entry to and on exit from the parallel construct if it is associated with a target or a subobject of a target that is PRIVATE, FIRSTPRIVATE, LASTPRIVATE, or REDUCTION inside the parallel construct. An allocatable array declared PRIVATE has an allocation status of *not currently allocated* on entry to and on exit from the construct.

6. PRIVATE or SHARED attributes can be declared for a Cray pointer but not for the pointee. The scope attribute for the pointee is determined at the point of pointer definition. You cannot declare a scope attribute for a pointee. Cray pointers cannot be specified in FIRSTPRIVATE or LASTPRIVATE clauses.

7. Scope clauses apply only to variables in the static extent of the directive on which the clause appears, with the exception of variables passed as actual arguments. Local variables in called routines that do not have the SAVE attribute are PRIVATE. Common blocks and modules in called routines in the dynamic extent of a parallel region always have an implicit SHARED attribute, unless they are THREADPRIVATE common blocks.

8. When a named common block is declared as PRIVATE, FIRSTPRIVATE, or LASTPRIVATE, none of its constituent elements may be declared in another scope attribute. When individual members of a common block are privatized, the storage of the specified variables is no longer associated with the storage of the common block itself.

9. Variables that are not allowed in the PRIVATE and SHARED clauses are not affected by DEFAULT(PRIVATE) or DEFAULT(SHARED) clauses, respectively.

10. Clauses can be repeated as needed, but each variable can appear explicitly in only one clause per directive, with the following exceptions:

    - A variable can be specified as both FIRSTPRIVATE and LASTPRIVATE.

    - Variables affected by the DEFAULT clause can be listed explicitly in a clause to override the default specification.

4.8 Directive Binding

Some directives are *bound* to other directives. A binding specifies the way in which one directive is related to another. For instance, a directive is bound to a second directive if it can appear in the dynamic extent of that second directive. The following rules apply with respect to the dynamic binding of directives:

- The DO, SECTIONS, SINGLE, MASTER, and BARRIER directives bind to the dynamically enclosing PARALLEL directive, if one exists.
- The ORDERED directive binds to the dynamically enclosing DO directive.
- The ATOMIC directive enforces exclusive access with respect to ATOMIC directives in all threads, not just the current team.
- The CRITICAL directive enforces exclusive access with respect to CRITICAL directives in all threads, not just the current team.
- A directive can never bind to any directive outside the closest enclosing PARALLEL.

Example 1. The directive binding rules call for a BARRIER directive to bind to the closest enclosing PARALLEL directive.

In the following example, the call from MAIN to SUB2 is valid because the BARRIER (in SUB3) binds to the PARALLEL region in SUB2. The call from MAIN to SUB1 is valid because the BARRIER binds to the PARALLEL region in subroutine SUB2.

```
      PROGRAM MAIN
      CALL SUB1(2)
      CALL SUB2(2)
      END

      SUBROUTINE SUB1(N)
!$OMP PARALLEL PRIVATE(I) SHARED(N)
!$OMP DO
      DO I = 1, N
      CALL SUB2(I)
      END DO
!$OMP END PARALLEL
      END

      SUBROUTINE SUB2(K)
!$OMP PARALLEL SHARED(K)
      CALL SUB3(K)
!$OMP END PARALLEL
      END

      SUBROUTINE SUB3(N)
      CALL WORK(N)
!$OMP BARRIER
```

```
      CALL WORK(N)
      END
```

Example 2. The following program shows inner and outer `DO` directives that bind to different `PARALLEL` regions:

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
!$OMP PARALLEL SHARED(I,N)
!$OMP DO
        DO J = 1, N
          CALL WORK(I,J)
        END DO
!$OMP END PARALLEL
      END DO
!$OMP END PARALLEL
```

A following variation of the preceding example also shows correct binding:

```
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
        CALL SOME_WORK(I,N)
      END DO
!$OMP END PARALLEL

      SUBROUTINE SOME_WORK(I,N)
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO J = 1, N
        CALL WORK(I,J)
      END DO
!$OMP END PARALLEL
      RETURN
      END
```

4.9 Directive Nesting

The following rules apply to the dynamic nesting of directives:

- A PARALLEL directive dynamically inside another PARALLEL directive logically establishes a new team, which is composed of only the current thread unless nested parallelism is enabled.
- DO, SECTIONS, and SINGLE directives that bind to the same PARALLEL directive cannot be nested one inside the other.
- DO, SECTIONS, and SINGLE directives are not permitted in the dynamic extent of CRITICAL and MASTER directives.
- BARRIER directives are not permitted in the dynamic extent of DO, SECTIONS, SINGLE, MASTER, and CRITICAL directives.
- MASTER directives are not permitted in the dynamic extent of DO, SECTIONS, and SINGLE directives.
- ORDERED sections are not allowed in the dynamic extent of CRITICAL sections.
- Any directive set that is legal when executed dynamically inside a PARALLEL region is also legal when executed outside a parallel region. When executed dynamically outside a user-specified parallel region, the directive is executed with respect to a team composed of only the master thread.

Example 1. The following example is incorrect because the inner and outer DO directives are nested and bind to the same PARALLEL directive:

```
      PROGRAM WRONG1
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
!$OMP DO
        DO J = 1, N
          CALL WORK(I,J)
        END DO
      END DO
!$OMP END PARALLEL
      END
```

The following dynamically nested version of the preceding code is also incorrect:

```
      PROGRAM WRONG2
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
```

```
        CALL SOME_WORK(I,N)
      END DO
!$OMP END PARALLEL

      SUBROUTINE SOME_WORK(I,N)
!$OMP DO
      DO J = 1, N
        CALL WORK(I,J)
      END DO
      RETURN
      END
```

Example 2. The following example is incorrect because the DO and SINGLE directives are nested, and they bind to the same PARALLEL region:

```
      PROGRAM WRONG3
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
!$OMP SINGLE
      CALL WORK(I)
!$OMP END SINGLE
      END DO
!$OMP END PARALLEL
      END
```

Example 3. The following example is incorrect because a BARRIER directive inside a SINGLE or a DO directive can result in deadlock:

```
      PROGRAM WRONG3
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP DO
      DO I = 1, N
        CALL WORK(I)
!$OMP BARRIER
        CALL MORE_WORK(I)
      END DO
!$OMP END PARALLEL
      END
```

Example 4. The following example is incorrect because the BARRIER results in deadlock due to the fact that only one thread at a time can enter the critical section:

```
      PROGRAM WRONG4
!$OMP PARALLEL DEFAULT(SHARED)
!$OMP CRITICAL
      CALL WORK(N,1)
!$OMP BARRIER
      CALL MORE_WORK(N,2)
!$OMP END CRITICAL
!$OMP END PARALLEL
      END
```

Example 5. The following example is incorrect because the `BARRIER` results in deadlock due to the fact that only one thread executes the `SINGLE` section:

```
      PROGRAM WRONG5
!$OMP PARALLEL DEFAULT(SHARED)
      CALL SETUP(N)
!$OMP SINGLE
      CALL WORK(N,1)
!$OMP BARRIER
      CALL MORE_WORK(N,2)
!$OMP END SINGLE
      CALL FINISH(N)
!$OMP END PARALLEL
      END
```

4.10 Analyzing Data Dependencies for Multiprocessing

The essential condition required to parallelize a loop correctly is that each iteration of the loop must be independent of all other iterations. If a loop meets this condition, then the order in which the iterations of the loop execute is not important. They can be executed backward or at the same time, and the answer is still the same. This property is captured by the notion of *data independence*.

For a loop to be data independent, no iterations of the loop can write a value into a memory location that is read or written by any other iteration of that loop. It is all right if the same iteration reads and/or writes a memory location repeatedly as long as no others do; it is all right if many iterations read the same location as long as none of them write to it.

In a Fortran program, memory locations are represented by variable names. So, to determine if a particular loop can be run in parallel, examine the way variables are used in the loop. Because data dependence occurs only when memory locations are modified, pay particular attention to variables that appear on the left-hand side of assignment statements. If a variable is neither modified nor passed to a function or subroutine, there is no data dependence associated with it.

The Fortran compiler supports four kinds of variable usage within a parallel loop: `SHARED`, `PRIVATE`, `LASTPRIVATE`, and `REDUCTION`. If a variable is declared as `SHARED`, all iterations of the loop use the same copy. If a variable is declared as `PRIVATE`, each iteration is given its own uninitialized copy. A variable is declared `SHARED` if it is only read (not written) within the loop or if it is an array where each iteration of the loop uses a different element of the array. A variable can be `PRIVATE` if its value does not depend on any other iteration and if its value is used only within a single iteration. The `PRIVATE` variable is essentially temporary; a new copy can be created in each loop iteration without changing the final answer. As a special case, if only the last value of a variable computed on the last iteration is used outside the loop (but would otherwise qualify as a `PRIVATE` variable), the loop can be multiprocessed by declaring the variable to be `LASTPRIVATE`.

It is often difficult to analyze loops for data dependence information. Each use of each variable must be examined to determine if it fulfills the criteria for `PRIVATE`, `LASTPRIVATE`, `SHARED`, or `REDUCTION`. If all of the uses conform, the loop can be parallelized. If not, the loop cannot be parallelized as written, but can possibly be rewritten into an equivalent parallel form.

An alternative to manually analyzing variable usage is to use the MIPSpro Auto-Parallelizing Option (APO). This optional software package analyzes loops for data dependence. If the APO software determines that a loop is data-independent, it automatically inserts the required compiler directives. If it cannot determine if the loop is independent, it produces a listing file detailing where the problems lie. For more information on APO, see Chapter 9, page 213.

4.10.1 Dependency Analysis Examples

This section contains examples that show dependency analysis.

Example 1. Simple independence. In this example, each iteration writes to a different location in `A`, and none of the variables appearing on the right-hand side are ever written to; they are only read from. This loop can be correctly run in parallel. All the variables are `SHARED` except for `I`, which is either `PRIVATE` or `LASTPRIVATE`, depending on whether the last value of `I` is used later in the code.

```
DO I = 1,N
A(I) = X + B(I)*C(I)
```

```
      END DO
```

Example 2. Data dependence. The following code fragment contains A(I) on the left-hand side and A(I-1) on the right. This means that one iteration of the loop writes to a location in A and the next iteration reads from that same location. Because different iterations of the loop read and write the same memory location, this loop cannot be run in parallel.

```
         DO I = 2,N
         A(I) = B(I) - A(I-1)
      END DO
```

Example 3. Stride not 1. This example is similar to the previous example. The difference is that the stride of the DO loop is now 2 rather than 1. A(I) now references every other element of A, and A(I-1) references exactly those elements of A that are not referenced by A(I). None of the data locations on the right-hand side is ever the same as any of the data locations written to on the left-hand side. The data are disjoint, so there is no dependence. The loop can be run in parallel. Arrays A and B can be declared SHARED, while variable I should be declared PRIVATE or LASTPRIVATE.

```
      DO I = 2,N,2
         A(I) = B(I) - A(I-1)
      END DO
```

Example 4. Local variable. In the following loop, each iteration of the loop reads and writes the variable X. However, no loop iteration ever needs the value of X from any other iteration. X is used as a temporary variable; its value does not survive from one iteration to the next.

This loop can be parallelized by declaring X to be a PRIVATE variable within the loop. Note that B(I) is both read and written by the loop. This is not a problem because each iteration has a different value for I, so each iteration uses a different B(I). The same B(I) is allowed to be read and written as long as it is done by the same iteration of the loop. The loop can be run in parallel. Arrays A and B can be declared SHARED, while variable I should be declared PRIVATE or LASTPRIVATE.

```
DO I = 1, N
     X = A(I)*A(I) + B(I)
     B(I) = X + B(I)*X
END DO
```

Example 5. Function call. The value of X in any iteration of the following loop is independent of the value of X in any other iteration, so X can be made a

PRIVATE variable. The loop can be run in parallel. Arrays A, B, C, and D can be declared SHARED, while variable I should be declared PRIVATE or LASTPRIVATE.

```
     DO I = 1, N
        X = SQRT(A(I))
        B(I) = X*C(I) + X*D(I)
      END DO
```

This loop invokes an intrinsic function, SQRT. It is possible to use functions and/or subroutines (intrinsic or user defined) within a parallel loop. However, verify that the parallel invocations of the routine do not interfere with one another. In particular, SQRT returns a value that depends only on its input argument, does not modify global data, and does not use static storage (it has no side effects).

The Fortran intrinsic functions have no side effects. The intrinsic functions can be used safely within a parallel loop. The intrinsic subroutines, however, can have side effects. Most Fortran library functions cannot be included in a parallel loop. In particular, rand is not safe for multiprocessing. For user-written routines, it is your responsibility to ensure that the routines can be correctly multiprocessed.

Caution: Do not use the -static option on the f90(1) command line when compiling routines called within a parallel loop.

Example 6. Rewritable data dependence. Here, the value of INDX survives the loop iteration and is carried into the next iteration. This loop cannot be parallelized as it is written. Making INDX a PRIVATE variable does not work; you need the value of INDX computed in the previous iteration. It is possible to rewrite this loop to make it parallel. See Section 4.10.2, page 131, for an example.

```
INDX = 0
DO I = 1, N
     INDX = INDX + I
     A(I) = B(I) + C(INDX)
END DO
```

Example 7. Exit branch. The following loop contains an exit branch; that is, under certain conditions the flow of control suddenly exits the loop. The compiler cannot parallelize loops containing exit branches.

```
       DO I = 1, N
          IF (A(I) .LT. EPSILON) EXIT
          A(I) = A(I) * B(I)
       END DO
```

Example 8. Complicated independence. Initially, it appears that the following loop cannot be run in parallel because it uses both `W(I)` and `W(I-K)`. However, because the value of `I` varies between `K+1` and `2*K`, then `I-K` goes from 1 to `K`. This means that the `W(I-K)` term varies from `W(1)` to `W(K)`, while the `W(I)` term varies from `W(K+1)` to `W(2*K)`. Therefore, `W(I-K)` in any iteration of the loop is never the same memory location as `W(I)` in any other iterations. Because there is no data overlap, there are no data dependencies. This loop can be run in parallel. Elements `W`, `B`, and `K` can be declared `SHARED`, but variable `I` should be declared `PRIVATE` or `LASTPRIVATE`.

```
DO I = K+1, 2*K
     W(I) = W(I) + B(I,K) * W(I-K)
END DO
```

The preceding code illustrates a general rule: the more complex the expression used to index an array, the harder it is to analyze. If the arrays in a loop are indexed only by the loop index variable, the analysis is usually straightforward.

Example 9. Inconsequential data dependence. The data dependence in the following loop is present because it is possible that at some point that `I` will be the same as `INDEX`, so there will be a data location that is being read and written by different iterations of the loop. In this special case, you can simply ignore it. You know that when `I` and `INDEX` are equal, the value written into `A(I)` is exactly the same as the value that is already there. The fact that some iterations of the loop read the value before it is written and some after it is written is not important because they all get the same value. Therefore, this loop can be parallelized. Array `A` can be declared `SHARED`, but variable `I` should be declared `PRIVATE` or `LASTPRIVATE`.

```
INDEX = SELECT(N)
DO I = 1, N
     A(I) = A(INDEX)
END DO
```

Example 10. Local array. In the following code fragment, each iteration of the loop uses the same locations in array `D`. However, closer inspection reveals that array `D` is being used as a temporary. This can be multiprocessed by declaring `D` to be `PRIVATE`. The Fortran compiler allows arrays (even multidimensional arrays) to be `PRIVATE` variables, with the following restrictions: the size of the array must be either a constant or an expression; the dimension bounds must be specified; the `PRIVATE` array cannot have been declared using a variable or the asterisk (*) syntax; and assumed-shape, deferred-shape, and pointer arrays are not permitted.

```
DO I = 1, N
     D(1) = A(I,1) - A(J,1)
     D(2) = A(I,2) - A(J,2)
     D(3) = A(I,3) - A(J,3)
     TOTAL_DISTANCE(I,J) = SQRT(D(1)**2 + D(2)**2 + D(3)**2)
END DO
```

The preceding loop can be parallelized. Arrays TOTAL_DISTANCE and A can be declared SHARED, and array D and variable I can be declared PRIVATE or LASTPRIVATE.

4.10.2 Rewriting Data Dependencies

Many loops that have data dependencies can be rewritten so that some or all of the loop can be run in parallel. You must first locate the statement(s) in the loop that cannot be made parallel and try to find another way to express it that does not depend on any other iteration of the loop. If this fails, try to pull the statements out of the loop and into a separate loop, allowing the remainder of the original loop to be run in parallel.

After you identify data dependencies, you can use various techniques to rewrite the code to break the dependence. Sometimes the dependencies in a loop cannot be broken, and you must either accept the serial execution rate or try to find a new parallel method of solving the problem. The following examples show how to deal with commonly occurring situations. These are by no means exhaustive but cover many situations that happen in practice.

Example 1. Loop-carried value. The following code segment is the same as the rewritable data dependence example in the previous section. INDX has its value carried from iteration to iteration. However, you can compute the appropriate value for INDX without making reference to any previous value.

```
INDX = 0
DO I = 1, N
   INDX = INDX + I
   A(I) = B(I) + C(INDX)
END DO
```

For example, consider the following code:

```
!$OMP PARALLEL DO PRIVATE (I, INDX)
    DO I = 1, N
       INDX = (I*(I+1))/2
       A(I) = B(I) + C(INDX)
    END DO
```

In this loop, the value of INDX is computed without using any values computed on any other iteration. INDX can correctly be made a PRIVATE variable, and the loop can now be multiprocessed.

Example 2. Indirect indexing. Consider the following code:

```
     DO I = 1, N
        IX = INDEXX(I)
        IY = INDEXY(I)
        XFORCE(I) = XFORCE(I) + NEWXFORCE(IX)
        YFORCE(I) = YFORCE(I) + NEWYFORCE(IY)
        IXX = IXOFFSET(IX)
        IYY = IYOFFSET(IY)
        TOTAL(IXX, IYY) = TOTAL(IXX, IYY) + EPSILON
      END DO
```

It is the final statement that causes problems. The indexes IXX and IYY are computed in a complex way and depend on the values from the IXOFFSET and IYOFFSET arrays. It is not known if TOTAL(IXX,IYY) in one iteration of the loop will always be different from TOTAL(IXX,IYY) in every other iteration of the loop.

You can pull the statement out into its own separate loop by expanding IXX and IYY into arrays to hold intermediate values, as follows:

```
!$OMP PARALLEL DO PRIVATE(IX, IY, I)
      DO I  = 1, N
         IX = INDEXX(I)
         IY = INDEXY(I)
         XFORCE(I) = XFORCE(I) + NEWXFORCE(IX)
         YFORCE(I) = YFORCE(I) + NEWYFORCE(IY)
         IXX(I) = IXOFFSET(IX)
         IYY(I) = IYOFFSET(IY)
      END DO
      DO I = 1, N
         TOTAL(IXX(I),IYY(I)) = TOTAL(IXX(I), IYY(I)) + EPSILON
      END DO
```

Here, IXX and IYY have been turned into arrays to hold all the values computed by the first loop. The first loop (containing most of the work) can now be run in parallel. Only the second loop must still be run serially. This is true if IXOFFSET or IYOFFSET are permutation vectors.

If you were certain that the value for IXX was always different in every iteration of the loop, then the original loop could be run in parallel. It could also be run in parallel if IYY was always different. If IXX (or IYY) is always different in every iteration, then TOTAL(IXX,IYY) is never the same location in any iteration of the loop, and so there is no data conflict.

This sort of knowledge is program-specific and should always be used with great care. It may be true for a particular data set, but to run the original code in parallel as it stands, you need to be sure it will always be true for all possible input data sets.

Example 3. Recurrence. The following example shows a *recurrence*, which exists when a value computed in one iteration is immediately used by another iteration. There is no good way of running this loop in parallel. If this type of construct appears in a critical loop, try pulling the statement(s) out of the loop as in the previous example. Sometimes another loop encloses the recurrence; in that case, try to parallelize the outer loop.

```
DO I = 1,N
     X(I) = X(I-1) + Y(I)
END DO
```

Example 4. Sum reduction. The following example shows an operation known as a *reduction*. Reductions occur when an array of values is combined and reduced into a single value.

```
SUM  = 0.0
DO I = 1,N
     SUM = SUM + A(I)
END DO
```

This example is a sum reduction because the combining operation is addition. Here, the value of SUM is carried from one loop iteration to the next, so this loop cannot be multiprocessed. However, because this loop simply sums the elements of A(I), you can rewrite the loop to accumulate multiple, independent subtotals and do much of the work in parallel, as follows:

```
      NUM_THREADS = OMP_GET_NUM_THREADS()
!
!  IPIECE_SIZE = N/NUM_THREADS ROUNDED UP
```

```
!
      IPIECE_SIZE = (N + (NUM_THREADS-1)) / NUM_THREADS
      DO K = 1, NUM_THREADS
        PARTIAL_SUM(K) = 0.0
!
!  THE FIRST THREAD DOES 1 THROUGH IPIECE_SIZE, THE
!  SECOND DOES IPIECE_SIZE + 1 THROUGH 2*IPIECE_SIZE,
!  ETC. IF N IS NOT EVENLY DIVISIBLE BY NUM_THREADS,
!  THE LAST PIECE NEEDS TO TAKE THIS INTO ACCOUNT,
!  HENCE THE "MIN" EXPRESSION.
!
      DO I = K*IPIECE_SIZE - IPIECE_SIZE + 1, MIN(K*IPIECE_SIZE,N)
           PARTIAL_SUM(K) = PARTIAL_SUM(K) + A(I)
        END DO
      END DO
!
!  NOW ADD UP THE PARTIAL SUMS
      SUM = 0.0
      DO I = 1, NUM_THREADS
         SUM = SUM + PARTIAL_SUM(I)
      END DO
```

The outer loop K can be run in parallel. In this method, the array pieces for the partial sums are contiguous, resulting in good cache utilization and performance.

Because this is an important and common transformation, automatic support is provided by the REDUCTION clause:

```
     SUM = 0.0
!$OMP PARALLEL DO PRIVATE (I), REDUCTION (+:SUM)
     DO 10 I = 1, N
        SUM = SUM + A(I)
10 CONTINUE
```

The previous code has essentially the same meaning as the much longer and more confusing code above. Adding an extra dimension to an array to permit parallel computation and then combining the partial results is an important technique for trying to break data dependencies. This technique is often useful.

Reduction transformations such as this do not produce the same results as the original code. Because computer arithmetic has limited precision, when you sum the values together in a different order, as was done here, the round-off errors accumulate slightly differently. It is probable that the final answer will be slightly different from the original loop. Both answers are equally correct. The difference is usually irrelevant, but sometimes it can be significant. If the difference is significant, neither answer is really trustworthy.

This example is a `sum` reduction because the operator is plus (+). The Fortran compiler supports the reduction operations described in Table 1, page 118.

For example,

```
!$OMP PARALLEL DO PRIVATE (I), REDUCTION(+:BG_SUM),
!$OMP+REDUCTION(*:BG_PROD), REDUCTION(MIN:BG_MIN), REDUCTION(MAX:BG_MAX)
          DO I = 1,N
             BG_SUM  = BG_SUM + A(I)
             BG_PROD = BG_PROD * A(I)
             BG_MIN  = MIN(BG_MIN, A(I))
             BG_MAX  = MAX(BG_MAX, A(I))
          END DO
```

The following is another example of a reduction transformation:

```
DO I = 1, N
   TOTAL = 0.0
   DO J = 1, M
      TOTAL = TOTAL + A(J)
   END DO
   B(I) = C(I) * TOTAL
END DO
```

Initially, it might look as if the inner loop should be parallelized with a `REDUCTION` clause. However, consider the outer `I` loop. Although `TOTAL` cannot be made a `PRIVATE` variable in the inner loop, it fulfills the criteria for a `PRIVATE` variable in the outer loop: the value of `TOTAL` in each iteration of the outer loop does not depend on the value of `TOTAL` in any other iteration of the outer loop. Thus, you do not have to rewrite the loop; you can parallelize this reduction on the outer `I` loop, making `TOTAL` and `J` local variables.

4.11 Work Quantum

A certain amount of overhead is associated with multiprocessing a loop. If the work occurring in the loop is small, the loop can actually run slower by multiprocessing than by single processing. To avoid this, make the amount of work inside the multiprocessed region as large as possible, as is shown in the following examples.

Example 1. Loop interchange. Consider the following code:

```
DO K = 1, N
     DO I = 1, N
        DO J = 1, N
           A(I,J) = A(I,J) + B(I,K) * C(K,J)
        END DO
     END DO
END DO
```

For the preceding code fragment, you can parallelize the `J` loop or the `I` loop. You cannot parallelize the `K` loop because different iterations of the `K` loop read and write the same values of `A(I,J)`. Try to parallelize the outermost `DO` loop if possible, because it encloses the most work. In this example, that is the `I` loop. For this example, use the technique called *loop interchange*. Although the parallelizable loops are not the outermost ones, you can reorder the loops to make one of them outermost.

Thus, loop interchange would produce the following code fragment:

```
!$OMP PARALLEL DO PRIVATE(I, J, K)
        DO J = 1, N
           DO K = 1, N
              DO I = 1, N
                 A(I,J) = A(I,J) + B(I,K) * C(K,J)
              END DO
           END DO
        END DO
```

Now the parallelizable loop encloses more work and shows better performance. In practice, relatively few loops can be reordered in this way. However, it does occasionally happen that several loops in a nest of loops are candidates for parallelization. In such a case, it is usually best to parallelize the outermost one.

Example 2. Conditional parallelism. The loop is worth parallelizing if `N` is sufficiently large. To overcome the parallel loop overhead, `N` needs to be around 1000, depending on the specific hardware and the context of the program. The optimized version would uses an `IF` clause on the `PARALLEL DO` directive:

```
!$OMP PARALLEL DO IF (N .GE. 1000), PRIVATE(I)
        DO I = 1, N
           A(I) = A(I) + X*B(I)
        END DO
```

4.12 Cache Effects and Optimization

It is best to try to write loops that take the cache into account, with or without parallelism. The technique for attaining the best cache performance is quite simple: make the loop step through the array in the same way that the array is laid out in memory. For Fortran, this means stepping through the array without any gaps and with the leftmost subscript varying the fastest. This does not depend on multiprocessing, nor is it required in order for multiprocessing to work correctly. However, multiprocessing can affect how the cache is used.

4.12.1 Performing a Matrix Multiply

Consider the following code segment:

```
DO I = 1, N
     DO K = 1, N
        DO J = 1, N
           A(I,J) = A(I,J) + B(I,K) * C(K,J)
        END DO
     END DO
END DO
```

To get the best cache performance, the `I` loop should be innermost. At the same time, to get the best multiprocessing performance, the outermost loop should be parallelized.

For this example, you can interchange the `I` and `J` loops, and get the best of both optimizations:

```
!$OMP PARALLEL DO PRIVATE(I, J, K)
        DO J = 1, N
           DO K = 1, N
              DO I = 1, N
                 A(I,J) = A(I,J) + B(I,K) * C(K,J)
              END DO
           END DO
        END DO
```

4.12.2 Optimization Costs

Sometimes you must choose between the possible optimizations and their costs. Look at the following code segment:

```
DO J = 1, N
     DO I = 1, M
        A(I) = A(I) + B(J)*C(I,J)
     END DO
END DO
```

This loop can be parallelized on `I` but not on `J`. You could interchange the loops to put `I` on the outside, thus getting a bigger work quantum.

```
!$OMP PARALLEL DO PRIVATE(I,J)
     DO I = 1, M
        DO J = 1, N
           A(I) = A(I) + B(J)*C(I,J)
        END DO
     END DO
```

However, putting `J` on the inside means that you will step through the `C` array in the wrong direction; the leftmost subscript should be the one that varies the fastest. It is possible to parallelize the `I` loop where it stands:

```
      DO J = 1, N
!$OMP PARALLEL DO PRIVATE(I)
        DO I = 1, M
           A(I) = A(I) + B(J)*C(I,J)
        END DO
     END DO
```

However, `M` needs to be large for the work quantum to show any improvement. In this example, `A(I)` is used to do a sum reduction, and it is possible to use reduction techniques to rewrite this in a parallel form. However, that involves converting array `A` from a one-dimensional array to a two-dimensional array to hold the partial sums; this is analogous to the way the scalar summation variable was converted into an array of partial sums.

If `A` is large, however, the conversion can take too much memory. It can also take extra time to initialize the expanded array and increase the memory bandwidth requirements.

```
!$OMP PARALLEL SHARED (NUM)
!$OMP SINGLE
      NUM = OMP_GET_NUM_THREADS()
!$OMP END SINGLE NOWAIT
!$OMP END PARALLEL
      IPIECE = (N + (NUM-1)) / NUM
!$OMP PARALLEL DO PRIVATE(K,J,I)
```

```
      DO K = 1, NUM
        DO J = K*IPIECE - IPIECE + 1, MIN(N, K*IPIECE)
           DO I = 1, M
              PARTIAL_A(I,K) = PARTIAL_A(I,K) + B(J)*C(I,J)
           END DO
        END DO
      END DO
!$OMP PARALLEL DO PRIVATE (I,K)
      DO I = 1, M
        DO K = 1, NUM
           A(I) = A(I) + PARTIAL_A(I,K)
        END DO
      END DO
```

You must analyze the various possible optimizations to find the combination that is right for the particular job.

4.12.3 Load Balancing

When the Fortran compiler divides a loop into pieces, by default it uses the simple method of separating the iterations into contiguous blocks of equal size for each process. It can happen that some iterations take significantly longer to complete than other iterations. At the end of a parallel region, the program waits for all processes to complete their tasks. If the work is not divided evenly, time is wasted waiting for the slowest process to finish.

Consider the following code:

```
DO I = 1, N
     DO J = 1, I
        A(J, I) = A(J, I) + B(J)*C(I)
     END DO
END DO
```

The previous code segment can be parallelized on the `I` loop. Because the inner loop goes from 1 to `I`, the first block of iterations of the outer loop will end long before the last block of iterations of the outer loop.

In this example, this is easy to see and predictable, so you can change the program:

```
!$OMP PARALLEL SHARED (NUM)
!$OMP SINGLE
      NUM_THREADS = OMP_GET_NUM_THREADS()
```

```
!$OMP END SINGLE NOWAIT
!$OMP END PARALLEL
!$OMP PARALLEL DO PRIVATE(I, J, K)
      DO K = 1, NUM_THREADS
        DO I = K, N, NUM_THREADS
           DO J = 1, I
              A(J, I) = A(J, I) + B(J)*C(I)
           END DO
        END DO
      END DO
```

In this rewritten version, instead of breaking up the `I` loop into contiguous blocks, break it into interleaved blocks. Thus, each execution thread receives some small values of `I` and some large values of `I`, giving a better balance of work between the threads. Interleaving usually, but not always, cures a load balancing problem.

You can use the `SCHEDULE` clause to automatically perform this desirable transformation, as in this example:

```
!$OMP PARALLEL DO PRIVATE(I,J), SCHEDULE(STATIC,1)
         DO I = 1, N
            DO J = 1, I
               A (J,I) = A(J,I) + B(J)*C(J)
            END DO
         END DO
```

The previous code has the same meaning as the rewritten form above.

Interleaving can cause poor cache performance because the array is no longer stepped through at stride 1. You can improve performance somewhat by using a chunk size larger than 1. Usually 4 or 8 is a good value for *int_expr*. Each small chunk will have stride 1 to improve cache performance, while the chunks are interleaved to improve load balancing.

The way that iterations are assigned to processes is known as *scheduling*. Interleaving is one possible schedule. Both interleaving and the simple scheduling methods are examples of *fixed* schedules; the iterations are assigned to processes by a single decision made when the loop is entered. For more complex loops, it may be desirable to use `DYNAMIC` or `GUIDED` schedules.

Comparing the output from SpeedShop allows you to see how well the load is being balanced so you can compare the different methods of dividing the load. For more information on SpeedShop, see the `ssrun`(1) man page.

Even when the load is perfectly balanced, iterations may still take varying amounts of time to finish because of random factors. One process may take a page fault, another may be interrupted to let a different program run, and so on. Because of these unpredictable events, the time spent waiting for all processes to complete can be several hundred cycles, even with near perfect balance.

Parallel Processing on Origin Series Systems [5]

This chapter describes directives that may be useful to you when developing programs for parallel processing on Origin Series systems. The techniques described in this chapter use directives from the OpenMP Fortran API standard and directives that are Silicon Graphics extensions to the standard.

Note: The directives and clauses that are part of the OpenMP Fortran API have the `!$OMP` prefix. The extension directives have the `!$SGI` prefix.

The multiprocessing features described in this chapter require support from the MP run-time library. IRIX operating system versions 6.3 and later include this library. If you need to access these features on a machine running a different IRIX version, contact your sales representative.

For information on environment variables that can control run-time features, see the `pe_environ`(5) man page.

5.1 Performance Tuning on Origin Series Systems

Origin series systems provide cache-coherent, shared memory in the hardware. Memory is physically distributed across processors. Processors can read data only from the primary cache. If the required data is not present in the primary cache, a *cache miss* is said to have occured. Therefore, references to locations in the remote memory of another processor take substantially longer to complete than references to locations in local memory. Cache misses adversely affect program performance.

Figure 2 shows a simplified version of the Origin series memory hierarchy.

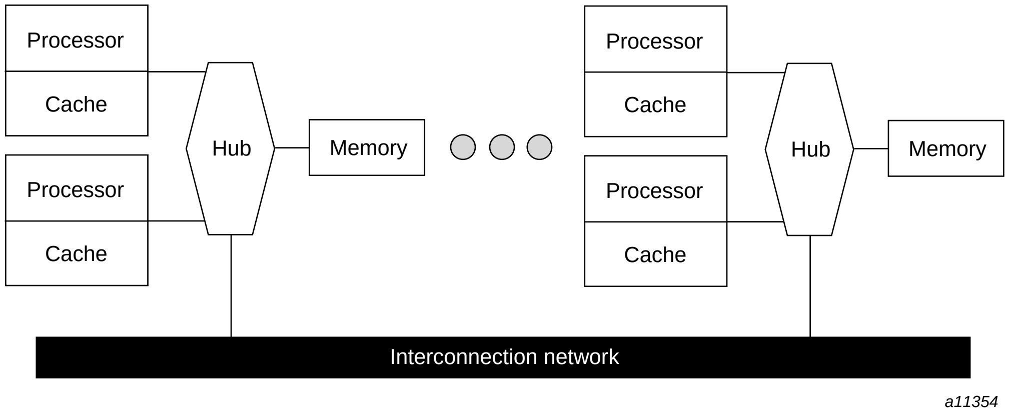

Figure 2. Origin series memory hierarchy

5.1.1 Improving Program Performance

To obtain good performance in parallel programs it is important to schedule computation and to distribute the data across the underlying processors and memory modules, ensuring that most cache misses are satisfied from local rather than from remote memory. The primary goal of programming support is to enable user control over data placement and user control over computation scheduling.

Cache behavior is the largest single factor affecting performance, and programs with infrequent cache misses usually have little need for explicit data placement. These programs write data to memory and reuse it as many times as possible before overwriting it. You can use `perfex`(1) to find information on your program's cache misses.

In programs with many cache misses, if the misses correspond to true data communication between processors, data placement is unlikely to help. In these cases, it may be necessary to redesign your program to reduce interprocessor communication. When redesigning your program to reduce interprocessor communication, keep the following in mind:

- Make sure the data needed by a processor is at least local to the processor's memory.
- Make sure that each processor is working independently and not relying on the changing data of other processors.
- Minimize cache misses.

If the misses are to data that is referenced primarily by a single processor, then data placement may be able to convert remote references to local references, thereby reducing the latency of the miss. The possible methods for data placement are *automatic page migration* or *explicit data distribution*, either regular or reshaped, described in detail in Section 5.3.1, page 159, and Section 5.3.2, page 160. The differences between these methods are shown in Figure 3, page 146. Some criteria for choosing between these methods are discussed in Section 5.1.2, page 147.

Automatic page migration requires no user intervention and is based on the run-time cache miss behavior of the program. It can, therefore, adjust to dynamic changes in the reference patterns. However, page migration is very conservative, and the system may be slow to react to changes in the reference patterns. It is also limited to performing page-level data allocation.

> **Note:** On most systems, page migration is disabled by default. When enabled, page migration can affect other codes running on the system. To determine whether page migration is enabled, contact your system administrator or examine the output from the `sn -v` command. For more information on this command, see the `sn`(1) man page.

Regular data distribution (performing only page-level placement of the array) is also limited to page-level allocation, but is useful when the page migration heuristics are slow and the desired distribution is known to the programmer.

Finally, reshaped data distribution changes the layout of the array. This overcomes the page-level allocation constraints, but it is useful only if a data structure has the same (static) distribution for the duration of the program. Given these differences, it may be necessary to use each of these methods for different data structures in the same program.

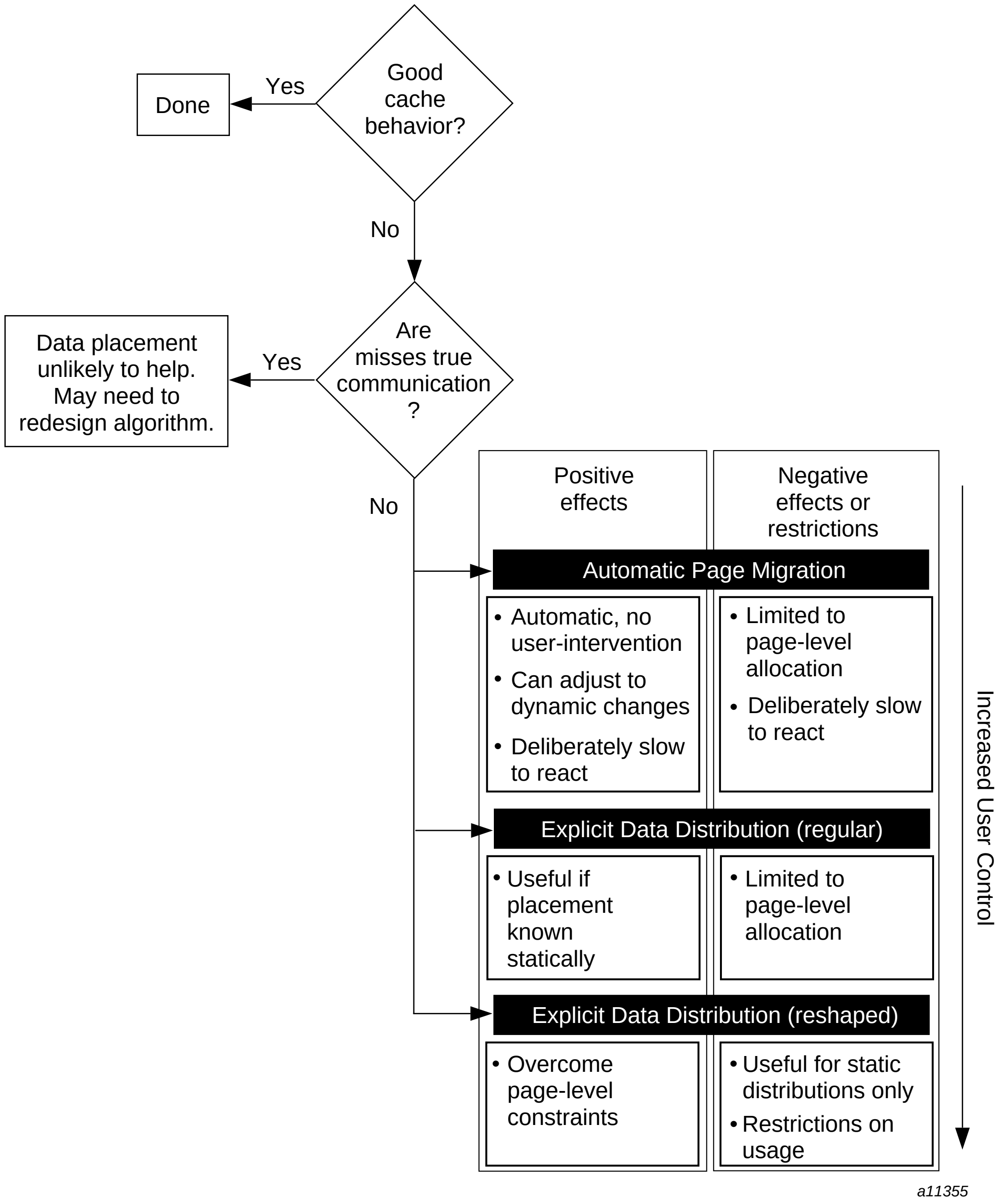
Good cache behavior?
Yes
Done
No
Are misses true communication ?
Yes
Data placement unlikely to help. May need to redesign algorithm.
No
Positive effects
Negative effects or restrictions
Automatic Page Migration
• Automatic, no user-intervention
• Can adjust to dynamic changes
• Deliberately slow to react
• Limited to page-level allocation
• Deliberately slow to react
Explicit Data Distribution (regular)
• Useful if placement known statically
• Limited to page-level allocation
Explicit Data Distribution (reshaped)
• Overcome page-level constraints
• Useful for static distributions only
• Restrictions on usage
Increased User Control
a11355

Figure 3. Cache behavior and solutions

5.1.2 Choosing a Tuning Method

For a given data structure in the program, you can choose between the automatic page migration method or the data distribution method. Your choice will be based on the following criteria:

- If the program repeatedly references the data structure and benefits from reuse in the cache, data placement is not needed.
- If the program incurs a large number of cache misses on the data structure, then you should identify the desired distribution in the array dimensions (such as `BLOCK` or `CYCLIC`) based on the desired parallelism in the program.

  The following example suggests a `A(BLOCK, *)` distribution:

  ```
  !$OMP PARALLEL DO
        DO I = 2, N
          DO J = 2, N
            A(I,J) = 3*I + 4*J + A(I, J-1)
          END DO
        END DO
  ```

  However, the following example suggests a `A(*, BLOCK)` distribution:

  ```
        DO I = 2, N
  !$OMP PARALLEL DO
          DO J = 2, N
            A(I,J) = 3*I + 4*J + A(I-1, J)
          END DO
        END DO
  ```

After identifying the desired distribution, you can select either *regular* or *reshaped* distribution based on the size of an individual processor's portion of the distributed array. Regular distribution is useful only if each processor's portion is substantially larger than the page size in the underlying system (16 KB on the Origin series systems). Otherwise, regular distribution is probably not useful, and you should use the `!$SGI DISTRIBUTE_RESHAPE` directive, which changes the layout of the array to overcome page-level constraints.

For example, consider the following code:

```
      REAL(KIND=8) A(M, N)
!$SGI DISTRIBUTE A(BLOCK, *)
```

In the preceding example, the size of each processor's portion is approximately m/P elements ($8 \times (m/P)$ bytes), where P is the number of processors. If m is

1,000,000, each processor's portion is likely to exceed a page and regular distribution is sufficient. However, if *m* is 10,000, the `!$SGI DISTRIBUTE_RESHAPE` directive is required to obtain the desired distribution.

In contrast, consider the following distribution:

```
!$SGI DISTRIBUTE A(*, BLOCK)
```

In the preceding example, the size of each processor's portion is approximately $(m \times n)/P$ elements ($8 \times (m \times n)/P$ bytes). Therefore, if *n* is 100, for example, regular distribution may be sufficient even if *m* is only 10,000.

Distributing the outer dimensions of an array increases the size of an individual processor's portion (favoring regular distribution), but distributing the inner dimensions is more likely to require reshaped distribution.

The IRIX operating system on Origin series systems follows a default *first-touch* page-allocation policy. This means that each page is allocated from the local memory of the processor that incurs a page-fault on that page. Therefore, in programs where the array is initialized and is consequently first referenced in parallel, even a regular distribution directive may not be necessary, because the underlying pages are allocated from the desired memory location automatically due to the first-touch policy.

OpenMP: The OpenMP Fortran API does not describe the `BLOCK` or `CYCLIC` data distributions. These are Silicon Graphics extensions.

5.2 Directives for Performance Tuning

The MIPSpro 7 Fortran 90 compiler supports directives for performance tuning on Origin series systems. These directives are extensions to the OpenMP Fortran API. You must be licensed for the MIPSpro Automatic Parallelization Option in order for these directives to be recognized. In addition, the `-mp` or `-pfa` options must be in effect during compilation.

The directives supported are as follows:

- `!$SGI DISTRIBUTE`
- `!$SGI DISTRIBUTE_RESHAPE`
- `!$OMP PARALLEL DO`
- `!$SGI DYNAMIC`

- `!$SGI PAGE_PLACE`
- `!$SGI REDISTRIBUTE`

  Note: The functionality of the preceding directives is the same as that provided in MIPSpro 7 Fortran 90 releases 7.2 and earlier. Only the prefix has changed. Beginning with MIPSpro 7 Fortran 90 release 7.2.1, the `!$` prefix is outmoded.

The MIPSpro 7 Fortran 90 compiler supports several clauses to the preceding directives that are extensions to the OpenMP Fortran API. These clauses can be used with the preceding directives and with the standard directives described by OpenMP. To preserve portability, the clauses must be preceded by a `!$SGI+` prefix and must appear on a separate line, as follows:

```
directive

!$SGI+clause

. . .
```

directive	Specify any OpenMP Fortran API directive or any Silicon Graphics parallel processing directive. The OpenMP directives are described in Chapter 4, page 89, and the Silicon Graphics parallel processing directives are described in this chapter.
clause	Specify any of clauses described in this chapter. There cannot be any intervening spaces between the plus sign (+) and the name of the clause.

The following code uses the Silicon Graphics `NEST` clause with the OpenMP Fortran API `DO` directive:

```
!$OMP PARALLEL
!$OMP DO
!$SGI+NEST (I,J)

      DO I = 1,10
         DO J = 1,10
            BLAH, BLAH
         ENDDO
      ENDDO

!$OMP ENDDO
```

```
!$OMP END PARALLEL
      END
```

The `!$OMP PARALLEL DO` directive is described in Section 4.5.1, page 102. The following sections describe the syntax of the Silicon Graphics directives and clauses that are extensions to the OpenMP Fortran API.

5.2.1 Determining the Data Distribution for an Array: `!$SGI DISTRIBUTE`, `!$SGI DISTRIBUTE_RESHAPE`, and `!$SGI REDISTRIBUTE`

The `!$SGI DISTRIBUTE` directive determines the data distribution for an array. The `!$SGI REDISTRIBUTE` directive dynamically redistributes an array. The `!$SGI DISTRIBUTE_RESHAPE` directive performs data distribution with reshaping.

The formats of these directives are as follows:

```
!$SGI DISTRIBUTE array (dist1,dist2)
  [ONTO (target1, target2 [, targetN] ...)]

!$SGI DISTRIBUTE_RESHAPE array (dist1,dist2)
  [ONTO (target1, target2 [, targetN] ...)]

!$SGI REDISTRIBUTE array (dist1,dist2)
  [ONTO (target1, target2 [, targetN] ...)]
```

array	Specify the name of an array.
dist	Specify the type of distribution for each dimension of the named array. The number of *dist* arguments specified must be equal to the number of array dimensions. *dist* can be one of the following:

- `BLOCK`. Indicates that `BLOCK` distribution should be used.
- `CYCLIC` [(*expr*)]. If *expr* is not specified, a chunk size of 1 is assumed.

  For performance reasons, use constants rather than *expr* when the value of *expr* is known to be a compile-time constant.
- An asterisk (`*`). Indicates that the dimension is not distributed.

target	Specify the target processor topology. This argument to the `ONTO` clause specifies how to partition the processors across the

distributed dimensions. There must be one *target* argument specified for each BLOCK and CYCLIC distribution specified.

The Silicon Graphics data distribution directives and the !$OMP PARALLEL DO directive have an optional ONTO clause. The ONTO clause allows you to specify the processor topology when two (or more) dimensions of processors are required.

The following example array is distributed in two dimensions, so you can use the ONTO clause to specify how to partition the processors across the distributed dimensions:

```
! ASSIGN PROCESSOR IN THE RATIO 1:2 TO THE TWO
! DIMENSIONS OF ARRAY A
      REAL(KIND=8) A(100, 200)
!$SGI DISTRIBUTE A (BLOCK, BLOCK) ONTO (1, 2)
```

You can supply a !$SGI DISTRIBUTE directive on a dummy argument, thereby specifying the distribution on the incoming actual argument. If different calls to the subroutine have arguments with different distributions, you can omit the !$SGI DISTRIBUTE directive on the dummy argument. Data affinity loops in that subroutine are automatically implemented through a run-time lookup of the distribution. This is allowed only for regular data distribution. For reshaped array parameters, the distribution must be fully specified on the formal parameter.

For more information on using the data distribution directives, see Section 5.3, page 158.

OpenMP: The OpenMP Fortran API does not describe the BLOCK, *, or CYCLIC distribution; or the ONTO clause.

5.2.2 Specifying a Parallel Region: !$OMP PARALLEL DO

The !$OMP PARALLEL DO directive is part of the OpenMP Fortran API. It accepts the Silicon Graphics AFFINITY and NEST clauses as extensions, however.

The following sections describe the AFFINITY and NEST clauses. For information on the !$OMP PARALLEL DO directive, see Section 4.5.1, page 102.

5.2.2.1 `AFFINITY` Clause

Affinity scheduling controls the mapping of iterations of a parallel loop for execution onto the underlying threads. The `!$OMP PARALLEL DO` directive with the `AFFINITY` clause must immediately precede the loop to which it applies, and it is in effect only for that loop.

An `AFFINITY` clause, if supplied, overrides an OpenMP `SCHEDULE` clause.

There are two type of affinity scheduling: data affinity and thread affinity.

An `AFFINITY` clause on an `!$OMP PARALLEL DO` directive has the following format:

```
!$OMP PARALLEL DO

!$SGI+AFFINITY(do_variable) = DATA(array_element)
```

```
!$OMP PARALLEL DO

!$SGI+AFFINITY(do_variable) = THREAD(expr)
```

do_variable	Specify one or more `DO` loop identifiers, separated by commas.
array_element	Enter an array element.
expr	Specify a thread number. *do_variable* is executed on the thread number specified, modulo the number of threads.

Because the threads may need to evaluate *expr* in each iteration of the loop, the variables used in the *expr* (other than the *do_variable*) must be declared `SHARED` and must not be modified during the execution of the loop. Violating these rules can lead to incorrect results. For information on declaring shared variables, see Section 4.7.2.2, page 114.

If the *expr* does not depend on the `DO` variable, all iterations execute on the same thread and do not benefit from parallel execution.

When `-O3` is in effect, loops that reference reshaped arrays default to data affinity scheduling for the most frequently accessed reshaped array in the loop (chosen by the compiler). To override this behavior, you can explicitly specify the `SCHEDULE` clause on the `!$SGI PARALLEL DO` directive.

Data affinity for loops with nonunit stride can sometimes result in nonlinear affinity expressions. In such situations the compiler issues a warning, ignores the affinity clause, and defaults to STATIC scheduling.

Example 1. The following code shows an example of data affinity:

```
!$SGI DISTRIBUTE A(BLOCK)
!$OMP PARALLEL DO
!$SGI+AFFINITY(I) = DATA(A(X*I+Y))
      DO I = 1, N
        A(X*I+Y) = 0
      END DO
```

The multiplier for A and the constant term B must both be literal constants, with A greater than zero.

This example distributes the iterations of the parallel loop to match the data distribution specified for array A, such that iteration I is executed on the processor that owns element A(X*I+Y) based on the distribution for A. The iterations are scheduled based on the specified distribution, and are not affected by the actual underlying data distribution, which may, for example, differ at page boundaries.

Example 2. In case of a multidimensional array, affinity is provided for the dimension that contains the loop index variable. The loop index variable cannot appear in more than one dimension in an AFFINITY clause. In the following example, the loop is scheduled based on the block distribution of the first dimension:

```
!$SGI DISTRIBUTE A (BLOCK, CYCLIC(1))
!$OMP PARALLEL DO
!$SGI+AFFINITY(I) = DATA(A(I+3, J))
      DO I = 1, N
        DO J = 1, N
          A(I+3, J) = A(I+3,J-1)
        END DO
      END DO
```

Example 3. The following directive executes iteration `I` on the thread number given by the user-supplied expression (modulo the number of threads):

```
!$OMP PARALLEL DO
!$SGI+AFFINITY (I) = THREAD( expr)
```

OpenMP: The OpenMP Fortran API does not describe the `AFFINITY` clause.

5.2.2.2 `NEST` Clause

The `NEST` clause on the `!$OMP PARALLEL DO` directive allows you to exploit nested concurrency in a limited manner. Although true nested parallelism is not supported, you can exploit parallelism across iterations of a perfectly nested loop nest.

The `NEST` clause to the `!$OMP PARALLEL DO` directive has the following format:

```
!$OMP PARALLEL DO

!$SGI+NEST ( do_variable , do_variable [, do_variable ] ...)
  [ONTO (target1, target2 [, targetn] ...)]
```

index	Specify a *do_variable* name that identifies a subsequent loop. At least two *do_variable* names must be specified. The loops identified must be perfectly nested.
target	Specify the target processor topology. The `ONTO` clause allows you to specify the processor topology when two (or more) dimensions of processors are required. This argument specifies

how to partition the processors across the distributed dimensions. *target* can be either an integer expression or an asterisk (*).

Example 1. In a nested `!$OMP PARALLEL DO` with two or more nested loops, you can use the `ONTO` clause to specify the partitioning of processors across the multiple parallel loops, as follows:

```
! USE 2 PROCESSORS IN THE OUTER LOOP,
! AND THE REMAINING IN THE INNER LOOP
!$OMP PARALLEL DO
!$SGI+NEST(I, J) ONTO(2, *)
      DO I = 1, N
        DO J = 1, M
          A(J,I) = ...
        END DO
      END DO
```

Example 2. The following directive specifies that the entire set of iterations across both loops can be executed concurrently:

```
!$OMP PARALLEL DO
!$SGI+NEST(I, J)
      DO I = 1, N
        DO J = 1, M
          A(I,J) = 0
          END DO
      END DO
```

It is restricted, however, in that loops `I` and `J` must be perfectly nested. No code is allowed between either the `DO I ...` and `DO J ...` statements or between the `END DO` statements.

You can combine a nested `!$OMP PARALLEL DO` directive with an `AFFINITY` clause or with a `SCHEDULE` clause specified as `STATIC`; `STATIC` scheduling is the default except when accessing reshaped arrays. `DYNAMIC`, `RUNTIME`, and `GUIDED` scheduling are not supported.

For more information on the `AFFINITY` clause, see Section 5.2.2.1, page 152. For more information on the `SCHEDULE` clause see Section 4.4.1, page 96.

The following code uses an `AFFINITY` clause:

```
!$OMP PARALLEL DO
!$SGI+NEST(I, J) AFFINITY(I,J) = DATA(A(I,J))
      DO I = 2, N-1
        DO J = 2, M-1
      A(I,J) = A(I,J) + I*J
    END DO
  END DO
```

OpenMP: The OpenMP Fortran API does not describe the NEST clause.

5.2.3 Requesting Dynamic Distribution for an Array: !$SGI DYNAMIC

The !$SGI DYNAMIC directive informs the compiler that a particular array can be dynamically redistributed. This directive is required for arrays in procedures that contain !$OMP PARALLEL DO loops with data affinity for arrays in the loops.

By default, the compiler assumes that a distributed array is not dynamically redistributed, and it directly schedules a parallel loop for the specified data affinity. In contrast, a redistributed array can have multiple possible distributions, and data affinity for a redistributed array must be implemented in the run-time system based on the particular distribution.

However, the compiler does not know if an array is redistributed because the array may be redistributed in another procedure or in another file. Therefore, you must explicitly specify the !$SGI DYNAMIC declaration for redistributed arrays. The !$SGI DYNAMIC directive implements data affinity for that array at run time rather than at compile time. If you know an array has a specified distribution throughout the duration of a procedure, you do not have to supply the !$SGI DYNAMIC directive. The result is more efficient compile time affinity scheduling. This directive is required only in those procedures that contain a !$OMP PARALLEL DO loop with data affinity for that array. This tells the compiler that the array can be dynamically redistributed. Data affinity for such arrays is implemented through a run-time lookup.

The format of this directive is as follows:

```
!$SGI DYNAMIC (array)
```

array	Specify the name of an array.

The run-time lookup incurs some extra overhead compared to a direct compile-time implementation. Because the compiler assumes that a distributed array is not redistributed at run time, the distribution is known at compile time, and data affinity for the array can be implemented directly by the compiler. In contrast, because a redistributed array can have multiple possible distributions at run time, data affinity for a redistributed array is implemented in the run-time system based on the distribution at run time, incurring extra run-time overhead.

You can avoid this overhead when a procedure contains data affinity for a redistributed array and the distribution of the array for the entire duration of that procedure is known. In this situation, you can supply the `!$SGI DISTRIBUTE` directive with the particular distribution and omit the `!$SGI DYNAMIC` directive.

Because reshaped arrays cannot be dynamically redistributed, this is an issue only for regular data distribution.

5.2.4 Designating Memory: `!$SGI PAGE_PLACE`

The `!$SGI PAGE_PLACE` directive allows you to explicitly place data structures in the physical memory of a particular processor. This directive is useful when dealing with irregular data structures such as pointers and sparse-matrix arrays.

The format of this directive is as follows:

```
!$SGI PAGE_PLACE (object, size, threadnum)
```

object	Specify the name of the object.
size	Specify the size of *object*, in bytes.
threadnum	Specify the processor number upon which *object* is to be placed.

This directive causes all the pages spanned by the virtual address range (*address* to *address+size*) to be allocated from the local memory of processor number *threadnum*. It is an executable statement; therefore, you can use it to place either statically or dynamically allocated data. This directive is only a performance hint; it does not allocate memory, and it has no effect on the virtual address space of the program.

An example of this directive is as follows:

```
      REAL(KIND=8) A(100)
!$SGI PAGE_PLACE (A, 800, 3)
```

5.3 Using the Data Distribution Directives

The data distribution directives, `!$SGI DISTRIBUTE`, `!$SGI REDISTRIBUTE`, and `!$SGI DISTRIBUTE_RESHAPE`, allow you to specify distributions for array data structures. For irregular data structures, the directives can explicitly place data directly on a specific processor.

The `!$SGI DISTRIBUTE`, `!$SGI DYNAMIC`, and `!$SGI DISTRIBUTE_RESHAPE` directives are declarations that must be specified in the declaration part of the program, along with the array declaration. The `!$SGI REDISTRIBUTE` directive is an executable statement and can appear in any executable portion of the program.

You can specify a data distribution directive for any local, global, or common block array. Each dimension of a multidimensional array can be independently distributed. The possible distribution types for an array dimension are `BLOCK`, `CYCLIC[(`*expr*`)]`, and `*`, as follows:

- As shown in Figure 4, page 158, a `BLOCK` distribution is one that partitions the elements of the dimension of size *N* into *P* blocks (one per processor), with each block of size *B* `= ceiling (`*N*/*P*).

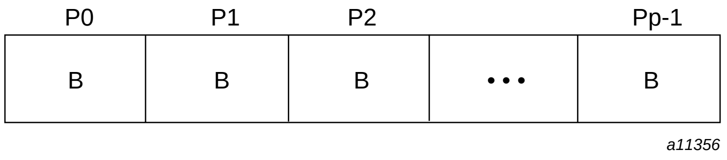

Figure 4. Block distribution

- A `CYCLIC` distribution can include an *expr* to indicate the chunk size. A chunk size that is either greater than 1 or is determined at run time is sometimes also called `BLOCK-CYCLIC`.
- The `*` distribution indicates that the array is not distributed.

As shown in Figure 5, page 159, a `CYCLIC[(`*expr*`)]` distribution partitions the elements of the dimension into pieces of size *expr* each and distributes them sequentially across the processors:

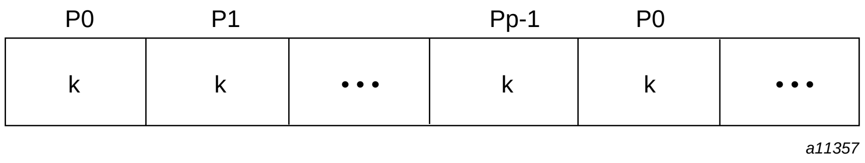

Figure 5. Cyclic distribution

A distributed array is distributed across all of the processors being used in that particular execution of the program, as determined by the `OMP_NUM_THREADS` environment variable. If a distributed array is distributed in more than one dimension, then by default the processors are apportioned as equally as possible across each distributed dimension. For example, if an array has two distributed dimensions, then an execution with 16 processors assigns 4 processors to each dimension (4 x 4=16), whereas an execution with 8 processors assigns 4 processors to the first dimension and 2 processors to the second dimension. You can override this default and explicitly control the number of processors in each dimension using the `ONTO` clause with a data distribution directive.

5.3.1 Regular Data Distribution

The `DISTRIBUTE` and `REDISTRIBUTE` data distribution directives achieve the desired distribution by influencing the mapping of virtual addresses to physical pages without affecting the layout of the data structure. Because the granularity of data allocation is a physical page (at least 16 KB), the achieved distribution is limited by the underlying page granularity. However, the advantages are that regular data distribution directives can be added to an existing program without any restrictions, and they can be used for affinity scheduling.

For example, the following directive dynamically redistributes array `A`:

```
!$SGI REDISTRIBUTE A (BLOCK, CYCLIC(K))
```

The `!$SGI REDISTRIBUTE` directive is an executable statement that changes the distribution permanently (or until another `!$SGI REDISTRIBUTE` statement). It also affects subsequent affinity scheduling.

The `!$SGI DYNAMIC` directive specifies that the named array is redistributed in the program, and is useful in controlling affinity scheduling for dynamically redistributed arrays.

For more information on the `!$SGI REDISTRIBUTE` and `!$SGI DYNAMIC` directives, see Section 5.2.1, page 150, and Section 5.2.3, page 156.

5.3.2 Data Distribution with Reshaping

Similar to regular data distribution, the RESHAPE directive specifies the desired distribution of an array. In addition, however, the !$SGI DISTRIBUTE_RESHAPE directive declares that the program makes no assumptions about the storage layout of that array. The compiler performs aggressive optimizations for reshaped arrays that may violate standard Fortran layout assumptions, but it guarantees the desired data distribution for that array.

As shown in the following example, the !$SGI DISTRIBUTE_RESHAPE directive accepts the same distributions as the regular data distribution directive:

```
!$SGI DISTRIBUTE_RESHAPE (BLOCK, CYCLIC(1))
```

5.3.2.1 Restrictions on Reshaped Arrays

Because the !$SGI DISTRIBUTE_RESHAPE directive specifies that the program does not depend on the storage layout of the reshaped array, restrictions on the arrays that can be reshaped include the following:

- Deferred-shape arrays (pointers, assumed-shape arrays, dummy arguments, and allocatable arrays) cannot be reshaped.
- The distribution of a reshaped array cannot be changed dynamically (that is, there is no REDISTRIBUTE_RESHAPE directive).
- Initialized data cannot be reshaped.
- Arrays that are explicitly allocated through the alloca(3C) or MALLOC(3F) routines and accessed through Cray pointers cannot be reshaped.
- An array that is equivalenced to another array cannot be reshaped.
- I/O for a reshaped array cannot be mixed with namelist I/O or a function call in the same I/O statement.
- A common block that contains a reshaped array cannot be declared THREADPRIVATE. For more information on the THREADPRIVATE directive, see Section 4.7.1, page 111.

Caution: A common block containing a reshaped array cannot be loaded with the -Wl,-Xlocal option. This user error is not detected by the compiler or loader.

There are two possible outcomes if a reshaped array is passed as an actual parameter to a subroutine:

- The array is passed in its entirety; that is, `CALL FUNC(A)` passes the entire array `A`, whereas `CALL FUNC(A(I,J))` passes a portion of `A`. The compiler automatically clones a copy of the called subroutine and compiles it for the incoming distribution. The actual arguments and dummy arguments must match in the number of dimensions and the size of each dimension.

  You can restrict a subroutine to accept a particular reshaped distribution on a parameter by specifying a `!$SGI DISTRIBUTE_RESHAPE` directive on the dummy argument within the subroutine. All calls to this subroutine with a mismatched distribution will lead to compile time or load time.

- A portion of the array can be passed as an actual argument, but the callee must access only a single processor's portion. If the callee exceeds a single processor's portion, the results are undefined. You can use the intrinsics described on the `MP`(3F) man page under the heading *Query Intrinsics for Distributed Arrays* to find details about the array distribution.

5.3.2.2 Error Detection for Reshaped Arrays

Most errors in accessing reshaped arrays are detected either at compile time or at load time. These errors include:

- Inconsistencies in reshaped arrays across common blocks (including across files)
- Using the `EQUIVALENCE` statement to declare a reshaped array as equivalent to another array
- Inconsistencies in reshaped distributions on actual and dummy arguments
- Other errors such as disallowed I/O statements involving reshaped arrays, reshaping initialized data, or reshaping dynamically allocated data

Errors such as matching the declared size of an array dimension typically can be caught only at run time. You can use the `-MP:CHECK_RESHAPE=ON` option on the `f90`(1) command to perform these tests at run time. These run-time checks are not generated by default because they incur overhead, but they are useful during program development.

The types of run-time checks performed can detect the following:

- Inconsistencies in array bounds declarations on each actual and dummy argument
- Inconsistencies in declared bounds of a dummy argument that corresponds to a portion of a reshaped actual argument

5.3.2.3 Implementation of Reshaped Arrays

The compiler transforms a reshaped array into a pointer to a *processor array*. The processor array has one element per processor, with the element pointing to the portion of the array local to the corresponding processor.

Figure 6, page 162, shows the effect of a `!$SGI DISTRIBUTE_RESHAPE` directive with a `BLOCK` distribution on a one-dimensional array, as follows:

```
      REAL A(N)
!$SGI DISTRIBUTE_RESHAPE A(BLOCK)
```

N is the size of the array dimension, *P* is the number of processors, and *B* is the block-size on each processor, `CEILING(`*N*`/`*P*`)`.

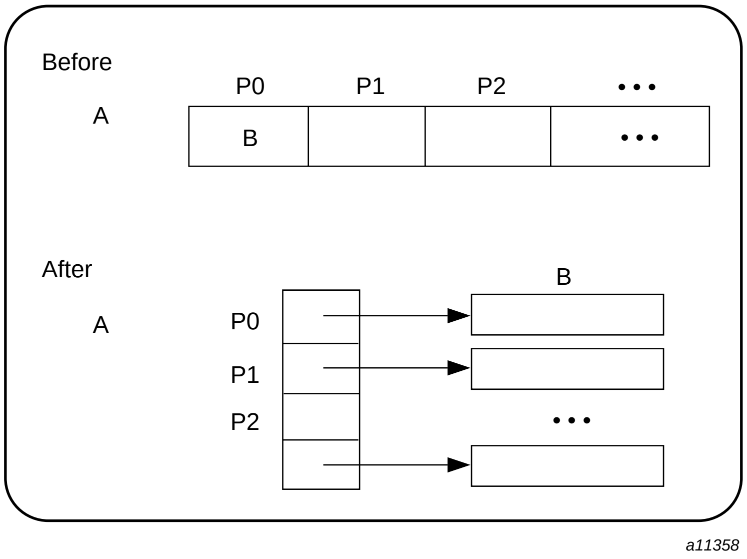

Figure 6. Implementation of the `!$SGI DISTRIBUTE_RESHAPE A(BLOCK)` distribution directive

With this implementation array reference `A(I)` is transformed into the two-dimensional reference `A[I/B][I%B]` (in C syntax with C dimension order), where *B* is the size of each block and given by `CEILING(`*N*`/`*P*`)`. Thus `A[I/B]` points to a processor's local portion of the array, and `A[I/B][I%B]` refers to a specific element within the local processor's portion.

A `CYCLIC` distribution with a chunk size of 1 is implemented as shown in Figure 7, page 163.

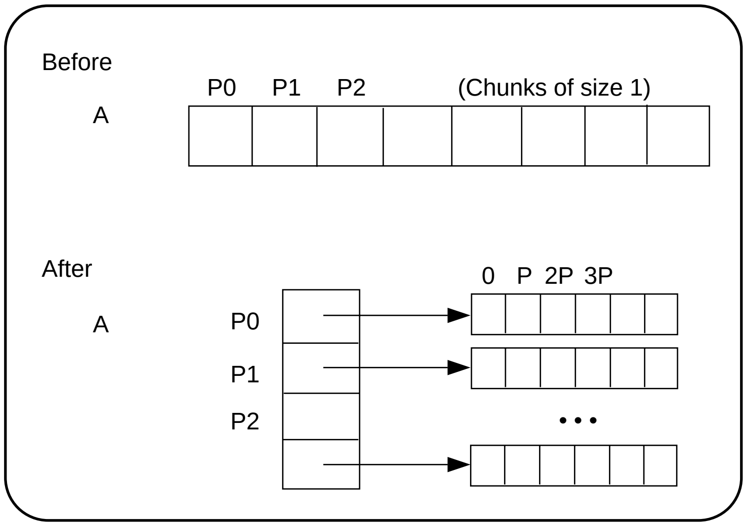

Figure 7. Implementation of the
`!$SGI DISTRIBUTE_RESHAPE A(CYCLIC(1))` distribution directive

An array reference, `A(I)`, is transformed to `A[I%P][I/P]`, where *P* is the number of threads in that distributed dimension.

Finally, a `CYCLIC` distribution with a chunk size that is either a constant greater than 1 or a run-time value (also called `BLOCK-CYCLIC`) is implemented as Figure 8, page 164, shows.

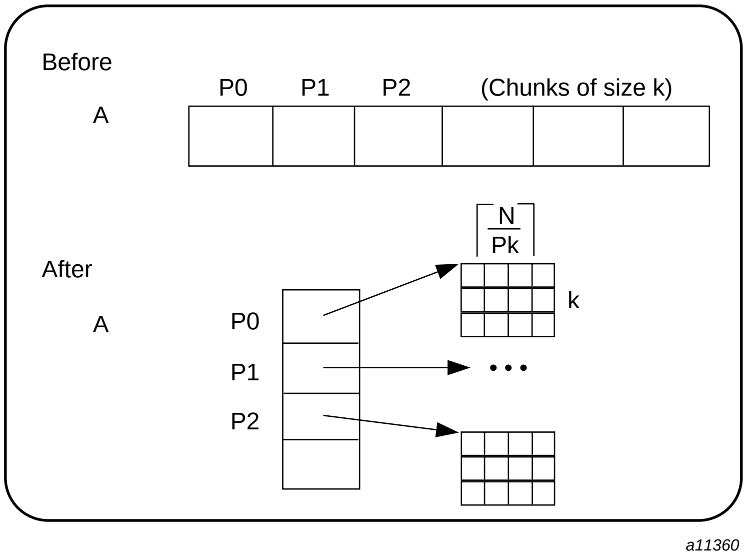

Figure 8. Implementation of the `!$SGI DISTRIBUTE_RESHAPE A(CYCLIC(K))` directive (a `BLOCK-CYCLIC` Distribution)

An array reference, `A(I)`, is transformed to the three-dimensional reference `A[(I/K)%P][I/(PK)][I%K]`, where *P* is the total number of threads in that dimension and *K* is the chunk size.

The compiler tries to optimize these divide/modulo operations out of inner loops through aggressive loop transformations such as blocking and peeling.

5.3.3 Regular versus Reshaped Data Distribution

Regular distributions have an advantage in that they do not impose any restrictions on the distributed arrays and can be freely applied in existing codes. Furthermore, they work well for distributions where page granularity is not a problem. For example, consider a `BLOCK` distribution of the columns of a two-dimensional Fortran array of size `A(R,C)` (column-major layout) and distribution `(*, BLOCK)`. If the size of each processor's portion, `CEILING=(`*C*`/`*P*`)(`*R*`)(`*element_size*`)` is significantly greater than the page size (16 KB on Origin2000 systems), then regular data distribution should be effective in placing the data in the desired fashion.

However, regular data distribution is limited by page-granularity considerations. For instance, consider a `(BLOCK,BLOCK)` distribution of a two-dimensional array in which the size of a column is much smaller than a page. Each physical page is likely to contain data belonging to multiple processors, making the data distribution quite ineffective. However, data distribution may still be useful from the standpoint of affinity scheduling considerations.

Reshaped data distribution addresses the problems of regular distributions by changing the layout of the array in memory to guarantee the desired distribution. However, because the array no longer conforms to standard Fortran storage layout, there are restrictions on the usage of reshaped arrays.

Given both types of data distribution, you can choose between the two based on the characteristics of the particular array in an application.

5.4 Examples

The following sections provide several examples of data distribution and affinity scheduling.

5.4.1 Distributing Columns of a Matrix

Example 1. This example distributes the columns of a matrix in a round-robin fashion across threads. Such a distribution places data effectively only if the size of an individual column exceeds that of a page.

```
      REAL(KIND=8) A(N, N)
! DISTRIBUTE COLUMNS IN CYCLIC FASHION
!$SGI DISTRIBUTE A (*, CYCLIC(1))

! PERFORM GAUSSIAN ELIMINATION ACROSS COLUMNS
! THE AFFINITY CLAUSE DISTRIBUTES THE LOOP ITERATIONS BASED
! ON THE COLUMN DISTRIBUTION OF A
      DO I = 1, N
!$OMP PARALLEL DO
!$SGI+AFFINITY(J) = DATA(A(I,J))
        DO J = I+1, N
!         ... REDUCE COLUMN J BY COLUMN I ...
        END DO
      END DO
```

If the columns are smaller than a page, it may be beneficial to reshape the array. This is specified by using a `!$SGI DISTRIBUTE_RESHAPE` directive in place of the `!$SGI DISTRIBUTE` directive.

In addition to overcoming size constraints as shown in the preceding example, the `!$SGI DISTRIBUTE_RESHAPE` directive is useful when the desired distribution is contrary to the layout of the array.

Example 2. This example uses the `!$SGI DISTRIBUTE_RESHAPE` directive to distribute the rows of a two-dimensional matrix. It shows how to overcome the storage layout constraints to provide the desired distribution.

```
REAL(KIND=8) A(N, N)
! DISTRIBUTE ROWS IN BLOCK FASHION
!$SGI DISTRIBUTE_RESHAPE A (BLOCK, *)
      REAL(KIND=8) SUM(N)
!$SGI DISTRIBUTE SUM(BLOCK)

! PERFORM SUM-REDUCTION ON THE ELEMENTS OF EACH ROW
!$OMP PARALLEL DO PRIVATE (J)
!$SGI+AFFINITY(I) = DATA(A(I,J))
      DO I = 1,N
        DO J = 1,N
          SUM(I) = SUM(I) + A(I,J)
        ENDDO
      ENDDO
```

5.4.2 Using Data Distribution and Data Affinity Scheduling

The following example demonstrates regular data distribution and data affinity. This example, run on a 4-processor Origin2000 server, uses simple block scheduling. Processor 0 calculates the values of the first 25,000 elements of `A`, processor 1 calculates the second 25,000 values of `A`, and so on. Arrays `B` and `C` are initialized using one processor. Therefore, all of the memory pages are touched by the master processor (processor 0) and are placed in processor 0's local memory.

Using data distribution changes the placement of memory pages for arrays `A`, `B`, and `C` to match the data reference pattern.

Without data distribution:

```
      REAL(KIND=8) A(1000000), B(1000000)
      REAL(KIND=8) C(1000000)
      INTEGER I

!$OMP PARALLEL SHARED(A, B, C) PRIVATE(I)
!$OMP DO
      DO I = 1, 1000000
         A(I) = B(I) + C(I)
      END DO
!$OMP END PARALLEL
```

With data distribution:

```
      REAL(KIND=8) A(1000000), B(1000000)
      REAL(KIND=8) C(1000000)
      INTEGER I
!$SGI DISTRIBUTE A(BLOCK), B(BLOCK), C(BLOCK)

!$OMP PARALLEL SHARED(A, B, C) PRIVATE(I)
!$OMP DO
!$SGI+AFFINITY(I) = DATA(A(I))
      DO I = 1, 100000
         A(I) = B(I) + C(I)
      END DO
!$OMP END PARALLEL
```

5.4.3 Argument Passing

The following code shows how a distributed array can be passed as an argument to a subroutine that has a matching declaration for the dummy argument:

```
      REAL(KIND=8) A(M, N)
!$SGI DISTRIBUTE_RESHAPE A (BLOCK, *)
      CALL FOO(A, M, N)
      END

      SUBROUTINE FOO(A, P, Q)
      REAL(KIND=8) A(P, Q)
!$SGI DISTRIBUTE_RESHAPE A (BLOCK, *)
!$OMP PARALLEL DO
!$SGI+AFFINITY(I) = DATA(A(I, J))
        DO I = 1, P
        END DO
      END
```

Because the array is reshaped, the `!$SGI DISTRIBUTE_RESHAPE` directive in the caller and the callee must match exactly. Furthermore, all calls to subroutine `FOO` must pass in an array with the exact same distribution.

If the array was only distributed (but not reshaped) in the preceding example, then subroutine `FOO` could be called from different places with different incoming distributions. In that case, you could omit the distribution directive on the dummy argument, thereby ensuring that any data affinity within the loop is based on the distribution (at run time) of the incoming actual argument, as shown in this example:

```
      REAL(KIND=8) A(M, N), B(P, Q)
      REAL(KIND=8) A (BLOCK, *)
      REAL(KIND=8) B (CYCLIC(1), *)
      CALL FOO(A, M, N)
      CALL FOO(B, P, Q)
! ---------------------------------------------------------
      SUBROUTINE FOO(X, S, T)
      REAL(KIND=8) X(S, T)

!$OMP PARALLEL DO
!$SGI+AFFINITY(I) = DATA(X(I+2, J))
      DO I =
        ...
      END DO
```

5.4.4 Redistributed Arrays

Example 1. The following example shows how an array is redistributed at run time:

```
      SUBROUTINE BAR(X, N)
      REAL(KIND=8) X(N, N)
      ...
!$SGI REDISTRIBUTE X (*, CYCLIC(expr))
      ...
      END
!---------------------------------------------------------
      SUBROUTINE FOO
      REAL(KIND=8) LOCALARRAY(1000, 1000)
!$SGI DISTRIBUTE LOCALARRAY (*, BLOCK)
! THE CALL TO SUBROUTINE BAR MAY REDISTRIBUTE LOCALARRAY
!$SGI DYNAMIC LOCALARRAY
```

```
      ...
      CALL BAR(LOCALARRAY, 100)
! THE DISTRIBUTION FOR THE FOLLOWING DOACROSS
! IS NOT KNOWN STATICALLY
!$OMP PARALLEL DO
!$SGI+AFFINITY(I) = DATA(A(I, J))
      END
```

Example 2. The following example illustrates a situation in which the `!$SGI DYNAMIC` directive can be optimized away. The main routine contains local array `A` that is both distributed and dynamically redistributed. This array is passed as an argument to `FOO` before being redistributed and to `FOO` after being (possibly) redistributed. The incoming distribution for `FOO` is statically known; you can specify a `!$SGI DISTRIBUTE` directive on the dummy argument, thereby obtaining more efficient static scheduling for the `!$OMP PARALLEL DO` directive with data affinity. The subroutine `BAR`, however, can be called with multiple distributions, requiring run-time scheduling of the `!$OMP PARALLEL DO` loop.

```
      PROGRAM MAIN
!$SGI DISTRIBUTE A (BLOCK, *)
!$SGI DYNAMIC A
      CALL FOO(A)
      IF (X .NE. 17) THEN
!$SGI REDISTRIBUTE A (CYCLIC(X), *)
      END IF
      CALL BAR(A)
      END

      SUBROUTINE FOO (A)
!Incoming distribution is known to the user
!$SGI DISTRIBUTE A(BLOCK, *)
!$OMP PARALLEL DO
!$SGI+AFFINITY(I) = DATA(A(I, J))
      ...
      END

      SUBROUTINE BAR(A)
```

```
!Incoming distribution is not known statically
!$SGI DYNAMIC A
!$OMP PARALLEL DO
!$SGI+AFFINITY(I) = DATA(A(I, J))
      ...
      END
```

5.4.5 Irregular Distributions and Thread Affinity

This example consists of a large array that is conceptually partitioned into unequal portions, one for each processor. This array is indexed through index array IDX, which stores the starting index value and the size of each processor's portion.

```
      REAL(KIND=8) A(N)
! IDX ---> INDEX ARRAY CONTAINING START INDEX INTO A (IDX(P, 0))
! AND SIZE (IDX(P, 1)) FOR EACH PROCESSOR
      REAL(KIND=4) IDX (P, 2)
!$SGI PAGE_PLACE (A(IDX(0, 0)), IDX(0, 1)*8, 0)
!$SGI PAGE_PLACE (A(IDX(1, 0)), IDX(1, 1)*8, 1)
!$SGI PAGE_PLACE (A(IDX(2, 0)), IDX(2, 1)*8, 2)
      ...
!$OMP PARALLEL DO
!$SGI+ AFFINITY(I) = THREAD(I)
      DO I = 0, P-1
!     ... PROCESS ELEMENTS ON PROCESSOR I
!     ... A(IDX(I, 0)) TO A(IDX(I,0)+IDX(I,1))
      END DO
```

CF90 Directives [6]

The MIPSpro 7 Fortran 90 compiler, running on IRIX systems, recognizes some of the directives that are supported by the CF90 compiler on UNICOS and UNICOS/mk systems. The directives themselves and the sections in which they are discussed are as follows:

- Section 6.1, page 171, describes using directives.
- Section 6.2, page 173, describes the `BOUNDS` and `NOBOUNDS` directives.
- Section 6.3, page 174, describes the `FREE` and `FIXED` directives.
- Section 6.4, page 174, describes the `ID` directive.
- Section 6.5, page 176, describes the `IGNORE_TKR` directive.
- Section 6.6, page 177, describes the `IVDEP` directive.
- Section 6.7, page 180, describes the `NAME` directive.
- Section 6.8, page 180, describes the `NOINTERCHANGE` directive.
- , describes the `NOSIDEEFFECTS` directive.
- Section 6.10, page 181, describes the `PREFERTASK` directive.
- Section 6.11, page 182, describes the `TASK` and `NOTASK` directives.
- Section 6.12, page 183, describes the `UNROLL` and `NOUNROLL` directives.

6.1 Using Directives

The following sections describe how to use the CF90 directives and the effects they have on IRIX platforms.

For additional general information on using directives, see Section 3.1, page 69.

6.1.1 Directive Continuation

In the following example, an asterisk (*) appears in column 6 to indicate that the second line is a continuation of the preceding line:

```
!DIR$ NA
!DIR$*ME
```

The `FIXED` and `FREE` directives must appear alone on a directive line and cannot be continued.

If you want to specify more than one directive on a line, separate each directive with a comma. Some directives require that you specify one or more arguments; when specifying a directive of this type, no other directive can appear on the line.

Spaces can precede, follow, or be embedded within a directive, regardless of source form.

Do not use source preprocessor (#) directives within multiline compiler directives.

6.1.2 Directive Range and Placement

The range and placement of directives is as follows:

- The `FIXED` and `FREE` directives can appear anywhere in your source code. All other directives must appear within a program unit.
- The `BOUNDS/NOBOUNDS` and `TASK/NOTASK` directives take effect at the point at which they appear in the source code.
- The `ID` and `NOSIDEEFFECTS` directives do not apply to any particular range of code. They add information to the *file*`.o` generated from the input program.
- The following directives apply only to the next loop encountered lexically:
  - `IVDEP`
  - `NOINTERCHANGE`
  - `PREFERTASK`
  - `UNROLL/NOUNROLL`
- The `NAME` and `IGNORE_TKR` directives do not apply to particular ranges of code. They are declarative directives that alter the status of entities in ways that affect compilation.

6.1.3 Interaction of Directives with the `-x` Command Line Option

The `-x` option on the `f90`(1) command line accepts one or more directives as arguments. When your input is compiled, the compiler ignores directives named as arguments to the `-x` option. For example, if you specify `-x mipspro`, all directives are ignored. If you specify `-x` *dirname*, the particular directive named in *dirname* is ignored. For more information on this command line option, see Section 2.74, page 65.

6.2 Check Array Bounds: BOUNDS and NOBOUNDS

Array bounds checking provides a check of most array references at both compile time and run time to ensure that each subscript is within the array's declared size.

The `-C` option on the `f90`(1) command line controls bounds checking for a whole compilation. The `BOUNDS` and `NOBOUNDS` directives toggle the feature on and off within a program unit. Either directive can specify particular arrays or can apply to all arrays. The formats of these directives are as follows:

```
!DIR$ BOUNDS [ array [, array ] ... ]

!DIR$ NOBOUNDS [ array [, array ] ... ]
```

array	The name of an array. The name cannot be a subobject of a derived type. When no array name is specified, the directive applies to all arrays.

`BOUNDS` remains in effect for a given array until the appearance of a `NOBOUNDS` directive that applies to that array, or until the end of the program unit. Bounds checking can be enabled and disabled many times in a single program unit.

Note: To be effective, these directives must follow the declarations for all affected arrays. It is suggested that they be placed at the end of a program unit's specification statements unless they are meant to control particular ranges of code.

The bounds checking feature detects any reference to an array element whose subscript exceeds the array's declared size. For example:

```
      REAL A(10)
!  DETECTED AT COMPILE TIME:
      A(11) = X
!  DETECTED AT RUN TIME IF IFUN(M) EXCEEDS 10:
      A(IFUN(M)) = W
```

The compiler generates a message when it detects an out-of-bounds subscript. If the compiler cannot detect the out-of-bounds subscript (for example, if the subscript includes a function reference), a message is issued for out-of-bound subscripts when your program runs.

Bounds checking increases program run time. If an array's last dimension declarator is `*`, checking is not performed on the last dimension's upper bound. Arrays in formatted `WRITE` and `READ` statements are not checked.

If bounds checking detects an out-of-bounds array reference, a message is issued and the program halts.

6.3 Specify Source Form: `FREE` and `FIXED`

The `FREE` and `FIXED` directives specify whether the source code in the program unit is written in free source form or fixed source form. The `FREE` and `FIXED` directives override the `-fixedform` and `-freeform` options, if specified, on the `f90`(1) command line. For more information on the `-fixedform` and `-freeform` options, see Section 2.21, page 14, and Section 2.24, page 15.

The formats of these directives are as follows:

```
!DIR$ FREE

!DIR$ FIXED
```

These directives apply to the source file in which they appear, and they allow you to switch source forms within a source file.

You can change source form within an `INCLUDE` file. After the `INCLUDE` file has been processed, the source form reverts back to the source form that was being used prior to processing of the `INCLUDE` file.

Note: The source preprocessor does not recognize the `FREE` and `FIXED` directives. These directives must not be specified in a file that is submitted to the source preprocessor.

6.4 Create Identification String: `ID`

The `ID` directive inserts a character string into the *file*`.o` produced for a Fortran source file. The format of this directive is as follows:

```
!DIR$ ID "character_string"
```

character_ string	The character string to be inserted into *file*`.o`. The syntax box shows quotation marks as the *character_string* delimiter, but you can use either apostrophes (`' '`) or quotation marks (`" "`).

The *character_string* can be obtained from *file*`.o` in one of the following ways:

- Method 1. Using the `what`(1) command. To use the `what`(1) command to retrieve the character string, begin the character string with the sentinel characters `@(#)`. For example, assume that `id.f` contains the following source code:

  ```
  !DIR$ ID  "@(#)file.f 01 July 1997"
        PRINT *, 'hello'
        END
  ```

  The next step is to use file `id.o` as the argument to the `what`(1) command, as follows:

  ```
  % what id.o
  % id.o:
  %   file.f 01 July 1997
  ```

  Note that `what`(1) does not include the special sentinel characters in the output.

  In the following example, *character_string* does not begin with the characters `@(#)`. The output shows that `what`(1) does not recognize the string.

  Input file `id2.o` contains the following:

  ```
  !DIR$ ID  'file.f 01 July 1997'
        PRINT *, 'Hello, world'
        END
  ```

  The `what`(1) command generates the following output:

  ```
  % what id2.o
  % id2.o:
  ```

- Method 2. Using the `strings`(1) or `od`(1) command. The following example shows how to obtain output using the `strings`(1) command.

  Input file `id.f` contains the following:

```
!DIR$ ID  "File: id.f   Date: 1 July 1997"

         PRINT *, 'hello'
         END
```

The `strings`(1) command generates the following output:

```
% f90 -c id.o
% strings id.o
File: id.f  Date: 1 July 1997
% od -c id.o
```

... portion of dump deleted

```
0002300  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0  \0
0002320   F   i   l   e   :       i   d   .   f               D   a   t
0002340   e   :       1       J   u   l   y       1   9   9   7 001  \0
0002360  \0  \0  \0  \0 024 003 240 031  \0  \0 203 031  \0  \0 205 005
```

... portion of dump deleted

6.5 Disregard Dummy Argument Type, Kind, and Rank: `IGNORE_TKR`

The `IGNORE_TKR` directive directs the compiler to ignore the type, kind, and rank (TKR) of specified dummy arguments in a procedure interface. For information on Fortran TKR rules, see chapters 4 and 6 of the *Fortran Language Reference Manual, Volume 2*.

The format for this directive is as follows:

```
!DIR$ IGNORE_TKR [ darg_name [ , darg_name ] ... ]
```

darg_name	If specified, indicates the dummy arguments for which TKR rules should be ignored. Dummy arguments for assumed-shape arrays or Fortran pointers cannot be specified. If not specified, TKR rules are ignored for all dummy arguments in the procedure that contains the directive.

The directive causes the compiler to ignore type and kind and rank of the specified dummy arguments when resolving a generic to a specific call. The compiler also ignores type and kind and rank on the specified dummy arguments when checking all the specifics in a generic call for ambiguities.

Example. The following directive instructs the compiler to ignore type, kind, and rank rules for the dummy arguments supplied for the `SHMEM_PUT64`(3) function call:

```
      INTERFACE SHMEM_PUT64
         SUBROUTINE SHMEM_PUT64(targ, src, len, pe)
!DIR$    IGNORE_TYPE targ, src
         INTEGER(KIND=4) len
         INTEGER(KIND=4) pe
         END SUBROUTINE SHMEM_PUT64
      END INTERFACE
```

The preceding code specifies that `targ` and `src` can be any data type, but `len` and `pe` must be `INTEGER(KIND=4)` data.

6.6 Ignore Vector Dependencies: IVDEP

The `IVDEP` directive directs the compiler to perform a more liberal dependency analysis for the purpose of software pipelining and other optimizations. The format of this directive is as follows:

```
!DIR$ IVDEP
```

This directive's effects depend on command line settings. When this directive is in effect, certain dependencies are ignored depending on the state of the following `f90`(1) command line options:

Option	Effect
`-OPT:cray_ivdep=OFF`	Default command line setting. IRIX semantics are used when performing dependency analysis. Loop-carried dependencies in the subsequent loop are ignored between any two array references whenever the location referred to by at least one of the array references varies inside the loop. For more information on this command line option, see Section 2.57.3, page 46.
`-OPT:cray_ivdep=ON`	UNICOS semantics are used when performing dependency analysis. The compiler disregards backward dependencies only.

For more information on this command line option, see Section 2.57.3, page 46.

`-OPT:liberal_ivdep=ON`

All dependencies are disregarded. For more information on this command line option, see Section 2.57.19, page 50.

The `IVDEP` directive applies only to inner loops, and it applies to the first `DO` loop that follows the directive within the same program unit.

Example 1. There are two basic types of dependencies in the loop below: loop-carried and non-loop-carried. A *loop-carried dependency* occurs across iterations of the loop. A *non-loop-carried dependency* occurs within an iteration of the loop.

```
!DIR$ IVDEP
      DO I = 1,N
         A(INDEX(1,I)) = B(I)
         A(INDEX(2,I)) = C(I)
      END DO
```

A loop-carried dependency would occur if `INDEX(1,I)` in some iteration of `I` was equal to `INDEX(1,I+K)` in some other iteration of `I`. A non-loop-carried dependency would occur if `INDEX(1,I)` was equal to `INDEX(2,I)` in any iteration of `I`.

Example 2. The following loop is executed with default command line options:

```
!DIR$ IVDEP
      DO I = 1,N
         A(B(K)) = A(C(K)) + D(I)
      END DO
```

Neither the reference to `A(B(K))` nor to `A(C(K))` vary inside the loop, so the `IVDEP` directive does not break the dependence.

Example 3. The following loop is executed with default command line options:

```
!DIR$ IVDEP
      DO I = 1,N
         A(I) = A(I-1) + 3.0
      END DO
```

The `IVDEP` directive breaks the dependence, but the compiler issues a message indicating that an obvious dependence is being broken.

Example 4. The following loop is executed with default command line options, and the `IVDEP` directive breaks the dependence:

```
!DIR$ IVDEP
      DO I = 1,N
         A(B(I)) = A(B(I)) + 3.0
      END DO
```

Example 5. The following loop is executed with default command line options, and the `IVDEP` directive does not break the dependence on `A(I)` because the dependence is non-loop-carried:

```
!DIR$ IVDEP
      DO I = 1,N
         A(I) = B(I)
         C(I) = A(I) + 3.0
      END DO
```

Example 6. The following loop is executed with `-OPT:cray_ivdep=ON` in effect:

```
!DIR$ IVDEP
      DO I = 1,N
         A(I) = A(I-1) + 3.0
      END DO
```

UNICOS semantics are used, and the `IVDEP` directive breaks all lexically backward dependencies. When the loop is executed, however, the compiler issues a message indicating that it is breaking an obvious dependence.

Example 7. When the following loop is executed, the `IVDEP` directive does not break the dependence. This is because the dependence is from the load to the store, and the load comes lexically before the store. Assume that the code fragment in this example was compiled with `-OPT:cray_ivdep=ON`.

```
!DIR$ IVDEP
      DO I = 1,N
         A(I) = A(I+1) + 3.0
      END DO
```

To break all dependencies, specify `-OPT:liberal_ivdep=ON`. Both `-OPT:cray_ivdep` and `-OPT:liberal_ivdep` are disabled by default.

For UNICOS vector codes being transitioned to IRIX, it is recommended that `-OPT:cray_ivdep=ON` be used.

6.7 External Name Mapping Directive: NAME

The NAME directive allows you to specify a case-sensitive external name, or a name that contains characters outside of the Fortran character set, in a Fortran program. This directive must appear inside a program unit. The case-sensitive external name is specified on the NAME directive, in the following format:

```
!DIR$ NAME (fortran_name="external_name"
   [, fortran_name="external_name" ] ... )
```

fortran_name	The name used for the object throughout the Fortran program.
external_name	The external form of the name.

Rules for Fortran naming do not apply to the *external_name* string; any character sequence is valid. You can use this directive, for example, when writing calls to C routines.

Example:

```
      PROGRAM MAIN
!DIR$ NAME (FOO="XyZ")
      CALL FOO             ! XyZ IS REALLY BEING CALLED
      END PROGRAM
```

6.8 Inhibit Loop Interchange: NOINTERCHANGE

The NOINTERCHANGE directive inhibits the compiler's ability to interchange the loop that follows the directive with another inner or outer loop. The format of this directive is as follows:

```
!DIR$ NOINTERCHANGE
```

6.9 Determine Register Storage: NOSIDEEFFECTS

The NOSIDEEFFECTS directive allows the compiler to keep information in registers across a single call to a subprogram without reloading the information from memory after returning from the subprogram. The directive is not needed for intrinsic functions.

NOSIDEEFFECTS declares that a called subprogram does not redefine any variables that meet the following conditions:

- Local to the calling program
- Passed as arguments to the subprogram
- Accessible to the calling subprogram through host association
- Declared in a common block or module
- Accessible through USE association

The format of this directive is as follows:

```
!DIR$ NOSIDEEFFECTS f [, f ] ...
```

f	Symbolic name of a subprogram that the user ensures to have no side effects. *f* must not be the name of a dummy procedure, module procedure, or internal procedure.

A procedure declared NOSIDEEFFECTS should not define variables in a common block or module shared by a program unit in the calling chain. All arguments should be intent IN; that is, the procedure must not modify its arguments. If these conditions are not met, results are unpredictable.

The NOSIDEEFFECTS directive must appear in the specification part of a program unit and must appear before the first executable statement.

The compiler may move invocations of a NOSIDEEFFECTS subprogram from the body of a DO loop to the loop preamble if the arguments to that function are invariant in the loop. This may affect the results of the program, particularly if the NOSIDEEFFECTS subprogram calls functions such as the random number generator or the real-time clock.

The effects of the NOSIDEEFFECTS directive are similar to those that can be obtained by specifying the PURE prefix on a function or a subroutine declaration. For more information on the PURE prefix, see *Fortran Language Reference Manual, Volume 2*.

6.10 Designate a Nest to Task: PREFERTASK

The PREFERTASK directive allows loops with large iteration counts to be considered as candidates for tasking.

The compiler analyzes loops that follow a `PREFERTASK` directive to determine whether the loop is suitable for Autotasking. The `PREFERTASK` directive disables the compiler's threshold checking.

Note: The Autotasking directives are outmoded. Silicon Graphics encourages you to write new codes using the OpenMP Fortran API directives.

This directive can be used if there is more than one loop in the nest that can be autotasked. Autotasking must be enabled for this directive to take effect. The format of this directive is as follows:

```
!DIR$ PREFERTASK
```

In the following example, both loops can be autotasked, but the `PREFERTASK` directive directs the compiler to autotask the inner `DO J` loop. Without the directive and without any knowledge of `N` and `M`, the compiler would task the outer `DO I` loop. With the directive, the loops are interchanged, to increase parallel granularity, and the resulting outer `DO J` loop is autotasked.

```
      DO I = 1, N
!DIR$ PREFERTASK
        DO J = 1, M
          E(J,I) = F(J,I) + G(J,I)
        END DO
      END DO
```

6.11 Tasking Directives: TASK and NOTASK

The `NOTASK` directive suppresses compiler attempts to task loops and disables recognition of Autotasking directives. `NOTASK` takes effect at the next statement and applies to the rest of the program unit unless it is superseded by a `TASK` directive. These directives are disabled if tasking is disabled.

Note: The Autotasking directives are outmoded. Silicon Graphics encourages you to write new codes using the OpenMP Fortran API directives.

The formats of these directives are as follows:

```
!DIR$ TASK

!DIR$ NOTASK
```

When `!DIR$ NOTASK` has been used within the same program unit, `!DIR$ TASK` causes the compiler to resume its attempts to task loops. After a `TASK` directive is specified, the compiler again attempts to autotask loops and array syntax statements and `!MIC$` directives are again recognized.

The `TASK` directive affects subsequent loops. The `NOTASK` directive also affects subsequent loops, but if it is specified within the body of a loop, it affects the loop in which it is contained and all subsequent loops.

6.12 Unroll Loops: `UNROLL` and `NOUNROLL`

Loop unrolling can improve program performance by revealing cross-iteration memory optimization opportunities such as read-after-write and read-after-read. The effects of loop unrolling also include:

- Improved loop scheduling by increasing basic block size
- Reduced loop overhead
- Improved chances for cache hits

The formats of these directives are as follows:

```
!DIR$ UNROLL [ n ]

!DIR$ NOUNROLL
```

n	Specifies the total number of loop body copies to be generated. *n* must be a positive integer. If you specify a value for *n*, the compiler does not attempt to determine the number of copies to generate based on the number of inner loops in the loop nest.

The `UNROLL` directive should be placed immediately before the `DO` statement of the loop that should be unrolled.

Warning: If placed prior to a noninnermost loop, the `UNROLL` directive asserts that the following loop has no dependencies across iterations of that loop. If dependencies exist, incorrect code could be generated.

The `UNROLL` directive can be used only on loops whose iteration counts can be calculated before entering the loop. If `UNROLL` is specified on a loop that is not the innermost loop in a loop nest, the inner loops must be nested perfectly. That is, all loops in the nest can contain only 1 loop, and only the innermost loop can contain work.

The `NOUNROLL` directive inhibits loop unrolling.

Source Preprocessing [7]

Source preprocessing can help you port a program from one platform to another by allowing you to specify source text that is platform specific.

For a source file to be preprocessed automatically, it must have an uppercase extension, either `.F` (for a file in fixed source form) or `.F90` (for a file in free source form). Files with these suffixes are preprocessed automatically by the Fortran preprocessor.

To specify preprocessing of source files with other extensions, including lowercase ones, use the `-cpp`, `-E`, `-ftpp`, or `-P` options described in Chapter 2, page 5.

7.1 General Rules

You can alter the source code through source preprocessing directives. These directives are fully explained in Section 7.2, page 187. The directives must be used according to the following rules:

- Do not attempt macro substitution in Fortran comments. This will cause macros beginning with a `C` in column 1 (in fixed source form) not to be substituted.
- When the Fortran preprocessor is used, you must specify `-macro_expand` on the `f90`(1) command line if you want to enable macro expansion outside of preprocessor directive lines.
- Do not use source preprocessor (#) directives within multiline compiler directives.
- You cannot include a source file that contains an `#if` directive without a balancing `#endif` directive within the same file.

  The `#if` directive includes the `#ifdef` and `#ifndef` directives.
- If a directive is too long for one source line, the backslash character (\) is used to continue the directive on successive lines. Successive lines of the directive can begin in any column (up to the column limit of 132).

  The backslash character (\) can appear in any location within a directive in which whitespace can occur. A backslash character (\) in a comment is treated as a comment character. It is not recognized as signaling continuation.

- Every directive begins with the pound character (#), and the pound character (#) must be in column 1.
- Blank and tab (HT) characters can appear between the pound character (#) and the directive keyword.
- You cannot write form feed (FF) or vertical tab (VT) characters to separate tokens on a directive line. That is, if a source preprocessing line spans lines, it must be continued by using a backslash character (\).
- Blanks are significant, so the use of spaces within a source preprocessing directive is independent of the source form of the file. The fields of a source preprocessing directive must be separated by blank or tab (HT) characters.
- Because source preprocessing directives are independent of source form, a directive can be up to 132 columns on a single source line.

  Any directive text that extends past column 132 is ignored. The directive text is truncated, which is likely to produce parsing errors or unexpected results. If a directive is too long to fit on a single line, you can continue the line by using the backslash character (\). It cannot be continued using standard Fortran continuation methods.
- Any user-specified identifier that is used in a directive must follow Fortran rules for identifier formation. There are two exceptions to this rule:
  - The first character in the name can be an underscore character (_).
  - Although Fortran rules state that only the first 31 characters of identifiers are significant, to the source preprocessor, the first 132 characters are significant.
- Source preprocessing identifier names are case sensitive.
- Numeric literal constants must be integer literal constants or real literal constants, as defined for Fortran.
- Comments written in the style of the C language, beginning with `/*` and ending with `*/`, can appear anywhere within a source preprocessing directive in which blanks or tabs can appear. The comment, however, must begin and end on a single source line.
- The blanks shown in the syntax descriptions of the source preprocessing directives are significant. The tab character (HT) can be used in place of a

blank. Multiple blanks can appear wherever a single blank appears in a syntax description.

7.2 Directives

The following sections describe the source preprocessing directives.

7.2.1 `#include` Directive

The `#include` directive directs the system to use the content of a file or directory. Just as with the `INCLUDE` line processing defined by the Fortran standard, an `#include` directive effectively replaces that directive line with the content of *filename*. This directive has the following formats:

```
#include "filename"

#include <filename>
```

filename	A file or directory to be used. In the first form, if *filename* does not begin with a slash (`/`) character, the system searches for the named file, first in the directory of the file containing the `#include` directive, then in the sequence of directories specified by the `-I` option(s) on the `f90`(1) command line, and then the standard (default) sequence. If *filename* begins with a slash (`/`) character, it is used as is and is assumed to be the full path to the file. The second form directs the search to begin in the sequence of directories specified by the `-I` option(s) on the `f90`(1) command line and then search the standard (default) sequence.

The Fortran standard prohibits recursion in `INCLUDE` files, so recursion is also prohibited in the `#include` form.

The `#include` directives can be nested.

When the compiler is invoked to do only source preprocessing, not compilation, text will be included by `#include` directives but not by Fortran `INCLUDE` lines. For information on the source preprocessing command line options, see Section 7.4, page 194.

7.2.2 `#define` Directive

The `#define` directive lets you declare a source preprocessing variable and associate a token string with the variable. It also allows you to define a function-like macro. This directive has the following formats:

```
#define identifier value

#define identifier(dummy_arg_list) value
```

The first format defines an object-like macro (also called a *source preprocessing variable*), and the second defines a function-like macro. In the second format, the left parenthesis that begins the *dummy_arg_list* must immediately follow the identifier, with no intervening white space.

identifier	Specifies the name of the variable or macro being defined.
dummy_arg_list	Specifies a list of dummy argument identifiers.
value	Specifies the *value* as a sequence of tokens. The *value* can be continued onto more than one line using backslash (\) characters.

If a preprocessor *identifier* appears in a subsequent `#define` directive without being the subject of an intervening `#undef` directive, and the *value* in the second `#define` directive is different from the value in the first `#define` directive, then the preprocessor issues a warning message about the redefinition. The second directive's *value* is used. For more information on the `#undef` directive, see Section 7.2.3, page 189.

When an object-like macro's identifier is encountered as a token in the source file, it is replaced with the value specified in the macro's definition. This is referred to as an *invocation* of the macro. By default, tokens are not processed in Fortran source code. They are recognized only when used in other source preprocessing directives.

The invocation of a function-like macro is more complicated. It consists of the macro's identifier, immediately followed by a left parenthesis with no intervening white space, then a list of actual arguments separated by commas, and finally a terminating right parenthesis. There must be the same number of actual arguments in the invocation as there are dummy arguments in the `#define` directive. Each actual argument must be balanced in terms of any internal parentheses. The invocation is replaced with the value given in the macro's definition, with each occurrence of any dummy argument in the definition replaced with the corresponding actual argument in the invocation.

The following two examples must be compiled with `-macro_expand` specified on the `f90`(1) command line:

- The following program prints `Hello, world.` when compiled and run:

```
      PROGRAM P
#define GREETING 'Hello, world.'
      PRINT *, GREETING
      END PROGRAM P
```

- The following program prints `Hello, Hello, world.` when compiled and run:

```
      PROGRAM P
#define GREETING(str1, str2) str1, str1, str2
      PRINT *, GREETING('Hello, ', 'world.')
      END PROGRAM P
```

For information on the `-macro_expand` option, see Section 2.44, page 37.

7.2.3 #undef Directive

The `#undef` directive sets the definition state of *identifier* to an undefined value. If *identifier* is not currently defined, the `#undef` directive has no effect. This directive has the following format:

```
#undef identifier
```

identifier	Specifies the name of the source preprocessing variable or macro being undefined.

7.2.4 # (Null) Directive

The null directive simply consists of the pound character (#) in column 1 with no significant characters following it. That is, the remainder of the line is typically blank or is a source preprocessing comment. This directive is generally used for spacing out other directive lines.

7.2.5 Conditional Directives

Conditional directives cause lines of code to either be produced by the source preprocessor or to be skipped. The conditional directives within a source file form *if-groups*. An if-group begins with an `#if`, `#ifdef`, or `#ifndef` directive, followed by lines of source code that you may or may not want skipped. Several similarities exist between the Fortran `IF` construct and if-groups:

- The `#elif` directive corresponds to the `ELSE IF` statement.
- The `#else` directive corresponds to the `ELSE` statement.
- Just as an `IF` construct must be terminated with an `END IF` statement, an if-group must be terminated with an `#endif` directive.
- Just as with an `IF` construct, any of the blocks of source statements in an if-group can be empty.

  For example, you can write the following directives:

  ```
  #if MIN_VALUE == 1
  #else
    ...
  #endif
  ```

Determining which group of source lines (if any) to compile in an if-group is essentially the same as the Fortran determination of which block of an `IF` construct should be executed.

7.2.5.1 `#if` Directive

The `#if` directive has the following format:

```
#if expression
```

expression An expression. The values in *expression* must be integer literal constants or previously defined preprocessor variables. The expression is an integer constant expression as defined by the C language standard. All the operators in the expression are C operators, not Fortran operators. The *expression* is evaluated according to C language rules, not Fortran expression evaluation rules.

Note that unlike the Fortran `IF` construct and `IF` statement logical expressions, the *expression* in an `#if` directive need not be enclosed in parentheses.

The `#if` expression can also contain the unary `defined` operator, which can be used in either of the following formats:

```
defined identifier

defined(identifier)
```

When the `defined` subexpression is evaluated, the value is 1 if *identifier* is currently defined, and 0 if it is not.

All currently defined source preprocessing variables in *expression*, except those that are operands of `defined` unary operators, are replaced with their values. During this evaluation, all source preprocessing variables that are undefined evaluate to 0.

Note that the following two directive forms are **not** equivalent:

- `#if X`
- `#if defined(X)`

In the first case, the condition is true if `X` has a nonzero value. In the second case, the condition is true only if `X` has been defined (has been given a value that could be 0).

7.2.5.2 `#ifdef` Directive

The `#ifdef` directive is used to determine if *identifier* is predefined by the source preprocessor, has been named in a `#define` directive, or has been named in the `-D` option on the `f90`(1) command line. For more information on the `-D` option, see Section 7.4, page 194.

This directive has the following format:

```
#ifdef identifier
```

The `#ifdef` directive is equivalent to either of the following two directives:

- `#if defined` *identifier*
- `#if defined(`*identifier*`)`

7.2.5.3 `#ifndef` Directive

The `#ifndef` directive tests for the presence of an *identifier* that is not defined. This directive has the following format:

```
#ifndef identifier
```

This directive is equivalent to either of the following two directives:

- `#if !  defined` *identifier*
- `#if !  defined(`*identifier*`)`

7.2.5.4 `#elif` Directive

The `#elif` directive serves the same purpose in an if-group as does the `ELSE IF` statement of a Fortran `IF` construct. This directive has the following format:

```
#elif expression
```

expression	The expression follows all the rules of the integer constant expression in an `#if` directive.

7.2.5.5 `#else` Directive

The `#else` directive serves the same purpose in an if-group as does the `ELSE` statement of a Fortran `IF` construct. This directive has the following format:

```
#else
```

7.2.5.6 `#endif` Directive

The `#endif` directive serves the same purpose in an if-group as does the `END IF` statement of a Fortran `IF` construct. This directive has the following format:

```
#endif
```

7.3 Predefined Macros

The MIPSpro 7 Fortran 90 source preprocessor supports a number of predefined macros. They are divided into groups as follows:

- Macros that are based on the host machine
- Macros that are based on IRIX system targets

The following predefined macros are based on the host system (the system upon which the compilation is being done):

Macro	Notes
`__unix`	Always defined. The leading characters consist of 2 consecutive underscores.

The following predefined macros are based on an IRIX system target:

Macro	Notes
`_ABI`*abi*`=`*n*	Defined when *abi* is set to `N32` or `64`. Its value is the instruction set architecture. For example, `_ABIN32=2` is set when `-n32` is specified on the `f90`(1) command line; `_ABI64=3` is set when `-64` is specified on the `f90`(1) command line. For information on the `f90`(1) command line, see Chapter 2, page 5.
`_COMPILER_VERSION`	Defined as the compiler version. For example, for the MIPSpro 7.2.1 release it is set as follows: `_COMPILER_VERSION=721`.
`LANGUAGE_FORTRAN90`, `_LANGUAGE_FORTRAN90`	
`__host_mips`	The leading characters in the second form consist of 2 consecutive underscores.

`LANGUAGE_FORTRAN`, `_LANGUAGE_FORTRAN`	
`MIPSEB`, `_MIPSEB`	
`__mips`	Set to the instruction set architecture, either `3` or `4`. The leading characters consist of 2 consecutive underscores.
`_MIPS_ISA`	Set to the instruction set architecture, either `3` or `4`.
`_MIPS_SIM`	Set to the instruction set architecture, as follows: `_MIPS_SIM=_ABIN32` when `-n32` is specified on the `f90`(1) command line; `_MIPS_SIM=_ABI64` when `-64` is specified on the `f90`(1) command line. For information on the `f90`(1) command line, see Chapter 2, page 5.
`_OPENMP`	
`__sgi`	The leading characters consist of 2 consecutive underscores.
`_SYSTYPE_SVR4`	

7.4 Command Line Options

Several `f90`(1) command line options affect source preprocessing. The following list indicates the sections in this manual that describe preprocessing options:

- Section 2.12, page 11, describes the `-cpp` option.
- Section 2.15, page 12, describes the `-D`*var*[=*def*][`,`*var*[=*def*]]... option.
- Section 2.18, page 13, describes the `-E` option.
- Section 2.25, page 16, describes the `-ftpp` option.
- Section 2.44, page 37, describes the `-macro_expand` option.
- Section 2.52, page 44, describes the `-nocpp` option.
- Section 2.58, page 55, describes the `-P` option.

- Section 2.69, page 64, describes the `-U`*var* option.

The `-D` *identifier*[=*value*] `[,` *identifier*[=*value*]] `...`, `-F`, and `-U` *identifier* `[,` *identifier*] `...` options are ignored unless the Fortran input source file is specified as either *file*`.F` or *file*`.F90`.

Interlanguage Calling [8]

You may want to call external procedures written in C, C++, or some other language, or you may need to call a Fortran procedure from one of those languages. This chapter focuses on the interface between Fortran and C/C++.

If your application has source programs written in different languages, you should compile each file separately, with the appropriate compiler, and then load them in a separate step. You can create object files suitable for loading by specifying the `-c` option on the `f90`(1) command, which disables the load step and writes the binary file to *file*`.o`. For information on the `-c` option, see Section 2.6, page 10.

In the following example, the C/C++ compiler and the MIPSpro 7 Fortran 90 compilers produce object files that can be loaded. These files are named `main.o` and `rest.o`:

```
%   cc -c main.c
%   f90 -c rest.f
```

This chapter provides more details on compiling and loading application programs that are written in Fortran, C, and C++.

8.1 External and Public Names

When your Fortran program defines the body of a procedure, the compiler places the name of the procedure, as a character string, in the object file it generates. This is a *public name*, which is accessible to other object files.

When your Fortran program uses a procedure, the compiler places the name of the procedure in the generated object file. This is an *external name*, which is used by the object file but not defined in it. Names of common blocks and names of data and procedures declared within object files are also external names. You can use the `nm`(1) utility to display the public and external names defined in a file. For more information on this utility, see the *MIPSpro Compiling and Performance Tuning Guide*.

It is up to the IRIX loader, `ld`(1), to resolve each reference to an external name by finding that same name as a public name in some other module. This is the main job of the loader.

8.1.1 Fortran Treatment of External and Public Names

The Fortran compiler ignores the case of the input source text (other than the contents of character literals). As a result, it may change the case of the names of procedures and named common blocks while it translates the source file. The names recorded in the object file are changed in the following two ways from the way you may have written them:

- They are converted to all lowercase letters.
- They are normally extended with a final underscore (_) character.

Procedure names and common block names are translated, too.

The following declarations produce the identifiers `matrix_`, `mixedcase_`, and `cblk_` in the object file:

```
SUBROUTINE MATRIX
external function MixedCase()
COMMON /CBLK/a,b,c
```

These changes cause no problems when loading program units compiled by Fortran. The same convention is used for both the public and external names, so the names match.

> **Note:** Some IRIX-based FORTRAN 77 compilers support the `-U` command line option, which prevents the compiler from forcing all uppercase input to lowercase. As a byproduct, it becomes possible to put mixed case public names in the object file. This option is not supported by the MIPSpro 7 Fortran 90 compiler.
>
> In addition, some IRIX-based FORTRAN 77 compilers take the use of the `$` character as the final letter of a procedure name as a signal to suppress the underscore in the public name. The `$` is not permitted to appear in a name if the program is to be compiled by the MIPSpro 7 Fortran 90 compiler. There is no way to suppress the final underscore in an external name.
>
> You can override the default conventions by using the `!DIR$ NAME` directive described in Section 6.7, page 180.

Module and internal procedure names are connected with `.in.` to make a unique name. For example, the following code creates procedures named `MPROC.in.MMM` and `IPROC.in.MPROC.in.MMM`:

```
MODULE MMM
...
CONTAINS
```

```
SUBROUTINE MPROC()
...
CONTAINS
  SUBROUTINE IPROC()
  ...
```

8.1.2 Calling a Fortran Subprogram from C

To call a Fortran subprogram from a C procedure, spell the name the way the Fortran compiler spells it, using all lowercase letters and a trailing underscore.

For example, consider the following Fortran declaration:

```
SUBROUTINE HYPOT()
```

This must be declared in a C function as follows (note the use of lowercase with a trailing underscore):

```
extern int hypot_()
```

Note: You cannot call Fortran subroutines that contain assumed-shape dummy arguments or Fortran pointer arguments from C.

8.1.3 Calling a C Function from Fortran

To call a C function from a Fortran program, ensure that the C function's name is spelled the way the Fortran compiler expects it to be. When you control the name of the C function, the simplest solution is to give it a name that consists of lowercase letters with a terminal underscore. For example, the following C function:

```
int fromfort_() {...}
```

could be declared in a Fortran program as follows:

```
external FROMFORT
```

When you do not control the name of a C function, you must either supply a function name that the Fortran compiler can call or use the `!DIR$ NAME` directive described in Section 6.7, page 180. If you choose to supply a function name that the Fortran compiler can call, you need to write a C function that accepts the same arguments and has a name composed of lowercase letters followed by an underscore. This C function can then call the function whose name contains mixed case letters. You can write such a wrapper function manually, or you can use the `mkf2c`(1) utility to do it automatically.

8.2 Correspondence of Fortran and C Data Types

When you exchange data between Fortran and C, either as arguments, as function results, or as members of common blocks, you must make sure that the two languages agree on the size, alignment, and subscript of each data object.

8.2.1 Corresponding Scalar Types

The correspondence between Fortran and C scalar data types is shown in Table 2. This table assumes that the default command line options that affect precision are in effect. Certain `f90`(1) command line options (such as `-i2` or `-r8`) affects storage sizes for integer, logical, real, and double precision data types. For information on the `-i2` and `-r8` options, see Section 2.30, page 17, and Section 2.62, page 57.

Table 2. Corresponding Fortran and C Data Types

Fortran Data Type Declaration	C Data Type
`INTEGER(KIND=1)`, `LOGICAL(KIND=1)`	`signed char`
`CHARACTER`	`unsigned char`
`INTEGER(KIND=2)`, `LOGICAL(KIND=2)`	`short`
`INTEGER`, `INTEGER(KIND=4)`, `LOGICAL`, `LOGICAL(KIND=4)`	`int`
`INTEGER(KIND=8)`, `LOGICAL(KIND=8)`	`long long`
`REAL`, `REAL(KIND=4)`	`float`
`DOUBLE PRECISION`, `REAL(KIND=8)`	`double`
`REAL(KIND=16)`	`long double`
`COMPLEX`, `COMPLEX(KIND=4)`	`struct{float real, imag;};`

Fortran Data Type Declaration	C Data Type
`DOUBLE COMPLEX`, `COMPLEX(KIND=8)`	`struct{double real, imag;};`
`COMPLEX(KIND=16)`	`struct{long double real, imag;};`
`CHARACTER(`*n*`)`	`char fstr_`*n*`[`*n*`]`

For type character, Fortran declarations with a length designator, such as `CHARACTER(LEN=N) :: X`, are equivalent to a C declaration of `unsigned char X[N]`.

To set a NULL character in a Fortran string, use `CHAR(0)`. Examples:

```
character*4 aaa
aaa(1:3) = 'abc'
aaa(4:4) = CHAR(0)
```

8.2.2 Corresponding Character Types

The Fortran `CHARACTER` data type declaration corresponds to the C type `unsigned char`. However, the two languages differ in the treatment of strings of characters.

A Fortran variable can be declared as `CHARACTER(`*n*`)`, where *n*>1, contains exactly *n* characters at all times. When a shorter character expression is assigned to it, it is padded on the right with spaces to reach *n* characters.

A C vector of characters is normally sized 1 greater than the longest string assigned to it. It may contain fewer meaningful characters than its size allows, and the end of meaningful data is marked by a null byte. There is no null byte at the end of a Fortran string (except by chance memory alignment).

There is no terminal null byte, so most of the string library functions familiar to C programmers (`strcpy()`(3C), `strcat()`(3C), `strcmp()`(3C), and so on) cannot be used with Fortran string values. The `strncpy()`(3C), `strncmp()`(3C), `bcopy()`(3C), and `bcmp()`(3C) functions can be used because they depend on a count rather than a delimiter.

8.2.3 Unsupported Array Arguments

Fortran supports assumed-shape arrays, deferred-shape arrays, and array sections. You cannot pass any of these types of array to a non-Fortran procedure because Fortran represents such arrays in memory using a descriptor containing indirect pointers and other data. The format of this descriptor is not part of the published programming interface to the MIPSpro 7 Fortran 90 compiler because it is subject to change.

8.3 How Fortran Passes Arguments

When calling non-Fortran functions, you must know how arguments are passed. When calling Fortran subprograms from other languages, you must cause the other language to pass arguments correctly.

Note: All compilers for a given version of IRIX use identical conventions for passing arguments. These conventions are documented at the machine instruction level in the *MIPSpro Assembly Language Programmer's Guide*, which also describes the differences in the conventions used in different releases.

An argument passed to a subprogram, regardless of its data type, is passed as the address of the actual in memory. This rule is extended for two special cases:

- The length of each `CHARACTER(`*n*`)` declaration is passed as an implicit additional `INTEGER(KIND=4)` value, following the explicit arguments.
- When a function returns type `CHARACTER(`*n*`)`, the address of the space to receive the result is passed as the first argument to the function, and the length of the result space is passed as the second implicit argument, preceding all explicit arguments.

Example 1. Consider the following code:

```
COMPLEX(KIND=8) :: CMPLX8
CHARACTER*(16) :: CSTR1, CSTR2
EXTERNAL CPXASC
CALL CPXASC(CSTR1,CSTR2,CMPLX8)
```

The code generated from the subroutine call in this example passes the following arguments:

- The address of `CSTR1`
- The address of `CSTR2`
- The address of `CMPLX8`
- The length of `CSTR1`, an integer value of 16
- The length of `CSTR2`, an integer value of 16

Example 2. Consider the following code:

```
CHARACTER*(8) :: SYMBL,PICKSYM
CHARACTER*(100) :: SENTENCE
INTEGER NSYM
SYMBL = PICKSYM(SENTENCE,NSYM)
```

The code generated from the function call in the preceding example passes the following arguments:

- The address of an unnamed result variable
- The length of an unnamed result variable
- The address of `SENTENCE`, the first explicit argument
- The address of `NSYM`, the second explicit argument
- The length of `SENTENCE`, an integer value of 100

8.4 Calling Fortran from C

There are two types of callable Fortran subprograms: subroutines and functions. In C terminology, both types of subprograms are external functions. The difference is the use of the function return value from each.

8.4.1 Calling a Fortran Subroutine from C

From the standpoint of a C function, a Fortran subroutine is an external function returning `void`.

Example 1. The following example shows a simple Fortran subroutine that adds arrays of complex numbers:

```
SUBROUTINE ADDC32(Z, A, B, N)
INTEGER :: N
COMPLEX(KIND=16),DIMENSION(N) :: Z,A,B
Z = A + B
RETURN
END SUBROUTINE
```

The Fortran subroutine could be called from C using the following code fragment:

```
typedef struct{long double real, imag;} cpx32;
extern void
   addc32_(cpx32 *,cpx32 *,cpx32 *,int *);
cpx32 z[MAXARRAY], a[MAXARRAY], b[MAXARRAY];
...
      int n = MAXARRAY;
      addc32_(&z, &a, &b, &n);
```

The preceding code fragments show how the Fortran subroutine is named in the C code using lowercase letters and a terminal underscore. This is the way the Fortran compiler spells the public name in the object file.

Example 2. The following subroutine takes assumed-length character arguments:

```
SUBROUTINE PRT(BEF, VAL, AFT)
CHARACTER*(*) :: BEF, AFT
REAL :: VAL
PRINT *, BEF, VAL, AFT
RETURN
END SUBROUTINE PRT
```

The following C code prepares `CHARACTER(16)` values and passes them to the Fortran subroutine:

```
typedef char fstr_16[16];
extern void
     prt_(fstr_16 *, float *, fstr_16 *,
                  int, int);
main()
{
      float val = 2.1828e0;
      fstr_16 bef,aft;
      strncpy(bef,"Before..........",sizeof(bef));
      strncpy(aft,"..........After",sizeof(aft));
      prt_(bef, &val, aft, sizeof(bef), sizeof(aft));
}
```

Note that the subroutine call requires five actual arguments: the addresses of the three explicit arguments and the lengths of the two string arguments. In the C code, the string length arguments are generated using `sizeof()`, which returns the memory size of the typedef `fstr_16`.

When the Fortran code does not require a specific string length, the C code that calls it can pass an ordinary C character vector, as shown in the following code fragment:

```
extern int
prt_(char *, float *, char *, int, int);
main()
{
          float val = 2.1828e0;
          char *bef = "Start:";
          char *aft = ":End";
          (void)prt_(bef, &val, aft, strlen(bef), strlen(aft));
}
```

In this example, the string length implicit argument values are calculated dynamically using `strlen()`.

8.4.2 Calling a Fortran Function from C

A Fortran function that returns a scalar value as its result corresponds exactly to the C concept of a function with an explicit return value. When a Fortran function returns any type shown in Table 2, page 200, other than `CHARACTER(`*n*`)`, where *n*>1, you can call the function from C and handle its return value exactly as if it were a C function returning that data type.

Example 1. The following function accepts and returns `COMPLEX(KIND=8)` values.

```
FUNCTION FSUB8(INP)
COMPLEX(KIND=8) :: INP,FSUB8
FSUB8 = INP
END FUNCTION FSUB8
```

Although a complex value is declared as a structure in C, it can be used as the return type of a function. The following C code shows how the preceding Fortran function is declared and called:

```
typedef  struct{ double real, imag; } cpx8;
extern cpx8 fsub8_(cpx8 *);
main()
{
      cpx8 inp = { -3.333, -5.555 };
      cpx8 oup = { 0.0, 0.0 };
      printf("testing fsub8...");
      oup = fsub8_( &inp );
      if ( inp.real == oup.real && inp.imag == oup.imag )
          printf("Ok\n");
      else
          printf("Nope\n");
}
```

The arguments to a function, like the arguments to a subroutine, are passed as pointers, but the value returned is a value, not a pointer to a value.

Example 2. The following function has a `CHARACTER(16)` return value.

```
FUNCTION FS16(J, K, S)
   CHARACTER*(16) :: FS16, S
   INTEGER J, K
   FS16 = S(J:K)
RETURN
END FUNCTION FS16
```

When a Fortran function returns `CHARACTER(`*n* `)`, where *n*>1, value, the returned value is not the explicit result of the function. Instead, you must pass the address and length of the result area as the first two arguments of the function, preceding the explicit arguments. This is demonstrated in the following C code:

```
typedef char fstr_16[16];
extern void
fs16_ (fstr_16 *, int, int *, int *, fstr_16 *, int);
main()
{
      char work[64];
      fstr_16 inp, oup;
      int j = 7;
      int k = 11;
      strncpy(inp,"0123456789abcdef", sizeof(inp));
      fs16_ ( oup, sizeof(oup), &j, &k, inp, sizeof(inp) );
      strncpy(work, oup, sizeof(oup));
      work[sizeof(oup)] = ’\0’;
      printf("FS16 returns <%s>\n", work);
}
```

In this example, the address and length of the function result are the first two arguments of the function. Because type `fstr_16` is an array, its name, `oup`, evaluates to the address of its first element. The next three arguments are the addresses of the three named arguments. The final argument is the length of the string argument.

8.5 Calling C from Fortran

You can call units of C code from Fortran as if they were written in Fortran, provided that the C modules follow the Fortran conventions for passing arguments. For more information on this, see Section 8.3, page 202.

When the C function expects arguments passed using other conventions, you normally need to build a wrapper for the C function using the `mkf2c`(1) command.

8.5.1 Calls to C Functions

The following C function is written to use the Fortran conventions for its name (lowercase with final underscore) and for argument passing:

```
/*
|| C functions to export the facilities of strtoll()
|| to Fortran programs.  Effective Fortran declaration:
||
|| FUNCTION ISCAN(S,J)
|| INTEGER(KIND=8) :: ISCAN
|| CHARACTER*(*) S
|| INTEGER J
||
|| String S(J:) is scanned for the next signed long value
|| as specified by strtoll(3c) for a "base" argument of 0
|| (meaning that octal and hex literals are accepted).
||
|| The converted long long is the function value, and J is
|| updated to the nonspace character following the last
|| converted character, or to 1+LEN(S).
||
|| Note: if this routine is called when S(J:J) is neither
|| whitespace nor the initial of a valid numeric literal,
|| it returns 0 and does not advance J.
*/
#include <ctype.h> /* for isspace() */
long long iscan_(char *ps, int *pj, int ls)
{
   int  scanPos, scanLen;
   long long ret = 0;
   char wrk[1024];
   char *endpt;
   /* when J>LEN(S), do nothing, return 0 */
   if (ls >= *pj)
   {
      /* convert J to origin-0, permit J=0 */
      scanPos = (0 < *pj)? *pj-1 : 0 ;

      /* calculate effective length of S(J:) */
      scanLen = ls - scanPos;

      /* copy S(J:) and append a null for strtoll() */
      strncpy(wrk,(ps+scanPos),scanLen);
      wrk[scanLen] = '\0';

      /* scan for the integer */
      ret = strtoll(wrk, &endpt, 0);

      /*
      || Advance over any whitespace following the number.
      || Trailing spaces are common at the end of Fortran
      || fixed-length char vars.
      */
      while(isspace(*endpt)) { ++endpt; }
      *pj = (endpt - wrk)+scanPos+1;
   }
   return ret;
}
```

The following Fortran code fragment demonstrates a call to the preceding C function:

```
EXTERNAL ISCAN
INTEGER(KIND=8) ISCAN
INTEGER(KIND=8) RET
INTEGER J,K
CHARACTER*(50) INP
INP = '1  -99   3141592  0xfff  033 '
J = 0
DO WHILE (J .LT. LEN(INP))
      K = J
      RET = ISCAN(INP,J)
      PRINT *, K,': ',RET,' -->',J
END DO
END
```

8.5.2 Using Fortran Common Blocks in C Code

A C function can refer to the contents of a common block defined in a Fortran program. The name of the block as given in the COMMON statement is altered as described in Section 8.1.1, page 198. (The name is converted to lowercase and extended with an underscore). The name of the blank common is _BLNK__, with one leading underscore and two trailing ones.

To refer to the contents of a common block, take these steps:

1. Declare a C structure with fields that have the appropriate data types to match the successive elements of the Fortran common block. For information on corresponding data types, see Table 2, page 200.
2. Declare the common block name as an external structure of that type.

The following example employs this method:

```
        INTEGER STKTOP, STKLEN, STACK(100)
        COMMON /WITHC/ STKTOP, STKLEN, STACK

struct fstack {
      int stktop, stklen;
      int stack[100];
}
extern fstack withc_;
int peektop_()
{
      if (withc_.stktop) /* stack not empty */
          return withc_.stack[withc_.stktop-1];
      else...
}
```

The restrictions on this capability are as follows:

- You cannot map a common block that contains Fortran pointer-based variables.

- If the common block contains a variable of Fortran derived type (a structure), ensure that the derived type is declared with the `SEQUENCE` attribute. Otherwise, its fields may not appear in the expected sequence in memory.
- When `-O3` is in effect, the compiler may split up common blocks. For information on the `-O3` option to the `f90`(1) command, see Section 2.56, page 45.

8.5.3 Using Fortran Arrays in C Code

A C program can access arrays created in Fortran. The following example illustrates this.

The following Fortran code fragment declares a matrix in a common block and then calls a C subroutine to modify the array:

```
INTEGER IMAT(10,100), R, C
COMMON /WITHC/ IMAT
R = 74
C = 6
CALL CSUB(C, R, 746)
PRINT *, IMAT(6,74)
END
```

The following C function stores its third argument in the common array using the subscripts passed in the first two arguments. In the C function, the order of the dimensions of the array are reversed, so the subscript values are reversed to match, and decremented by 1 to provide 0-origin indexing:

```
extern struct { int imat[100][10]; } withc_;
void csub_(int *pc, int *pr, int *pval)
{
      withc_.imat[*pr-1][*pc-1] = *pval;
}
```

8.5.4 Calls to C Using `LOC` and `%VAL`

You can use the nonstandard intrinsic functions `%VAL` and `LOC` to pass arguments in ways other than the standard Fortran conventions described in Section 8.3, page 202.

8.5.4.1 Using %VAL

The %VAL function is used in an argument list to cause an argument to be passed by value rather than by reference. Suppose that you need to call a C function having the following prototype in file ti.c:

```
#includevoid takesint_(int i, char *s, int len)
{
   printf("i: %d\n", i);
   printf("s: %.*s\n", len, s);
}
```

The first argument to this function is an integer value, not the address of an integer value in memory. You could call this function from the following Fortran code in file ti_f.f:

```
      CHARACTER(80) SENTENCE
      INTEGER(4) J
      J = 13
      SENTENCE = "Hello, there."
      CALL TAKESINT(%VAL(J), SENTENCE)
      END
```

The use of %VAL(*j*) causes the contents of *j* to be passed, rather than the address of *j*.

```
% f90 -n32 ti_f.f ti.c
ti_f.f:
ti.c:
% ./a.out
i: 13
s: Hello, there.
```

8.5.4.2 Using LOC

The LOC function returns the address of its argument. It can be used with %VAL to prevent passing the length of a character value as a hidden argument. In other words, the argument %VAL(LOC(*char_var*)) passes only the address of *char_var*. It does not pass the implicit length argument.

8.6 Calling Assembly Language from Fortran

You can write modules in MIPS assembly language, following the guidelines in the *MIPSpro Assembly Language Programmer's Guide*. Procedures in these modules can be called from Fortran. There is only one special consideration.

Operating in assembly language, you can change the operating mode and the rounding mode of the CPU. When running Fortran programs that contain quad precision operations, you must run the compiler in round-to-nearest mode. This mode is in effect by default, so you usually do not need to set it. When writing programs that call your own assembly routines, ensure that this mode is set. For more information, see the `swapRM`(3C) man page.

The Auto-Parallelizing Option (APO) [9]

The Auto-Parallelizing Option (APO) enables the MIPSpro 7 Fortran 90 compiler to optimize parallel codes and enhances performance on multiprocessor systems. APO is controlled with command line options and source directives.

Note: APO is licensed and sold separately from the MIPSpro 7 Fortran 90 compiler. APO features in your code are ignored unless you are licensed for this product. For sales and licensing information, contact your Silicon Graphics sales representative.

APO is integrated into the compiler; it is not a source-to-source preprocessor. Although runtime performance suffers slightly on single-processor systems, parallelized programs can be created and debugged with APO enabled.

Parallelization is the process of analyzing sequential programs for parallelism and restructuring them to run efficiently on multiprocessor systems. The goal is to minimize the overall computation time by distributing the computational workload among the available processors. Parallelization can be automatic or manual.

During *automatic parallelization*, the compiler analyzes and restructures the program with little or no intervention by you. With APO, the compiler automatically generates code that splits the processing of loops among multiple processors. An alternative is *manual parallelization*, in which you perform the parallelization using compiler directives and other programming techniques.

As the following figure shows, APO integrates automatic parallelization with other compiler optimizations, such as interprocedural analysis (IPA), optimizations for single processors, and loop nest optimization (LNO):

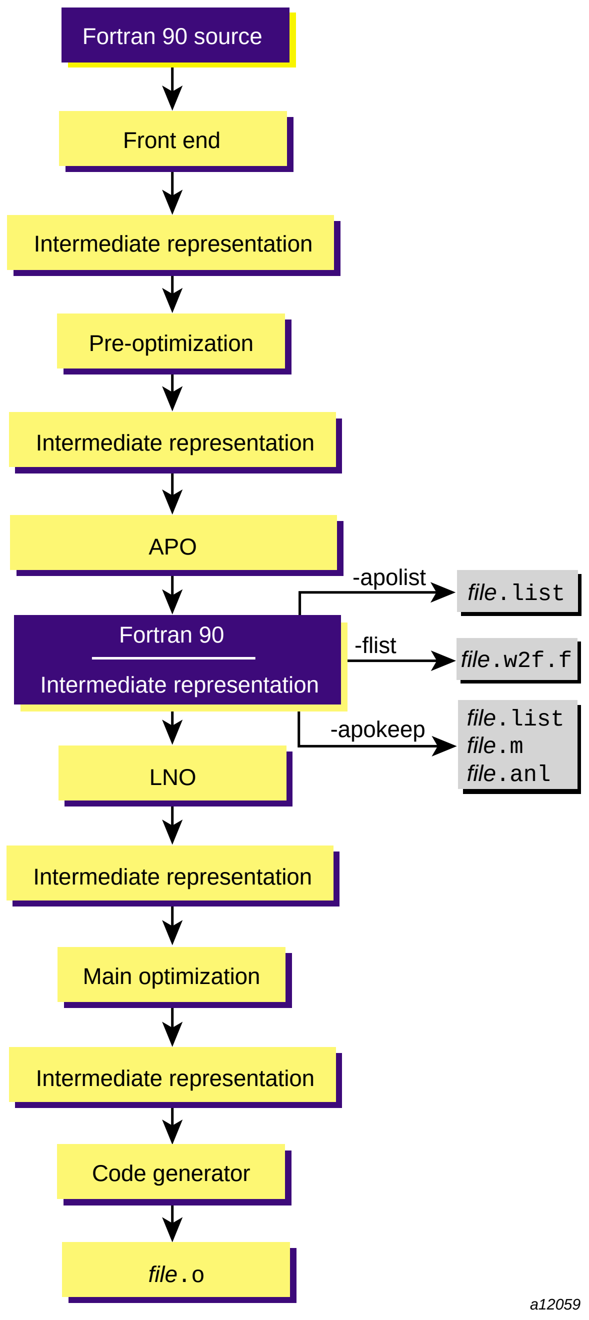

Figure 9. Files Generated by the ProDev Automatic Parallelization Option

9.1 `f90`(1) Command Line Options That Affect APO

Several `f90`(1) command line options described in Chapter 2, page 5, control APO's effect on your program. The following command line, for example, invokes APO and requests aggressive optimization:

```
f90 -apo -O3 zebra.f
```

The following subsections describe the effects that various `f90`(1) command line options have on APO.

Note: If you invoke the loader separately, you must specify the `-apo` option on the `ld`(1) command line.

9.1.1 `-apo`

The `-apo` option invokes APO. When this option is enabled, the compiler automatically converts sequential code into parallel code by inserting parallel directives where it it safe and beneficial to do so. Specifying `-apo` also enables the `-mp` option, which enables recognition of the parallel directives inserted into your code.

9.1.2 `-apokeep` and `-apolist`

The `-apokeep` and `-apolist` options control output files. Both options generate *file*`.list`, which is a listing file that contains information on the loops that were executed in parallel and explains why others were not executed in parallel.

When `-apokeep` is specified, the compiler writes *file*`.list`, and in addition, it retains *file*`.anl` and *file*`.m`. The ProDev ProMP tools use *file*`.anl`. For more information on ProDev ProMP, see the *ProDev ProMP User's Guide*. *file*`.m` is an annotated version of your source code that shows the insertion of multiprocessing directives.

For more information on the content of *file*`.list`, *file*`.anl`, and *file*`.m`, see Section 9.2.2, page 219.

Note: Because of data conflicts, do not specify the `-mplist` or `-FLIST` options when `-apokeep` is specified.

9.1.3 `-flist`

This option generates a Fortran listing and directs the compiler to write the transformed source code and multiprocessing directives to *file*`.w2f.f`. For more information on the content of *file*`.w2f.f`, see Section 9.2, page 218.

9.1.4 `-IPA:...`

Interprocedural analysis (IPA) is invoked by the `-IPA` command line option. It performs program optimizations that can only be done by examining the whole program rather than processing each procedure separately.

When APO is invoked with IPA, only those loops with calls determined to be safe are parallelized.

If IPA expands subroutines inline in a calling routine, the subroutines are compiled with the options of the calling routine. If the calling routine is not compiled with `-apo`, none of its inlined subroutines are parallelized. This is true even if the subroutines are compiled separately with `-apo` because with IPA, automatic parallelization is deferred until link time.

If `-apokeep` or `-pfakeep` are specified in conjunction with `-ipa` or `-IPA`, the default settings for IPA suboptions are used with the exception of the `inline=`*setting* suboption. For that suboption, the default becomes `OFF`.

For more information on the effect of IPA, see Section 9.4.1.2, page 224. For more information on IPA itself, see the `ipa`(5) man page.

9.1.5 `-LNO:...`

The `-LNO` options control the Loop Nest Optimizer (LNO). LNO is enabled by default at `-O3`. LNO performs loop optimizations that better exploit caches and instruction-level parallelism. The following LNO options are of particular interest to APO users:

- `-LNO:auto_dist=on`. This option requests that APO insert data distribution directives to provide the best memory utilization on Origin2000 systems.
- `-LNO:ignore_pragmas=`*setting*. This option directs APO to ignore all of the directives and assertions described in Section 9.5, page 231.
- `-LNO:parallel_overhead=`*num_cycles*. This option allows you to override certain compiler assumptions regarding the efficiency to be gained

by executing certain loops in parallel rather than serially. Specifically, changing this setting changes the default estimate of the cost to invoke a parallel loop in your runtime environment. This estimate varies depending on your particular runtime environment, but it is typically several thousand machine cycles.

9.1.6 `-O3`

To obtain maximum performace, specify `-O3` when compiling with APO enabled. The optimizations at this level maximize code quality even if they require extensive compile time or relax the language rules. In addition, LNO is enabled by default at this `-O` level.

The `-O3` option uses transformations that are usually beneficial but can sometimes hurt performance. This optimization may cause noticeable changes in floating-point results due to the relaxation of operation-ordering rules. Floating-point optimization is discussed further in Section 9.1.7, page 217.

For more information on the `-O3` option, see Section 2.56, page 45.

9.1.7 `-OPT:...`

The `-OPT` command line option controls general optimizations that are not associated with a distinct compiler phase.

The `-OPT:roundoff=`*n* option controls floating-point accuracy and the behavior of overflow and underflow exceptions relative to the source language rules.

When `-O3` is in effect, the default rounding setting is `-OPT:roundoff=2`. This setting allows transformations with extensive effects on floating-point results. It allows associative rearrangement across loop iterations and the distribution of multiplication over addition and subtraction. It disallows only transformations known to cause overflow, underflow, or cumulative round-off errors for a wide range of floating-point operands.

At `-OPT:roundoff=2` or `3`, APO can change the sequence of a loop's floating-point operations in order to parallelize it. Because floating-point operations have finite precision, this change can cause slightly different results. If you want to avoid these differences by not having such loops parallelized, you must compile with `-OPT:roundoff=0` or `-OPT:roundoff=1`.

Example. APO parallelizes the following loop when compiled with the default settings of `-OPT:roundoff=2` and `-O3`:

```
REAL A, B(100)
DO I = 1, 100
    A = A + B(I)
END DO
```

At the start of the loop, each processor gets a private copy of `A` in which to hold a partial sum. At the end of the loop, the partial sum in each processor's copy is added to the total in the original, global copy. This value of `A` can be different from the value generated by a version of the loop that is not parallelized.

For more information on the `-OPT` option, see Section 2.57, page 45.

9.1.8 `-pfa` and `-pfalist`

The `-pfa` option invokes APO. The `-pfalist` option produces a listing.

> **Note:** These options are outmoded. The preferred way of invoking APO is through the `-apo` option, and the preferred way to obtain a listing is through the `-apolist` option.

9.1.9 *file*

Your input file.

For information on files used and generated when APO is enabled, see Section 9.2. For information on Fortran input files, see Section 2.76, page 66.

9.2 Files

APO provides a number of options to generate listings that describe where parallelization failed and where it succeeded. You can use these listings to identify constructs that inhibit parallelization. When you remove these constructs, you can often improve program performance dramatically.

When looking for loops to run in parallel, focus on the areas of the code that use most of the execution time. To determine where the program spends its execution time, you can use tools such as SpeedShop and the ProDev ProMP Parallel Analyzer View described in the *ProDev ProMP User's Guide*.

The following sections describe the content of the files generated by APO.

9.2.1 The *file*`.list` File

The `-apolist` and `-apokeep` options generate files that list the original loops in the program along with messages indicating if the loops were parallelized. For loops that were not parallelized, an explanation is provided.

Example. The following subroutine resides in file `testl.f`:

```
SUBROUTINE SUB(ARR, N)
REAL(KIND=8), DIMENSION(N) :: ARR
INTEGER :: N, I

ARR(2:N) = ARR(1:N-1) + ARR(2:N)

DO I = 1, N
  ARR(I) = ARR(I) + 7.0
  CALL FOO(A)
END DO

ARR = ARR + 7.0

END
```

The preceding code produces the following APO list file:

```
Parallelization Log for Subprogram sub_
5: Not Parallel
     Array dependence from ARR on line 5 to ARR on line 5.

7: Not Parallel
     Call foo_ on line 9.

12: PARALLEL (Auto) __mpdo_sub_1
```

9.2.2 The *file*`.w2f.f` File

The `-flist` option generates *file*`.w2f.f`. File *file*`.w2f.f` contains code that mimics the behavior of programs after they undergo automatic parallelization. The representation is designed to be readable so that you can see what portions of the original code were not parallelized. You can use this information to change the original program.

The compiler creates *file*`.w2f.f` by invoking the appropriate translator to turn the compiler's internal representations into FORTRAN 77 (not Fortran 95). In most cases, the files contain valid code that can be recompiled, although compiling *file*`.w2f.f` without APO enabled does not produce object code that is exactly the same as that generated when APO is enabled on the original source.

By default, the parallelized program in *file*`.w2f.f` uses OpenMP directives. To generate a parallelized program that uses the outmoded MIPS multiprocessing directives, described in Appendix D, page 267, specify `-FLIST:emit_omp=OFF`.

Example. File `testw2.f` is compiled with the following command:

```
f90 -O3 -n32 -mips4 -c -apo -apokeep testw2.f
```

```
      SUBROUTINE INIT(A)
REAL(KIND=4), DIMENSION(10000) :: A

A = 0.0

END
```

Compiling `testw2.f` generates an object file, `testw2.o`, and listing file `testw2.w2f.f`, which contains the following code:

```
C **********************************
C Fortran file translated from WHIRL
C **********************************

CSGI$ start 1
        SUBROUTINE init(A)
        IMPLICIT NONE
        REAL(4) A(10000_8)
C
C       **** Temporary variables ****
C
        INTEGER(4) f90li_0_1
C
C       **** statements ****
C
C       PARALLEL DO will be converted to SUBROUTINE __mpdo_init_1
CSGI$ start 2
C$OMP PARALLEL DO private(f90li_0_1), shared(A)
        DO f90li_0_1 = 0, 9999, 1
          A(f90li_0_1 + 1) = 0.0
        END DO
CSGI$ end 2
        RETURN
```

```
                  END
CSGI$ end 1
```

Note: WHIRL is the name for the compiler's intermediate representation. It is written in the style of the FORTRAN 77 standard, not the Fortran 95 standard.

As explained in Section 9.2.1, page 219, parallel versions of loops are put in their own subroutines. In this example, that subroutine is `__mpdo_init_1`. `C$OMP PARALLEL DO` is an OpenMP directive that specifies a parallel region containing a single `DO` directive.

9.2.3 About the `.m` and `.anl` Files

The `-apokeep` option generates *file*`.list`. It also generates *file*`.m` and *file*`.anl`, which are used by Workshop Pro MP.

file`.m` is similar to the *file*`.w2f.f` file; it is based on OpenMP and mimics the behavior of the program after automatic parallelization.

ProDev ProMP is a Silicon Graphics product that provides a graphical interface to aid in both automatic and manual parallelization for Fortran. The ProDev ProMP Parallel Analyzer View helps you understand the structure and parallelization of multiprocessing applications by providing an interactive, visual comparison of their original source with transformed, parallelized code. For more information, see the *ProDev ProMP User's Guide* and the *Developer Magic: Performance Analyzer User's Guide*.

SpeedShop, another Silicon Graphics product, allows you to run experiments and generate reports to track down the sources of performance problems. SpeedShop includes a set of commands and a number of libraries to support the commands. For more information, see the *SpeedShop User's Guide*.

Note: The code in *file*`.m` is written in the style of the FORTRAN 77 standard, not the Fortran 95 standard.

9.3 Running Your Program

You invoke a parallelized version of your program using the same command line as a sequential one. The same binary output file can be executed on various numbers of processors. The default is to have the run-time environment select the number of processors to use based on how many are available.

You can change the default behavior by setting the `OMP_NUM_THREADS` environment variable, which tells the system to use a particular number of processors. The following statement causes the program to create two threads regardless of the number of processors available:

```
setenv OMP_NUM_THREADS 2
```

The `OMP_DYNAMIC` environment variable allows you to control whether the run-time environment should dynamically adjust the number of threads available for executing parallel regions to optimize use of system resources. The default value is `TRUE`. If `OMP_DYNAMIC` is set to `FALSE`, dynamic adjustment is disabled.

For more information on these and other environment variables, see the `pe_environ`(5) man page.

9.4 Troubleshooting Incomplete Optimizations

Some loops cannot be safely parallelized and others are written in ways that inhibit APO's efficiency. The following subsections describe the steps you can take to make APO more effective:

- Section 9.4.1, page 222, describes constructs that inhibit parallelization.
- Section 9.4.2, page 226, describes constructs that inhibit APO's effectiveness.

9.4.1 Constructs That Inhibit Parallelization

A program's performance can be severely constrained if APO cannot recognize that a loop is safe to parallelize. APO analyzes every loop in a program. If a loop does not appear safe, it does not parallelize that loop. The following sections describe constructs that can inhibit parallelization:

- Section 9.4.1.1, page 223, describes basic data dependencies.
- Section 9.4.1.2, page 224, describes function calls.
- Section 9.4.1.3, page 224, describes `GO TO` statements.
- Section 9.4.1.4, page 224, describes problematic array subscripts.
- Section 9.4.1.5, page 225, describes conditionally assigned local variables.

In many instances, loops containing the previous constructs can be parallelized after minor changes. Reviewing the information generated in program *file*`.list`, described in Section 9.2.1, page 219, can show you if any of these constructs are in your code.

9.4.1.1 Loops Containing Data Dependencies

Generally, a loop is safe if there are no data dependencies, such as a variable being assigned in one iteration of a loop and used in another. APO does not parallelize loops for which it detects data dependencies.

For example, APO cannot parallelize loop `I` in the following subroutine because it contains a data dependence on variable `X`:

```
      SUBROUTINE SUB(N, A, B)
INTEGER :: I, N
REAL :: X, A(N), B(N)

X = 0.0
DO I = 1, N
  A(I) = X
  IF (I .GT. N / 2) X = 1.0
END DO

END
```

Many times, such dependences can be removed by making simple modifications to the source code. In this case, we can assign to `X` in each iteration before we read `X`, as follows:

```
SUBROUTINE SUB(N, A, B)
INTEGER :: I, N
REAL :: X, A(N), B(N)

DO I = 1, N
  IF (I .LE. N / 2) THEN
    X = 0.0
  ELSE
    X = 1.0
  END IF
  A(I) = X
END DO

END
```

APO now can parallelize loop `I`.

9.4.1.2 Loops Containing Function Calls

By default, APO does not parallelize a loop that contains a function call because the function in one iteration of the loop can modify or depend on data in other iterations.

You can, however, use interprocedural analysis (IPA) to provide APO with enough information to parallelize some loops containing subroutine calls by inlining those calls. IPA is specified by the `-IPA` command line option. For more information on IPA, see the `ipa`(5) man page and the *MIPSpro Compiling and Performance Tuning Guide*.

You can also direct APO to ignore function call dependencies when analyzing the specified loops by using the `!*$* ASSERT CONCURRENT CALL` directive described in Section 9.5.1, page 232.

9.4.1.3 Loops Containing `GO TO` Statements

`GO TO` statements are unstructured control flows. APO converts most unstructured control flows in loops into structured flows that can be parallelized. However, `GO TO` statements in loops can still cause the following problems:

- Unstructured control flows. APO is unable to restructure all types of flow control in loops. You must either restructure these control flows or manually parallelize the loops containing them.
- Early exits from loops. Loops with early exits cannot be parallelized, either automatically or manually.

For improved performance, remove `GO TO` statements from loops to be considered candidates for parallelization.

9.4.1.4 Loops Containing Problematic Array Constructs

The following array constructs inhibit parallelization and should be removed whenever APO is used:

- Arrays with subscripts that are indirect array references. APO cannot analyze indirect array references. The following loop cannot be run safely in parallel if the indirect reference `IB(I)` is equal to the same value for different iterations of `I`:

```
DO I = 1, N
   A(IB(I)) = ...
```

```
END DO
```

If every element of array `IB` is unique, the loop can safely be made parallel. To achieve automatic parallelism in such cases, use the `!*$* ASSERT PERMUTATION` directive, discussed in Section 9.5.5, page 236.

- Arrays with unanalyzable subscripts. APO cannot parallelize loops containing arrays with unanalyzable subscripts. Allowable subscripts can contain the following elements:
  - Literal constants (`1`, `2`, `3`, ...)
  - Variables (`I`, `J`, `K`, ...)
  - The product of a literal constant and a variable, such as `N*5` or `K*32`
  - A sum or difference of any combination of the first three items, such as `N*21+K-251`

  In the following case, APO cannot analyze the division operator (`/`) in the array subscript and cannot reorder the loop:

  ```
  DO I = 2, N, 2
      A(I/2) = ...
  END DO
  ```

- Unknown information. In the following example there may be hidden knowledge about the relationship between variables `M` and `N`:

  ```
  DO I = 1, N
      A(I) = A(I+M)
  END DO
  ```

  The loop can be run in parallel if `M > N` because the array reference does not overlap. However, APO does not know the value of the variables and therefore cannot make the loop parallel. You can use the `!*$* ASSERT DO (CONCURRENT)` directive to have APO automatically parallelize this loop. For more information on this directive, see Section 9.5.2, page 234.

9.4.1.5 Loops Containing Local Variables

When parallelizing a loop, APO often localizes (privatizes) temporary scalar and array variables by giving each processor its own nonshared copy of them. In the following example, array `TMP` is used for local scratch space:

```
DO I = 1, N
    DO J = 1, N
      TMP(J) = ...
    END DO
    DO J = 1, N
      A(J,I) = A(J,I) + TMP(J)
    END DO
END DO
```

To successfully parallelize the outer loop (`I`), APO must give each processor a distinct, private copy of array `TMP`. In this example, it is able to localize `TMP` and, thereby, to parallelize the loop.

APO cannot parallalize a loop when a conditionally assigned temporary variable might be used outside of the loop, as in the following example:

```
SUBROUTINE S1(A, B)
    COMMON T
    ...
    DO I = 1, N
      IF (B(I)) THEN
        T = ...
        A(I) = A(I) + T
      END IF
    END DO
    CALL S2()
END
```

If the loop were to be run in parallel, a problem would arise if the value of `T` were used inside subroutine `S2()` because it is not known which processor's private copy of `T` should be used by `S2()`. If `T` were not conditionally assigned, the processor that executed iteration `N` would be used. Because `T` is conditionally assigned, APO cannot determine which copy to use.

The solution comes with the realization that the loop is inherently parallel if the conditionally assigned variable `T` is localized. If the value of `T` is not used outside the loop, replace `T` with a local variable. Unless `T` is a local variable, APO assumes that `S2()` might use it.

9.4.2 Constructs That Slow Down Parallelized Code

APO parallelizes a loop by distributing its iterations among the available processors. Loop nesting, loops with low trip counts, and other program characteristics can affect the efficiency of APO. The following sections describe the effect that these and other programming constructs can have on APO's ability to parallelize:

- Section 9.4.2.1, page 227, describes parallelizing nested loops.
- Section 9.4.2.2, page 228, describes parallelizing loops with small or indeterminate trip counts.
- Section 9.4.2.3, page 229, describes parallelizing loops that exhibit poor data locality.

9.4.2.1 Parallelizing Nested Loops

APO can parallelize only one loop in a loop nest. In these cases, the most effective optimization usually occurs when the outermost loop is parallelized. The effectiveness derives from that fact that more processors end up processing larger sections of the program. This saves synchronization and other overhead costs.

Example 1. Consider the following simple loop nest:

```
DO I = 1, L
    ...
    DO J = 1, M
      ...
      DO K = 1, N
        ...
```

When parallelizing nested loops `I`, `J`, and `K`, APO distributes only one of the loops. Effective loop nest parallelization depends on the loop that APO chooses, but it is possible for APO to choose an inferior loop to be parallelized. APO may attempt to interchange loops to make a more promising one the outermost. If the outermost loop attempt fails, APO attempts to parallelize an inner loop. Because of the potential for improved performance, it is useful for you to modify your code so that the outermost loop is the one parallelized.

Section 9.2.1, page 219, describes *file*`.list`. This output file contains information that tells you which loop in a nest was parallelized.

For every loop that could be parallelized, APO generates a test to determine whether the loop is being called from within either another parallel loop or from within a parallel region. In some cases, you can minimize the extra testing that APO must perform by inserting directives into your code to inhibit parallization testing. The following example demonstrates this:

Example 2:

```
SUBROUTINE CALLER
    DO I = 1, N
      CALL SUB
    END DO
    ...
END
SUBROUTINE SUB
    ...
    DO I = 1, N
      ...
    END DO
END
```

If the loop inside `CALLER()` is parallelized, the loop inside `SUB()` cannot be run in parallel when `CALLER()` calls `SUB()`. In this case, the test can be avoided.

If `SUB()` is always called from `CALLER()`, you can use the `!*$* ASSERT DO (SERIAL)` directive to force the sequential execution of the loop in `SUB()`. With the addition of the directive, the subroutine would be written as follows:

```
      SUBROUTINE CALLER
          DO I = 1, N
            CALL SUB
          END DO
          ...
      END
      SUBROUTINE SUB
          ...
!*$* ASSERT DO (SERIAL)
          DO I = 1, N
            ...
          END DO
      END
```

For more information on this compiler directive, see Section 9.5.3, page 234.

9.4.2.2 Parallelizing Loops with Small or Indeterminate Trip Counts

The *trip count* is the number of times a loop is executed. Loops with large trip counts are the best candidates for parallelization. The following paragraphs show how to modify your program if your program contains loops with small trip counts or loops with indeterminate trip counts:

- Loops with small trip counts generally run faster when they are not parallelized. Consider the following loop nest:

  ```
  DO I = 1, M
      DO J = 1, N
  ```

  APO may try to parallelize loop `I` because it is outermost. If `M` is very small, it would be better to interchange the loops and make loop `J` outermost before parallelization. Because APO often cannot know that `M` is small, you can use a `!*$* ASSERT DO PREFER (CONCURRENT)` directive to indicate to APO that it is better to parallelize loop `J`, as follows:

  ```
        DO I = 1, M
  !*$* ASSERT DO PREFER (CONCURRENT)
            DO J = 1, N
  ```

- If the trip count is not known (and sometimes even if it is), APO parallelizes the loop conditionally, generating code for both a parallel and a sequential version. By generating two versions, APO can avoid running a loop in parallel that may have small trip count. APO chooses the version to use based on the trip count, the code inside the loop's body, the number of processors available, and an estimate of the cost to invoke a parallel loop in that runtime environment.

  You can avoid the overhead incurred by having APO generate both sequential and parallel versions of a loop by using the `!*$* ASSERT DO PREFER (SERIAL)` directive.

9.4.2.3 Parallelizing Loops with Poor Data Locality

Computer memory has a hierarchical organization. Higher up the hierarchy, memory becomes closer to the CPU, faster, more expensive, and more limited in size. Cache memory is at the top of the hierarchy, and main memory is further down in the hierarchy. In multiprocessor systems, each processor has its own cache memory. Because it is time consuming for one processor to access another processor's cache, a program's performance is best when each processor has the data it needs in its own cache.

Programs, especially those that include extensive looping, often exhibit *locality of reference*, which means that if a memory location is referenced, it is probable that it or a nearby location will be referenced in the near future. Loops designed to take advantage of locality do a better job of concentrating data in memory, increasing the probability that a processor will find the data it needs in its own cache.

The following examples show the effect of locality on parallelization. Assume that the loops are to be parallelized and that there are *p* processors.

Example 1. Distribution of Iterations.

```
DO I = 1, N
    ...A(I)
END DO
DO I = N, 1, -1
    ...A(I)...
END DO
```

In the first loop, the first processor accesses the first `N/`*p* elements of `A`; the second processor accesses the next `N/`*p* elements; and so on. In the second loop, the distribution of iterations is reversed. That is, the first processor accesses the last `N/`*p* elements of `A`, and so on. Most elements are not in the cache of the processor needing them during the second loop. This code fragment would run more efficiently, and be a better candidate for parallelization, if you reverse the direction of one of the loops.

Example 2. Two Nests in Sequence.

```
DO I = 1, N
    DO J = 1, N
      A(I,J) = B(J,I) + ...
    END DO
END DO

DO I = 1, N
    DO J = 1, N
      B(I,J) = A(J,I) + ...
    END DO
END DO
```

In example 2, APO may parallelize the outer loop of each member of a sequence of nests. If so, while processing the first nest, the first processor accesses the first `N/`*p* rows of `A` and the first `N/`*p* columns of `B`. In the second nest, the first processor accesses the first `N/`*p* columns of `A` and the first `N/`*p* rows of `B`. This example runs much more efficiently if you parallelize the `I` loop in one nest and the `J` loop in the other. You can instruct APO to do this with the `!*$* ASSERT DO PREFER (CONCURRENT)` directive, as follows:

```
      DO I = 1, N
!*$* ASSERT DO PREFER (CONCURRENT)
          DO J = 1, N
```

```
           A(I,J) = B(J,I) + ...
         END DO
     END DO

!*$* ASSERT DO PREFER (CONCURRENT)
     DO I = 1, N
         DO J = 1, N
           B(I,J) = A(J,I) + ...
         END DO
     END DO
```

9.5 Compiler Directives

APO works in conjunction with the OpenMP Fortran API directives described in Chapter 4, page 89, and with the Origin series directives described in Chapter 5, page 143. You can use these directives to manually parallelize some loop nests, while leaving others to APO. This approach has the following positive and negative aspects:

- As a positive aspect, the OpenMP and Origin series directives are well defined and deterministic. If you use a directive, the specified loop is run in parallel. This assumes that the trip count is greater than one and that the specified loop is not nested in another parallel loop.
- The negative side to this is that you must carefully analyze the code to determine that parallelism is safe. Also, you must mark all private variables.

In addition to the OpenMP and Origin series directives, you can also use the APO-specific directives described in this section. These directives give APO more information about your code.

Note: APO also recognizes the Silicon Graphics multiprocessing directives described in Appendix D, page 267. These directives are outmoded. The OpenMP directive set is the preferred directive set for multiprocessing. You must include the `-mp` option on the `f90`(1) command line in order for the compiler to recognize the Silicon Graphics multiprocessing directives.

The APO directives can affect certain optimizations, such as loop interchange, during the compiling process. To direct the compiler to disregard any of the preceding directives, use the `-x`*dirlist* option described in Section 2.74, page 65.

The APO directives are as follows:

- `!*$* ASSERT CONCURRENT CALL`. This directive directs APO to ignore dependencies in subroutine calls that would inhibit parallelization. For more information on this directive, see Section 9.5.1, page 232.
- `!*$* ASSERT DO (CONCURRENT)`. This directive asserts that APO should not let perceived dependencies between two references to the same array inhibit parallelizing. For more information on this directive, see Section 9.5.2, page 234.
- `!*$* ASSERT DO (SERIAL)`. This directive requests that the following loop be executed in serial mode. For more information on this directive, see Section 9.5.3, page 234.
- `!*$* ASSERT DO PREFER (CONCURRENT)`. This directive parallelizes the following loop if it is safe. For more information on this directive, see Section 9.5.4, page 235.
- `!*$* ASSERT PERMUTATION (`*array_name*`)`. This directive asserts that array *array_name* is a permutation array. For more information on this directive, see Section 9.5.5, page 236.
- `!*$* NO CONCURRENTIZE` and `!*$* CONCURRENTIZE`. The `!*$* NO CONCURRENTIZE` directive inhibits either parallelization of all loops in a subroutine or parallelization of all loops in a file. The `!*$* CONCURRENTIZE` directive overrides the `!*$* NO CONCURRENTIZE` directive, and its effect varies with its placement. For more information on these directives, see Section 9.5.6, page 237.

Note: The compiler honors the following APO directives even if the `-apo` option is not included on your command line:

- `!*$* ASSERT CONCURRENT CALL`
- `!*$* ASSERT DO (CONCURRENT)`
- `!*$* ASSERT PERMUTATION (`*array_name*`)`

9.5.1 `!*$* ASSERT CONCURRENT CALL`

The `!*$* ASSERT CONCURRENT CALL` directive instructs APO to ignore the dependencies of subroutine and function calls contained in the loop that follows the assertion. The directive applies to the loop that immediately follows it and to all loops nested inside that loop.

Note: The directive affects the compilation even when `-apo` is not specified.

APO ignores the dependencies in subroutine `FRED()` when it analyzes the following loop:

```
!*$* ASSERT CONCURRENT CALL
         DO I = 1, N
           CALL FRED
           ...
         END DO
         SUBROUTINE FRED
           ...
         END
```

To prevent incorrect parallelization, make sure the following conditions are met when using `!*$* ASSERT CONCURRENT CALL`:

- A subroutine inside the loop cannot read from a location that is written to during another iteration. This rule does not apply to a location that is a local variable declared inside the subroutine.
- A subroutine inside the loop cannot write to a location that is read from or written to during another iteration. This rule does not apply to a location that is a local variable declared inside the subroutine.

Example. The following code shows an illegal use of the directive. Subroutine `FRED()` writes to variable `T`, which is also read from by `WILMA()` during other iterations:

```
!*$* ASSERT CONCURRENT CALL
         DO I = 1,M
           CALL FRED(B, I, T)
           CALL WILMA(A, I, T)
         END DO
         SUBROUTINE FRED(B, I, T)
           REAL B(*)
           T = B(I)
         END
         SUBROUTINE WILMA(A, I, T)
           REAL A(*)
           A(I) = T
         END
```

By localizing the variable `T`, you can manually parallelize the preceding example safely. However, APO does not know to localize `T`, so it illegally parallelizes the loop because of the directive.

9.5.2 `!*$* ASSERT DO (CONCURRENT)`

The `!*$* ASSERT DO (CONCURRENT)` directive instructs APO, when analyzing the loop immediately following this directive, to ignore all dependencies between two references to the same array. If there are real dependencies between array references, the `!*$* ASSERT DO (CONCURRENT)` directive can cause APO to generate incorrect code.

Note: This directive affects the compilation even when `-apo` is not specified.

The following example shows correct use of this directive when `M > N`:

```
!*$* ASSERT DO (CONCURRENT)
         DO I = 1, N
           A(I) = A(I+M)
```

Be aware of the following points when using this directive:

- If multiple loops in a nest can be parallelized, `!*$* ASSERT DO (CONCURRENT)` causes APO to parallelize the loop immediately following the assertion.
- Applying this directive to an inner loop can cause the loop to be made outermost by APO's loop interchange operations.
- This directive does not affect how APO analyzes `CALL` statements. For more information on APO's interaction with `CALL` statements, see Section 9.5.1, page 232.
- This directive does not affect how APO analyzes dependencies between two potentially aliased pointers.
- The compiler may find some obvious real dependencies. If it does so, it ignores this directive.

9.5.3 `!*$* ASSERT DO (SERIAL)`

The `!*$* ASSERT DO (SERIAL)` directive instructs APO not to parallelize the loop following the assertion; the loop is executed in serial mode. APO can, however, parallelize another loop in the same nest. The parallelized loop can be either inside or outside the designated sequential loop.

Note: This directive has the same effect as the `!*$* ASSERT DO PREFER (SERIAL)` directive. In order for the `!*$* ASSERT DO PREFER (SERIAL)` directive to be honored, however, the `-apo` option must appear on the `f90`(1) command line. The `!*$* ASSERT DO PREFER (SERIAL)` directive is outmoded.

The `!*$* ASSERT DO (SERIAL)` directive affects the compilation even when the `-apo` option is not specified.

Example. The following code fragment contains a directive that requests that loop `J` be run serially:

```
         DO I = 1, M
!*$* ASSERT DO (SERIAL)
           DO J = 1, N
             A(I,J) = B(I,J)
           END DO
           ...
         END DO
```

The directive applies only to the loop that immediately follows it. For example, APO still tries to parallelize loop `I`. This directive is useful in cases like this when the value of `N` is known to be very small.

9.5.4 !*$* ASSERT DO PREFER (CONCURRENT)

The `!*$* ASSERT DO PREFER (CONCURRENT)` directive instructs APO to parallelize the loop immediately following the directive if it is safe to do so.

Example. The following code fragment encourages APO to run loop `I` loop in parallel:

```
!*$* ASSERT DO PREFER (CONCURRENT)
         DO I = 1, M
           DO J = 1, N
             A(I,J) = B(I,J)
           END DO
           ...
         END DO
```

When dealing with nested loops, APO follows these guidelines:

- If the loop specified by the `!*$* ASSERT DO PREFER (CONCURRENT)` directive is safe to parallelize, APO parallelizes the specified loop even if other loops in the nest are safe.

- If the specified loop is not safe to parallelize, APO parallelizes a different loop that is safe.
- If this directive is applied to an inner loop, APO can interchange the loop and make the specified loop the outermost loop.
- If this directive is applied to more than one loop in a nest, APO parallelizes one of the specified loops.

9.5.5 `!*$* ASSERT PERMUTATION` (*array_name*)

When placed inside a subroutine, the `!*$* ASSERT PERMUTATION (`*array_name*`)` directive informs APO that *array_name* is a permutation array. A *permutation array* is one in which every element of the array has a distinct value.

This directive does not require the permutation array to be *dense*. That is, within the array, every `IB(I)` must have a distinct value, but there can be gaps between the values, such as `IB(1)` = 1, `IB(2)` = 4, `IB(3)` = 9, and so on.

Note: This directive affects compilation even when `-apo` is not specified.

Example. In the following code fragment, array `IB` is declared to be a permutation array for both loops in `SUB1()`:

```
        SUBROUTINE SUB1
          DO I = 1, N
            A(IB(I)) = ...
          END DO
 !*$* ASSERT PERMUTATION (IB)
          DO I = 1, N
            A(IB(I)) = ...
          END DO
        END
```

Note the following points about this directive:

- As shown in the example, you can use this directive to parallelize loops that use arrays for indirect addressing. Without this directive, APO cannot determine that the array elements used as indexes are distinct.
- `!*$* ASSERT PERMUTATION (`*array_name*`)` affects every loop in a subroutine, even those that appear ahead of it.

9.5.6 `!*$* NO CONCURRENTIZE` and `!*$* CONCURRENTIZE`

The `!*$* NO CONCURRENTIZE` and `!*$* CONCURRENTIZE` directives toggle parallelization. Their effects depend on their placement.

- When placed inside subroutines and functions, `!*$* NO CONCURRENTIZE` inhibits parallelization. In the following example, no loops inside `SUB1()` are parallelized:

```
         SUBROUTINE SUB1
!*$* NO CONCURRENTIZE
           ...
         END
```

- When placed outside of a subroutine, `!*$* NO CONCURRENTIZE` prevents the parallelization of all procedures in the file, even those that appear ahead of it in the file. Loops inside subroutines `SUB2()` and `SUB3()` are not parallelized in the following example:

```
         SUBROUTINE SUB2
           ...
         END
!*$* NO CONCURRENTIZE
         SUBROUTINE SUB3
           ...
         END
```

The `!*$* CONCURRENTIZE` directive, when placed inside a subroutine, overrides a `!*$* NO CONCURRENTIZE` directive that is placed outside of it. Thus, this directive allows you to selectively parallelize subroutines in a file that has been made sequential with a `!*$* NO CONCURRENTIZE` directive.

Libraries [A]

The MIPSpro 7 Fortran 90 compiler works with the following other commands, intrinsic procedures, and library routines:

- The `assign`(1) command. This command can be used to alter the details of a Fortran file connection, such as device residency, alternative file names, or file space allocations. The `assign`(1) options are associated with file names, file name patterns, or unit numbers. When associated with file names or file name patterns, the options are applied whenever a matching file name is opened from a Fortran program. When associated with a unit number, the options are applied whenever that unit becomes connected.

  For complete details about the `assign` command, see the `assign`(1) man page or the *Application Programmer's I/O Guide*.

- The Flexible File I/O (FFIO) system. This system lets you specify a comma-separated list of layers through which I/O data will be passed. The FFIO layers act as filters that manipulate the data file as it is being read or written. The layers include performance options and the capability to read and write files in different vendors' blocking formats. For more information on FFIO, see the `intro_ffio`(3F) man page and the *Application Programmer's I/O Guide*.

- Intrinsic procedures. These procedures are predefined by the computer programming language. They are invoked in the same way that other procedures are invoked. The Fortran standard defines intrinsic procedures, and the MIPSpro 7 Fortran 90 compiler includes other intrinsics as extensions to the standard.

  For details about the available intrinsic procedures, see the following publications: the *Intrinsic Procedures Reference Manual*, or the *Fortran Language Reference Manual, Volume 2*.

- POSIX library routines. The POSIX FORTRAN 77 Language Interfaces Standard IEEE Std 1003.9-1992 (POSIX.a) defines a standardized interface for accessing the system services of IEEE Std 1003.1-1990 (POSIX.1) and supports routines to access constructs not directly accessible with FORTRAN 77. These routines can also be used by Fortran 90/95 programs. For more information on these routines, see the `intro_pxf`(3F) man page.

- Miscellaneous library routines. A library is a collection of subprograms, usually grouped around a specific subject, such as input and output (I/O). You can call library routines explicitly in your program, or they can be called by the compiler. The following sections describe the library routines that are available to you.

A.1 Miscellaneous Library Routines

The following list describes the library routines that are available with the MIPSpro 7 Fortran 90 compiler. See the individual man pages for more details.

- FFIO routines (C routines used with the FFIO layers):
  - `fffcntl`(3C)
  - `ffopen`(3C)
  - `ffpos`(3C)
  - `ffread`(3C)
  - `ffseek`(3C)
- Interface routines (job control routines that control program terminations or execute a shell command):
  - `ABORT`(3F)
  - `EXIT`(3F)
  - `ISHELL`(3F)
- I/O routines to control input and output:
  - `ASNCTL`(3F)
  - `ASNQFILE`(3F)
  - `ASSIGN`(3F)
  - `FLUSH`(3F)
  - `NUMBLKS`(3F)
  - `RNL`(3F)
  - `RNLECHO`(3F)
  - `RNLSKIP`(3F)
  - `RNLTYPE`(3F)

- `WNL`(3F)
- `WNLLINE`(3F)
- `WNLLONG`(3F)

- Programming aids (routines for times and dates, packing and unpacking, and character argument counters):
  - `SECOND`(3F)
  - `SECONDR`(3F)
  - `SYSCLOCK`(3F)
  - `TIMEF`(3F)
- Multiprocessing routines for Fortran. There are a suite of routines developed specifically for multiprocessing. For information on these routines, see the `mp`(3F) man page.

A.2 Library Functions

The Fortran library routines provide an interface from Fortran programs to the IRIX system functions. System functions are facilities that are provided by the IRIX system kernel directly, as opposed to functions that are supplied by library code loaded with your program.

Table 3 summarizes the routines in the Fortran run-time library that can be used with the compiler. The table indicates PXF POSIX Library standard routines as recommended substitutions for IRIX system functions. See the individual man pages for details about each routine.

Table 3. Summary of System Interface Library Routines

Function	Recommended	Purpose
`abort`		Abnormal termination
`access`	`PXFACCESS`	Determine accessibility of a file
`acct`		Enable/disable process accounting
`alarm`	`PXFALARM`	Execute a subroutine after a specified time

Function	Recommended	Purpose
`barrier`		Perform barrier operations
`blockproc`		Block processes
`brk`		Change data segment space allocation
`chdir`	`PXFCHDIR`	Change default directory
`chmod`	`PXFCHMOD`	Change mode of a file
`chown`	`PXFCHOWN`	Change owner
`chroot`	`PXFCHROOT`	Change root directory for a command
`close`		Close a file descriptor
`creat`	`PXFCREAT`	Create or rewrite a file
`ctime`		Return system time
`dtime`		Return elapsed execution time
`dup`		Duplicate an open file descriptor
`etime`		Return elapsed execution time
`exit`	`PXFFASTEXIT`	Terminate process with status
`fcntl`		File control
`fdate`		Return date and time in an ASCII string
`fgetc`		Get a character from a logical unit
`fork`	`PXFFORK`	Create a copy of this process
`fputc`		Write a character to a Fortran logical unit
`free_barrier`		Free barrier
`fseek`		Reposition a file on a logical unit
`fseek64`		Reposition a file on a logical unit for 64-bit architecture
`fstat`		Get file status
`ftell`		Reposition a file on a logical unit
`ftell64`		Reposition a file on a logical unit for 64-bit architecture
`gerror`		Get system error messages
`getarg`	`PXFGETARG`	Return command line arguments
`getc`		Get a character from a logical unit

Function	Recommended	Purpose
getcwd	PXFGETCWD	Get pathname of current working directory
getdents		Read directory entries
getegid	PXFGETEGID	Get effective group ID
gethostid		Get unique identifier of current host
getenv	PXFGETENV	Get value of environment variables
geteuid	PXFGETEUID	Get effective user ID
getgid	PXFGETGID	Get user or group ID of the caller
gethostname		Get current host ID
getlog		Get user's login name
getpgrp	PXFGETPGRP	Get process group ID
getpid	PXFGETPID	Get process ID
getppid	PXFGETPPID	Get parent process ID
getsockopt		Get options on sockets
getuid	PXFGETUID	Get user or group ID of caller
gmtime		Return system time
iargc	IPXFARGC	Return command line arguments
idate		Return date or time in numerical form
ierrno		Get system error messages
ioctl		Control device
isatty	PXFISATTY	Determine if unit is associated with tty
itime		Return date or time in numerical form
kill	PXFKILL	Send a signal to a process
link	PXFLINK	Make a link to an existing file
loc		Return the address of an object
lseek		Move read/write file pointer
lseek64		Move read/write file pointer for 64-bit architecture
lstat		Get file status
ltime		Return system time

Function	Recommended	Purpose
m_fork		Create parallel processes
m_get_myid		Get task ID
m_get_numprocs		Get number of subtasks
m_kill_procs		Kill process
m_lock		Set global lock
m_next		Return value of counter
m_park_procs		Suspend child processes
m_rele_procs		Resume child processes
m_set_procs		Set number of subtasks
m_sync		Synchronize all threads
m_unlock		Unset a global lock
mkdir		Make a directory
mknod		Make a directory/file
mount		Mount a filesystem
new_barrier		Initialize a barrier structure
nice		Lower priority of a process
open	PXFOPEN	Open a file
oserror		Get/set system error
pause	PXFPAUSE	Suspend process until signal
perror		Get system error messages
pipe		Create an interprocess channel
plock		Lock process, test, or data in memory
prctl		Control processes
profil		Execution-time profile
ptrace		Process trace
putc		Write a character to a Fortran logical unit
putenv		Set environment variable
qsort		Quick sort

Function	Recommended	Purpose
`read`		Read from a file descriptor
`readlink`		Read value of symbolic link
`rename`	`PXFRENAME`	Change the name of a file
`rmdir`	`PXFRMDIR`	Remove a directory
`sbrk`		Change data segment space allocation
`schedctl`		Call to scheduler control
`send`		Send a message to a socket
`setblockproccnt`		Set semaphore count
`setgid`	`PXFSETGID`	Set group ID
`sethostid`		Set current host ID
`setoserror`		Set system error
`setpgrp`	`PXFSETPGRP`	Set process group ID
`setsockopt`		Set options on sockets
`setuid`	`PXFSETUID`	Set user ID
`sginap`		Put process to sleep
`sginap64`		Put process to sleep in 64-bit environment
`shmat`		Attach shared memory
`shmdt`		Detach shared memory
`sighold`		Raise priority and hold signal
`sigignore`		Ignore signal
`signal`		Change the action for a signal
`sigpause`		Suspend until receive signal
`sigrelse`		Release signal and lower priority
`sigset`		Specify system signal handling
`sleep`	`PXFSLEEP`	Suspend execution for an interval
`socket`		Create an endpoint for communication TCP
`sproc`		Create a new share group process
`stat`	`PXFSTAT`	Get file status

Function	Recommended	Purpose
`stime`		Set time
`symlink`		Make symbolic link
`sync`		Update superblock
`sysmp`		Control multiprocessing
`sysmp64`		Control multiprocessing in 64-bit environment
`system`		Issue a shell command
`taskblock`		Block tasks
`taskcreate`		Create a new task
`taskctl`		Control task
`taskdestroy`		Kill task
`tasksetblockcnt`		Set task semaphore count
`taskunblock`		Unblock task
`time`	`PXFTIME`	Return system time (must be declared EXTERNAL)
`ttynam`		Find name of terminal port
`uadmin`		Administrative control
`ulimit`		Get and set user limits
`ulimit64`		Get and set user limits in 64-bit architecture
`umask`	`PXFUMASK`	Get and set file creation mask
`umount`		Dismount a file system
`unblockproc`		Unblock processes
`unlink`	`PXFUNLINK`	Remove a directory entry
`uscalloc`		Shared memory allocator
`uscalloc64`		Shared memory allocator in 64-bit environment
`uscas`		Compare and swap operator
`usclosepollsema`		Detach file descriptor from a pollable semaphore
`usconfig`		Semaphore and lock configuration operations
`uscpsema`		Acquire a semaphore
`uscsetlock`		Unconditionally set lock

Function	Recommended	Purpose
`usctlsema`		Semaphore control operations
`usdumplock`		Dump lock information
`usdumpsema`		Dump semaphore information
`usfree`		User shared memory allocation
`usfreelock`		Free a lock
`usfreepollsema`		Free a pollable semaphore
`usfreesema`		Free a semaphore
`usgetinfo`		Exchange information through an arena
`usinit`		Semaphore and lock initialize routine
`usinitlock`		Initialize a lock
`usinitsema`		Initialize a semaphore
`usmalloc`		Allocate shared memory
`usmalloc64`		Allocate shared memory in 64-bit environment
`usmallopt`		Control allocation algorithm
`usnewlock`		Allocate and initialize a lock
`usnewpollsema`		Allocate and initialize a pollable semaphore
`usnewsema`		Allocate and initialize a semaphore
`usopenpollsema`		Attach a file descriptor to a pollable semaphore
`uspsema`		Acquire a semaphore
`usputinfo`		Exchange information through an arena
`usrealloc`		User share memory allocation
`usrealloc64`		User share memory allocation in 64-bit environment
`ussetlock`		Set lock
`ustestlock`		Test lock
`ustestsema`		Return value of semaphore
`usunsetlock`		Unset lock
`usvsema`		Free a resource to a semaphore
`uswsetlock`		Set lock

Function	Recommended	Purpose
`wait`	`PXFWAIT`	Wait for a process to terminate
`write`		Write to a file

A.3 Compatibility with `sproc`(2)

The parallelism used in Fortran is implemented using the `sproc`(2) system call. It is recommended that programs not attempt to use both `!$OMP PARALLEL DO` loops and `sproc` calls. It is possible, but there are several restrictions:

- Any threads you create may not execute `!$OMP PARALLEL DO` loops; only the original thread is allowed to do this.
- The calls to routines like `mp_block` and `mp_destroy` apply only to the threads created by `mp_create` or to those automatically created when the Fortran job starts; they have no effect on any user-defined threads.
- Calls to routines such as `m_get_numprocs` do not apply to the threads created by the Fortran routines. However, the Fortran threads are ordinary subprocesses; using the `kill` routine with the arguments `0` and `sig` (for example, `kill`(0,`sig`)) to signal all members of the process group might kill threads used to execute `!$OMP PARALLEL DO`. If you choose to intercept the `SIGCLD` signal, you must be prepared to receive this signal when the threads used for the `!$OMP PARALLEL DO` loops exit; this occurs when `mp_destroy` is called or at program termination.
- The `m_fork` call is implemented using `sproc`(2), so it is not legal to run `m_fork` on a family of processes that each subsequently executes `!$OMP PARALLEL DO` loops. Only the original thread can execute `!$OMP PARALLEL DO` loops.

Debugging and Profiling Multiprocessed Programs [B]

This appendix describes some aspects of debugging multiprocessed Fortran source code. The recommended debugger for use with the MIPSpro 7 Fortran 90 compiler is `dbx`(1). The `dbx`(1) debugger includes the following features to support the Fortran language: allocatable arrays, pointer-based variables, nonstandard stride arrays, modules, and derived types. For more information on this debugger, see the `dbx`(1) man page.

B.1 Setting Up Your Environment

When debugging a program with `dbx`(1), enter the following command:

```
% (dbx) ignore TERM
```

This command allows a multiprocessed program to terminate gracefully after execution is complete.

B.2 Profiling a Parallel Fortran Program

It is easiest to debug a program for execution on multiple processors in a single-processor environment. *After* your program executes successfully on a single processor, you can compile it for multiprocessing by using the `-mp` option on the `f90`(1) command line.

After converting a program from use on one processor to one that can be multiprocessed, you should examine execution profiles to judge the effectiveness of the transformation. Good profiles of the program are crucial to help you focus on the loops that use the most time. You can use SpeedShop to obtain these profiles. For more information on SpeedShop, see the *SpeedShop User's Guide* or the `ssrun`(1) man page.

If your job uses multiple threads, you can use SpeedShop to create multiple profile data files, one profile file for each thread. Use the `prof`(1) standard profile analyzer to examine this output. You can also use `timex`(1); this command indicates if the parallelized versions performed better overall than the serial version.

The profile of a Fortran parallel job is different from a standard profile. To produce a parallel program, the compiler pulls the parallel DO loops out into separate subroutines, one routine for each loop. Each of these loops is shown as a separate procedure in the profile. You can compare the amount of time spent in each loop by the various threads to determine how well the workload is balanced.

You can use par(1) to trace the activity of a single process, a related group of processes, or the system as a whole. The par(1) utility is a process activity reporter. For more information on par(1), see the par(1) man page.

In addition to the loops, the profile returned by the prof(1) command shows the special routines that actually do the multiprocessing. The __mp_parallel_do routine is the synchronizer and controller. Slave threads wait for work in the routine __mp_slave_wait_for_work; the less time they wait, the more time they work. This gives a rough estimate of the extent of parallelism in a program. For more information on these routines, see the mp(3F) man page.

B.3 Debugging Parallel Fortran

After you have isolated program bugs to one or two loops, you can begin to debug. To determine if a loop can be multiprocessed, change the order of the iterations on the parallel DO loop on a single-processor version. If the loop can be multiprocessed, the iterations can execute in any order and produce the same answer. If the loop cannot be multiprocessed, changing the order usually causes the single-processor version to fail. You can use single-process debugging techniques to determine the problem.

Example. Erroneous !$OMP PARALLEL DO. In this example, two references to A have the indexes in reverse order. If the indexes were in the same order (if both were A(I,J) or both were A(J,I)), the loop could be multiprocessed. As written, there is a data dependency, so the !$OMP PARALLEL DO is an error.

```
!$OMP PARALLEL DO PRIVATE(I,J)
        DO I = 1, N
           DO J = 1, N
              A(I,J) = A(J,I) + X*B(I)
           END DO
        END DO
```

Because a (correct) multiprocessed loop can execute its iterations in any order, you could rewrite this as:

```
!$OMP PARALLEL DO PRIVATE(I,J)
        DO I = N, 1, -1
           DO J = 1, N
              A(I,J) = A(J,I) + X*B(I)
           END DO
        END DO
```

This loop no longer gives the same answer as the original even when compiled without the `-mp` option. This reduces the problem to a normal debugging problem.

B.3.1 Other Debugging Tips for Multiprocessed Loops

If a multiprocessed loop produces the wrong answer, use the following checklist to determine the cause:

Item to investigate	Reasons
`PRIVATE` variables	Check the `PRIVATE` variables when the code runs correctly as a single process but fails when multiprocessed. Check any scalar variables that appear in the left-hand side of an assignment statement in the loop to be sure they are all declared as `PRIVATE`. Be sure to include the `DO` variable of any loop nested inside the parallel loop.
`LASTPRIVATE`	A problem occurs when you need the final value of a variable but the variable is declared `PRIVATE` rather than `LASTPRIVATE`. If the use of the final value happens several hundred lines farther down, or if the variable is in a common block and the final value is used in a completely separate routine, a variable can look as if it is `PRIVATE` when in fact it should be `LASTPRIVATE`. To combat this problem, simply declare all the `PRIVATE` variables `LASTPRIVATE` when debugging a loop.
`EQUIVALENCE`	Check for `EQUIVALENCE` problems. Two variables of different names may in fact refer to the same storage location if they are associated through an `EQUIVALENCE`.

	EQUIVALENCE statements affect storage of local variables and can cause data dependencies when parallelizing code. EQUIVALENCE statements with local variables cause the storage location to be initialized to zero and saved between calls to the subroutine.
Uninitialized variables	Some programs assume uninitialized variables are set to 0. This works with the `-static` option on the `f90`(1) command, but without it, uninitialized values assume the value that remains on the stack. When compiling with the `-mp` option on the `f90`(1) command, the program executes differently and the stack contents are different. You should suspect this type of problem when a program is compiled with `-mp` and is run on a single processor and produces a different result when it is compiled without `-mp`. To discover this type of problem, compile suspected routines with the `-static` option. If an uninitialized variable is the problem, you should initialize the variable rather than compile the program with the `-static` option.
Ranges on arrays	Perform array bounds checking analysis by compiling with the `-C` option on the `f90`(1) command. If arrays are indexed out of bounds, a memory location may be referenced in unexpected ways. This is particularly true of adjacent arrays in a common block.

Errors in choosing which arrays are `SHARED` can be detected only when running on multiple processors. When stepping through the code in the debugger, the program executes correctly.

The most likely candidates for this error are arrays with complicated subscripts. If the array subscripts are simply the variables of a `DO` loop, the analysis is probably correct. If the subscripts are more involved, examine those subscripts first.

If you suspect this type of error, print out all the values of all the subscripts on each iteration through the loop. Then use the `uniq`(1) command to look for duplicates. If duplicates are found, there is a data dependency.

Autotasking Directives (Outmoded) [C]

If your system includes multiple central processing units (CPUs), your program may be able to make use of *multitasking*, or running simultaneously on more than one CPU. This technology speeds up program execution by decreasing elapsed time. You can determine the number of CPUs on your system by entering the `hinv`(1) command.

The compiler automatically recognizes many parallel coding constructs, and it compiles them for multitasking without requiring additional user input; this capability is called *Autotasking*.

Autotasking directives let you specify the level of parallelism desired. You can start and end parallel processing at any number of suitable points within a subprogram. These directives are useful when the compiler fails to recognize parallelism that you know exists. This can occur, for example, when you have subroutine calls that can be executed in parallel.

Note: The directives in this section are outmoded, but they are still supported for older codes that require this functionality. Silicon Graphics encourages you to write new codes using the OpenMP directives described in Chapter 4, page 89.

This section provides an overview of the Autotasking directives recognized by the compiler.

⚠ **Caution:** The ability to use Autotasking directives in a subprogram that host associates a variable can result in undefined behavior. This applies only to Autotasking directives; it does not apply to parallelism detected by the compiler.

A branch out of a parallel region is not permitted and can produce incorrect results.

Autotasking directives control the way the compiler multitasks your program. You can insert tasking directive lines directly into your source code. The compiler supports the following Autotasking directives:

- `CASE`, `ENDCASE`
- `CNCALL`
- `DOALL`
- `DOPARALLEL`, `ENDDO`

- `GUARD`, `ENDGUARD`
- `NUMCPUS`
- `PARALLEL`, `ENDPARALLEL`
- `PERMUTATION`

The following sections describe the Autotasking directives.

C.1 Using Directives

The following sections describe how to use the CF90 Autotasking directives and the effects they have on programs.

For additional general information on using directives, see Section 3.1, page 69.

C.1.1 Directive Continuation

In the following example, an asterisk (`*`) appears in column 6 to indicate that the second line is a continuation of the preceding line:

```
!MIC$ GU
!MIC$*ARD
```

If you want to specify more than one directive on a line, separate each directive with a comma. Some directives require that you specify one or more arguments; when specifying a directive of this type, no other directive can appear on the line.

Spaces can precede, follow, or be embedded within a directive, regardless of source form.

Do not use source preprocessor (#) directives within multiline compiler directives (`CMIC$` or `!MIC$`).

C.1.2 Directive Range and Placement

The range and placement of directives is as follows:

- The Autotasking directives must appear within a program unit.
- The `ENDDO` directive must appear after the loop body of a `DOPARALLEL` loop, if it appears. The corresponding `DOPARALLEL` directive must be present.

- The following directives apply only to the next loop encountered lexically:
  - DOALL
  - DOPARALLEL
- The following Autotasking directives must appear as pairs within a program unit:
  - CASE, ENDCASE
  - GUARD, ENDGUARD
  - PARALLEL, ENDPARALLEL

C.1.3 Interaction of Directives with the -x Command Line Option

The -x option on the f90(1) command accepts one or more directives as arguments. When your input is compiled, the compiler ignores directives named as arguments to the -x option. If you specify -x mipspro, all directives are ignored. If you specify -x *dirname*, a particular directive is ignored. For more information on this command line option, see Section 2.74, page 65.

C.2 Concurrent Blocks: CASE and ENDCASE

The !MIC$ CASE directive serves as a separator between adjacent code blocks that can be executed concurrently. It marks the beginning of a control structure and signals that the code following it will be executed on a single processor.

!MIC$ ENDCASE serves as the terminator for a group of one or more parallel CASE directives. All work within the control structure must complete before execution continues with the code below the ENDCASE. The compiler does not automatically generate CASE directives.

The formats for these directives are as follows:

```
!MIC$ CASE

!MIC$ ENDCASE
```

Example. A single CASE/ENDCASE directive pair can also be used within a parallel region to allow only one processor to execute a code block, as follows:

```
!MIC$ PARALLEL
!MIC$ CASE
      CALL XYZ
!MIC$ ENDCASE
        :
!MIC$ DOPARALLEL
      DO I = 1, IMAX
        :
      END DO
!MIC$ ENDPARALLEL
```

In the preceding code, only one processor calls XYZ, and then all available processors execute the code following the ENDCASE.

C.3 Declare Lack of Side Effects: CNCALL

The !MIC$ CNCALL directive allows a loop to be Autotasked by asserting that subroutines called from the loop have no loop-related side effects (that is, they do not modify data referenced in other iterations of the loop) and therefore can be called concurrently by separate iterations of the loop. CNCALL is inserted immediately preceding the loop.

The format for this directive is as follows:

```
!MIC$ CNCALL
```

Example:

```
!MIC$ CNCALL
      DO I = 1, N
        CALL CRUNCH(A(I), B(I))
      END DO
```

C.4 Mark Parallel Loop: DOALL

The !MIC$ DOALL directive indicates that the DO loop beginning on the next line may be executed in parallel by multiple processors. No directive is needed to end a DOALL loop, (that is, the DOALL initiates a parallel region that contains only a DO loop with independent iterations). The loop index variable for a DOALL must be specified as a PRIVATE variable.

For a `!MIC$ DOALL` directive, all the variables and arrays in the region must be defined in a `SHARED` or `PRIVATE` parameter.

The format of this directive is as follows:

```
!MIC$ DOALL parameter [ [,] parameter ] ... [ [,] work_distribution ]
```

parameter	Table 4, page 258, describes parameters for the `DOALL` directive. More than one parameter can appear on the directive, but they must be separated by commas or blanks.
work_distribution	Parameters that specify the work distribution policy for iterations of the parallel `DO` loop. Only one can be used for a given `DO` loop. By default, iterations are distributed one at a time. Table 5, page 259, describes the work distribution parameters.

The default scheduling for a `DOALL` directive is `STATIC`. In addition, `CHUNKSIZE = CEILING(`*n*/*p*`)`, where *n* is the number of trips and *p* is the number of processors.

The `DOALL` directive does not accept the `MAXCPUS` or `AUTOSCOPE` clauses; their presence generates a fatal error.

Table 4. Autotasking directive *parameter* values

parameter	Description
`IF(`*expr*`)`	Performs a run-time test to choose between uniprocessing and multiprocessing. When not specified, multiprocessing is chosen if the loop is not in a routine that was called from within a parallel region. The logical expression (*expr*) determines (at run time) whether multiprocessing will occur. When *expr* is true, multiprocessing is enabled.
`PRIVATE(`*var*`[,`*var*`] ...)`	Specifies that the variables listed will have *private* scope; that is, each task (original or helper) will have its own private copy of these variables. The `PRIVATE` clause identifies those variables that are not shared between parallel processes. One variable cannot be declared both `PRIVATE` and `SHARED`. The loop control variable of the `DOALL` loop cannot be specified as `SHARED`; it must be specified as `PRIVATE`. Variables cannot be subobjects (that is, array elements or components of derived types).
`SAVELAST`	Specifies that the values of private variables, from the final iteration of a `DOALL` directive, will continue in the original task after execution of the iterations of the `DOALL`. By default, private variables are not guaranteed to retain the last iteration values. `SAVELAST` can be used only with `DOALL`, and if the full iteration set is not completed (for example, if the loop is exited early), the values of private variables are indeterminate.
`SHARED(`*var*`[,`*var*`] ...)`	Specifies that the variables listed will have *shared* scope; that is, they are accessible to both the original task and all helper tasks. The `SHARED` clause identifies those variables that are shared between parallel processes. One variable cannot be declared both `PRIVATE` and `SHARED`. The loop control variable of the `DOALL` loop cannot be specified as `SHARED`; it must be specified as `PRIVATE`. Variables cannot be subobjects (that is, array elements or components of derived types).

Table 5. Autotasking directive *work_distribution* values

work_distribution	Description
`CHUNKSIZE(`*n*`)`	Specifies the number of iterations to distribute to an available processor. *n* is an integer expression. For best performance, *n* should be an integer constant. For example, given 100 iterations and `CHUNKSIZE(4)`, 4 iterations at a time are distributed to each available processor until the 100 iterations are complete. By default, *n* is the number of loop iterations divided by the number of processors.
`GUIDED[(`*vl*`)]`	Specifies the use of guided self-scheduling to distribute the iterations to available processors. This mechanism minimizes synchronization overhead while providing acceptable dynamic load balancing. The *vl* argument is the vector length. *vl* must be of type integer and can be either a constant or a variable. The default *vl* is 1.

C.5 Mark Parallel Loop: DOPARALLEL and ENDDO

The `!MIC$ DOPARALLEL` directive indicates that the `DO` loop beginning on the next line may be executed in parallel by multiple processors. No directive is needed to end a `DOPARALLEL` loop.

The `!MIC$ ENDDO` directive extends a control structure beyond the `DO` loop. Without a `!MIC$ ENDDO` directive, all CPUs synchronize immediately after the loop, so that no processors can continue executing until all of the iterations are done. A `!MIC$ ENDDO` directive moves this point of synchronization from the end of the loop to the line of the `!MIC$ ENDDO` directive.

This lets the compiler use parallelism in loops containing some forms of reduction computations. These directives can be used only within a parallel region bounded by the `PARALLEL` and `ENDPARALLEL` directives.

All variables and arrays in a parallel region must be declared as `PRIVATE` or `SHARED`.

The formats for these directives are as follows:

```
!MIC$ DOPARALLEL [work_distribution]

!MIC$ ENDDO
```

The *work_distribution* arguments are described in Table 5, page 259. Only one *work_distribution* can be used for a given DO loop.

In the following example, a parallel region is defined by PARALLEL and ENDPARALLEL. A reduction computation is implemented by a DOPARALLEL/ENDDO pair, which ensures that all contributions to SUM and BIG are included, and GUARD/ENDGUARD, which protects the updating of shared variables SUM and BIG.

```
      SUM = 0.0
      BIG = -1.0
!MIC$ PARALLEL PRIVATE(XSUM,XBIG,I)
!MIC$*         SHARED(SUM,BIG,AA,BB,CC)
      XSUM = 0.0
      XBIG = -1.0
!MIC$ DOPARALLEL
      DO I = 1, 2000
         :
         XSUM = XSUM + (AA(I)*(BB(I)-CC(AA(I))))
         XBIG = MAX(ABS(AA(I)*BB(I)), XBIG)
         :
      END DO
!MIC$ GUARD
      SUM = SUM + XSUM
      BIG = MAX(XBIG,BIG)
!MIC$ ENDGUARD
!MIC$ ENDDO
!MIC$ ENDPARALLEL
```

C.6 Critical Region: GUARD and ENDGUARD

The !MIC$ GUARD and !MIC$ ENDGUARD directives delimit a critical region, providing the necessary synchronization to protect or guard the code inside the critical region. A *critical region* is a code block that is to be executed by only one processor at a time, although all processors that enter a parallel region will execute it.

The formats for these directives are as follows:

```
!MIC$ GUARD [n]

!MIC$ ENDGUARD [n]
```

n	Mutual exclusion flag; two regions with the same flag cannot be active concurrently. *n* must be of type integer and can be a variable or an expression, from which the low-order 6 bits are used. For example, `GUARD 1` and `GUARD 2` can be active concurrently, but two `GUARD 7` directives cannot.

For optimal performance, no *n* should be specified. Otherwise, *n* should be an integer constant; a general expression can be used for the unusual case that the critical region number must be passed to a lower-level routine. When *n* is not provided, the critical region blocks only other instances of itself, but no other critical regions. Critical regions may appear anywhere in a program. That is, they are not limited to parallel regions.

Numbered `GUARD` directives are not supported. They are implemented as unnamed `GUARD` directives. This can lead to deadlock if the user has nested `GUARD` directives.

C.7 Specify Maximum Number of CPUs for a Parallel Region: `NUMCPUS`

The `!MIC$ NUMCPUS` directive globally indicates the maximum number of CPUs that a section of code can use effectively. It does not guarantee that this number of processors will actually be assigned. The `NUMCPUS` directive is in effect until a subsequent `NUMCPUS` directive is encountered. The `NUMCPUS` directive stays in effect across program units. The `NUMCPUS` directive remains in effect for all subsequently called subroutines. Without this directive, CPUs are allocated based on the `MP_SET_NUMTHREADS` environment variable and workload.

The format for this directive is as follows:

```
!MIC$ NUMCPUS (ncpus)
```

ncpus	Globally specifies the maximum number of CPUs that a code can use effectively. *ncpus* must be of type integer and can be a constant, variable, or expression.

The number of CPUs specified with this directive should be equal to or less than the number of CPUs specified by the `MP_SET_NUMTHREADS` environment variable. If the number requested with the `NUMCPUS` directive is greater than the number specified by the `MP_SET_NUMTHREADS` environment variable, no error is issued, but the directive has no effect.

C.8 Mark Parallel Region: PARALLEL and ENDPARALLEL

The `!MIC$ PARALLEL` and `!MIC$ ENDPARALLEL` directives mark, respectively, the beginning and end of a parallel region. Parallel regions are combinations of redundant code blocks and partitioned code blocks. The formats for these directives are as follows:

```
!MIC$ PARALLEL [parameter [ [,] parameter ] ...]

!MIC$ ENDPARALLEL
```

The *parameter*s are described in Table 4, page 258.

The `PARALLEL` directive indicates where multiple processors enter execution. The portion of code that all processors execute until reaching a `DOPARALLEL` directive is called a *redundant code block*. Because the iterations of the `DO` loop within a `DOPARALLEL` directive are distributed across available processors, this portion of code is called the *partitioned code block*. The scope of a variable in a parallel region is either shared or private. Shared variables are used by all processors; private variables are unique to a processor.

When the compiler generates code for a `!MIC$ PARALLEL` directive, all the variables and arrays in the region must be defined in a `SHARED` or `PRIVATE` parameter.

C.9 Declare an Array with No Repeated Values: PERMUTATION

The `!MIC$ PERMUTATION` directive declares that an integer array has no repeated values. This is useful when the integer array is used as a subscript for another array (vector-valued subscript). The format for this directive is as follows:

```
!MIC$ PERMUTATION (ia [, ia ] ...)
```

ia	Integer array that has no repeated values for the entire routine.

When an array with a vector-valued subscript appears on both sides of the equal sign in a loop, many-to-one assignment is possible even when the subscript is identical. Many-to-one assignment occurs if any repeated elements exist in the subscripting array. If it is known that the integer array is used merely to permute the elements of the subscripted array, it can often be determined that many-to-one assignment does not exist with that array reference.

Sometimes a vector-valued subscript is used as a means of indirect addressing because the elements of interest in an array are sparsely distributed; in this case, an integer array is used to select only the desired elements, and no repeated elements exist in the integer array, as in the following example:

```
!MIC$ PERMUTATION(IPNT) ! IPNT has no repeated values
      ...
      DO I = 1, N
         A(IPNT(I)) = A(IPNT(I)) + B(I)
      END DO
```

C.10 Examples

The following examples show shared and private variables and arrays.

C.10.1 Read-only Variables

The following examples show read-only variables:

```
!MIC$ DOALL PRIVATE(I) SHARED(N1,N2,A)
      DO I = N1, N2
      ...= A
      END DO
```

A is a shared variable because it is a read-only variable. All processors share the same location for A.

```
!MIC$ DOALL SHARED(N1,N2,M1,M2,V) PRIVATE(I,J)
      DO 10 I = N1, N2
      DO 10 J = M1, M2
         ... = V(J)
      END DO
```

V is shared because it is a read-only array. N1, N2, M1, and M2 are also shared because they are read-only variables. I and J are written and then read, so they are private variables.

C.10.2 Array Indexed by Loop Index

The following example shows an array indexed by the loop index:

```
!MIC$ DOALL SHARED(N1,N2,V,U,J) PRIVATE(I,T)
      DO I = N1, N2
        T = V(I)
        U(I,J) = T
      END DO
```

U and V are shared arrays because they are indexed by the loop index. All processors share the same location for V and U. T is written and then read, so it is a private variable. J is shared because it is a read-only variable.

C.10.3 Read-then-write Variables

The following example shows read-then-write variables:

```
      SUM = 0.0
!MIC$ DOALL SHARED(N1,N2,V,SUM) PRIVATE(I,T)
      DO I = N1, N2
        T = V(I)
!MIC$ GUARD
        SUM = SUM + T
!MIC$ ENDGUARD
      END DO
```

SUM is a shared variable because it is read before it is written. Special care is needed in writing into a shared variable that is not indexed by the loop control variable.

C.10.4 Write-then-read Variables and Arrays

The following example shows write-then-read variables and arrays:

```
!MIC$ DOALL SHARED(N1,N2,M1,M2) PRIVATE(I,J,V)
      DO 10 I = N1, N2
      DO 10 J = M1, M2
      V(J) = ...
      ... = V(J)
      END DO
```

V is written to and then read. It must be a private array.

Multiprocessing Directives (Outmoded) [D]

The MIPSpro 7 Fortran 90 multiprocessing directives let you optimize your code by helping you to split your program into concurrently executing pieces. This appendix describes techniques for analyzing your code and preparing it for execution on multiple CPUs.

Note: The directives in this appendix are outmoded. They are supported for older codes that require this functionality. Section D.1, page 267 lists the equivalent new directives that you can use in place of the outmoded directives.

Silicon Graphics and Cray Research encourage you to write new codes using the OpenMP directives described in Chapter 4, page 89.

This appendix describes two sets of directives to use for multiprocessing. The first set consists of the loop-level multiprocessing directives. The second set consists of directives based on the work of the Parallel Computing Forum (PCF). The PCF directives allow you to specify multiprocessing based on the model of a parallel region. The following sections describe the multiprocessing directives and how to use them.

The `-mp` option must be specified on the `f90`(1) command line in order for the compiler to honor the directives in this chapter. For more information on multiprocessing, see the `mp`(3F) and `sync`(3F) man pages.

D.1 Migrating to OpenMP

The functionality of the directives in this chapter has been preserved in the OpenMP Fortran API directives. The following list indicates equivalent directives:

MIPSpro Directive	OpenMP Directive
`!$DOACROSS`	`!$OMP PARALLEL DO`
`!$CHUNK`	Optional second argument to the `SCHEDULE` `(`*type* `[,`*chunk*`])` clause
`!$MP_SCHEDTYPE`	First argument to the `SCHEDULE (`*type* `[,`*chunk*`])` clause
`!$` *statement*	`!$` or `_OPENMP` preprocessor macro

`!$PAR BARRIER`	`!$OMP BARRIER`
`!$PAR CRITICALSECTION`	`!$OMP CRITICAL [(`*name*`)]`
`!$PAR END CRITICALSECTION`	`!$OMP END CRITICAL`
`!$PAR PARALLEL`	`!$OMP PARALLEL`
`!$PAR END PARALLEL`	`!$OMP END PARALLEL`
`!$PAR PARALLELDO`	`!$OMP PARALLEL DO`
`!$PAR PDO`	`!$OMP DO`
`!$PAR ENDPDO`	`!$OMP END DO`
`!$PAR PSECTION`	`!$OMP SECTIONS`
`!$PAR SECTION`	`!$OMP SECTION`
`!$PAR END PSECTION`	`!$OMP END SECTIONS`
`!$PAR SINGLEPROCESS`	`!$OMP SINGLE`
`!$PAR END SINGLEPROCESS`	`!$OMP END SINGLE`
`!$DISTRIBUTE`	`!$SGI DISTRIBUTE`
`!$DISTRIBUTE_RESHAPE`	`!$SGI DISTRIBUTE_RESHAPE`
`!$REDISTRIBUTE`	`!$SGI REDISTRIBUTE`
`!$DYNAMIC`	`!$SGI DYNAMIC`
`!$PAGE_PLACE`	`!$SGI PAGE_PLACE`

The following list indicates equivalent clauses:

MIPSpro Clause	OpenMP Clause
`AFFINITY`	`$SGI& AFFINITY`
`BLOCKED`	`STATIC`
`CHUNK`	`SCHEDULE(*[[,` *chunk*`])`
`IF`	`IF`
`LASTLOCAL`	`LASTPRIVATE`
`LOCAL`	`PRIVATE`
`MP_SCHEDTYPE`	`SCHEDULE(`*type*`,[*])`

`NEST`	`!$SGI& NEST`
`PRIVATE`	`PRIVATE`
`REDUCTION`	`REDUCTION`
`SHARED`	`SHARED`
`NOWAIT`	`NOWAIT`

The following list indicates equivalent *mode*s for `!$MP_SCHEDTYPE`:

MIPSpro *mode*	OpenMP *mode*
`DYNAMIC`	`DYNAMIC`
`GSS`	`GUIDED`
`INTERLEAVE`	`SCHEDULE(STATIC, 1)`
`RUNTIME`	`RUNTIME`
`SIMPLE`	`STATIC`

D.2 Using Directives

Certain multiprocessing features are available to you either through the command line or through directives. For command line options and directives that accept either `ON` or `OFF` as arguments, the compiler turns the feature `OFF` when conflicting settings are present. If a feature accepts a numeric setting as an argument, the compiler compares the command line setting and the directive setting and uses the minimum setting.

The following sections contain general information that applies to both the loop-level and the PCF directives.

D.2.1 Directive Range

Directives placed in a file prior to program code are called *global directives*. The compiler interprets them as if they appeared at the top of each program unit in the file.

Directives appearing anywhere else in the file apply only until the end of the current program unit. The compiler resets the value of the directive to the global value at the start of the next program unit.

D.2.2 Directive Continuation

To continue the loop-level multiprocessing directives onto another line, use `!$&` as the first characters in the continued line(s). For example:

```
!$DOACROSS share(ALPHA, BETA, GAMMA, DELTA,
!$&  EPSILON, OMEGA), LASTLOCAL(I, J, K, L, M, N),
!$&  LOCAL(XXX1, XXX2, XXX3, XXX4, XXX5, XXX6, XXX7,
!$&  XXX8, XXX9)
```

To continue the PCF directives onto another line, begin the continued line with the characters `!$PAR&`.

D.3 Loop-level Multiprocessing Directives

It is possible for the compiler to execute different iterations of a `DO` loop on multiple processors. For example, suppose a `DO` loop consisting of 200 iterations will run on a machine with four processors using the simplest scheduling method. The first 50 iterations run on one processor, the next 50 on another, and so on.

A multiprocessing code adjusts itself at run time to the number of processors actually available to it on the machine. By default, the multiprocessing code does not use more than eight processors. If you want to use more processors, set the `MP_SET_NUMTHREADS` environment variable to a different value. If the 200-iteration loop was moved to a machine with only two processors, it would be divided into two blocks of 100 iterations each, without any need to recompile or reload. In fact, multiprocessing code can be run on single-processor machines; on such systems the iterations are divided into one block of 200 iterations. This allows code to be developed on a single-processor system and later run on a multiprocessor.

The processes that participate in the parallel execution of a task are arranged in a master/slave organization. The original process is the master. It creates zero or more slaves to assist. When a parallel `DO` loop is encountered, the master contacts the slaves for help. When the loop is complete, the slaves wait for the master, and the master resumes normal execution. The master process and each of the slave processes are called a *thread of execution* or simply a *thread*. By default, the number of threads is set to the number of processors on the machine or is set to 8, whichever is smaller. You can override the default and explicitly control the number of threads of execution used by a parallel job.

For multiprocessing to work correctly, the iterations of the loop must not depend on each other; each iteration must stand alone and produce the same answer regardless of when any other iteration of the loop is executed. Not all DO loops have this property, and loops without it cannot be correctly executed in parallel. However, many of the loops encountered in practice fit this model. Further, many loops that cannot be run in parallel in their original form can be rewritten to run wholly or partially in parallel. For information about determining data dependencies in loops, see Section 4.10, page 126.

The loop-level multiprocessing directives are as follows:

- DOACROSS
- CHUNK
- MP_SCHEDTYPE

The following sections describe the loop-level multiprocessing directives.

Note: Localized ALLOCATABLE or POINTER arrays are not supported on the DOACROSS directive. They cannot be specified in a LOCAL clause. Also, Cray Pointees are not supported in a LOCAL clause.

D.3.1 DOACROSS Directive

The basis for the loop-level multiprocessing directives is the DOACROSS directive. This directive indicates to the compiler that it should run iterations of the subsequent DO loop in parallel. This directive must appear directly before the loop that is to be operated on, and it remains in effect for that loop only.

The format of this directive is as follows:

```
!$DOACROSS [ clause [, clause ] ...]
```

clause This directive accepts one or more of the following *clause*s:

- AFFINITY
- BLOCKED
- CHUNK
- IF
- LASTLOCAL
- LOCAL

- MP_SCHEDTYPE
- NEST
- PRIVATE
- REDUCTION
- SHARED

The sections that follow describe the DOACROSS directive clauses.

Appendix B, page 249, contains information on debugging when DOACROSS directives are used.

Note: The Fortran compiler does not support direct nesting of DOACROSS loops.

For example, the following is illegal and generates a compilation error:

```
!$DOACROSS LOCAL(I)
        DO I = 1, N
!$DOACROSS LOCAL(J)
           DO J = 1, N
              A(I,J) = B(I,J)
           END DO
        END DO
```

However, to simplify separate compilation, a different form of nesting is allowed. A routine that uses !$DOACROSS can be called from within a multiprocessed region. This can be useful if a single routine is called from several different places: sometimes from within a multiprocessed region, sometimes not. Nesting does not increase the parallelism. When the first !$DOACROSS loop is encountered, that loop is run in parallel. While in the parallel loop, if a call is made to a routine that itself has a !$DOACROSS, the subsequent loop is executed serially.

D.3.1.1 AFFINITY Clause

Affinity scheduling allows you to map parallel loop iterations onto underlying threads. This clause is used most often on Origin series systems.

For more information on using this DOACROSS clause, see Section 5.2.2.1, page 152.

D.3.1.2 `BLOCKED` and `CHUNK` Clauses

These clauses affect work scheduling among the participating tasks in a loop. They break up the work into pieces specified by *int_expr*. These clauses are valid only when the `MP_SCHEDTYPE=DYNAMIC` or `MP_SCHEDTYPE=INTERLEAVE` clauses have also been specified.

The `BLOCKED` and `CHUNK` clauses have the following formats:

```
BLOCKED (int_expr)

CHUNK = int_expr
```

int_expr	Specify an integer expression that represents the size of the chunk (that is, the number of iterations per chunk).

If `CHUNK` or `BLOCKED` are specified, and `MP_SCHEDTYPE` is not, `MP_SCHEDTYPE` defaults to `DYNAMIC`. For more information on how these clauses interact with the `MP_SCHEDTYPE` clause, see Section D.3.1.5, page 275.

The `CHUNK` directive also affects the division of work. For more information on the `CHUNK` directive, see Section D.3.2, page 277.

D.3.1.3 `IF` Clause

The `IF` clause determines whether the loop is actually executed in parallel. This clause has the following format:

```
IF (logical_expr)
```

logical_expr	Specify a logical expression. If *logical_expr* evaluates to `TRUE`, the loop is executed in parallel. If *logical_expr* evaluates to `FALSE`, the loop is executed serially.

D.3.1.4 `LASTLOCAL`, `LOCAL`, `PRIVATE` and `SHARED` Clauses

The `LASTLOCAL`, `LOCAL`, and `SHARED` clauses specify lists of variables used within parallel loops. A variable can appear in only one of these lists. The effect of these clauses is as follows:

- The `LASTLOCAL` clause specifies variables that are local to each process. Unlike with the `LOCAL` clause, the compiler saves only the value of the logically last iteration of the loop when it exits. The name `LASTLOCAL` is preferred over `LAST LOCAL`.
- The `LOCAL` clause specifies variables that are local to each process. If a variable is declared as `LOCAL`, each iteration of the loop is given its own uninitialized copy of the variable. You can declare a variable as `LOCAL` if its value does not depend on any other iteration of the loop and if its value is used only within a single iteration. In effect, the `LOCAL` variable is just temporary; a new copy can be created in each loop iteration without changing the final answer. The name `LOCAL` is preferred over `PRIVATE`.

  Note: Localized `ALLOCATABLE` or `POINTER` arrays cannot be specified in a `LOCAL` clause. Also, Cray Pointees are not supported in a `LOCAL` clause.

- The `SHARED` clause specifies variables that are shared across all processes. If a variable is declared as `SHARED`, all iterations of the loop use the same copy of the variable. You can declare a variable as `SHARED` if it is only read (not written) within the loop or if it is an array in which each iteration of the loop uses a different element of the array. The name `SHARED` is preferred over `SHARE`.

By default, the `DO` variable is `LASTLOCAL` and all other variables are `SHARED`.

These clauses have the following formats:

```
LASTLOCAL var [ , var ... ]

LOCAL var [ , var ... ]

SHARED var [ , var ... ]
```

var Specify the name of a variable. If any *var* is an array, it is listed without any subscripts.

Common blocks, allocatable arrays, and Fortran 90 pointers cannot appear as *var* arguments in a `LOCAL` list.

`LOCAL` is a little faster than `LASTLOCAL`, so if you do not need the final value, it is good practice to put the `DO` index variable into the `LOCAL` list, although this is not required.

D.3.1.5 `MP_SCHEDTYPE` Clause

The `MP_SCHEDTYPE` clause affects the way the compiler schedules work among the participating tasks in a loop.

This clause has the following format:

```
MP_SCHEDTYPE = mode
```

mode Specify one of the following for *mode*:

- `DYNAMIC`. Specifying `MP_SCHEDTYPE=DYNAMIC` breaks the iterations into pieces the size of which is specified with the `CHUNK` clause. As each process finishes a piece, it enters a critical section to grab the next available piece. This gives good load balancing at the price of higher overhead. The `CHUNK` clause is valid with this *mode*.
- `GSS`. Specifying `MP_SCHEDTYPE=GSS` results in a variation of the guided self-scheduling algorithm. The piece size is varied depending on the number of iterations remaining. By parceling out relatively large pieces to start with and relatively small pieces toward the end, the system can achieve good load balancing while reducing the number of entries into the critical section. Specifying `GUIDED` for *mode* performs the same function as specifying `GSS`, but `GSS` is preferred.
- `INTERLEAVE`. Specifying `MP_SCHEDTYPE=INTERLEAVE` breaks the iterations into pieces of the size specified by the `CHUNK` clause, and execution of those pieces is interleaved among the processes. For example, if there are four processes and `CHUNK=2`, the first process executes iterations 1–2, 9–10, 17–18, …; the second process executes iterations 3–4, 11–12, 19–20,…; and so on. Although this is more complex than the

simple method, it is still a fixed schedule with only a single scheduling decision. The `CHUNK` clause is valid with this *mode*. Specifying `INTERLEAVED` for *mode* performs the same function as specifying `INTERLEAVE`, but `INTERLEAVE` is preferred.

- `RUNTIME`. Specifying `MP_SCHEDTYPE=RUNTIME` directs the scheduling routine to examine environment variables to select a *mode*. For the list of valid environment variables, see the `pe_environ`(5) man page.
- `SIMPLE`. Specifying `MP_SCHEDTYPE=SIMPLE` divides the iterations among processes by dividing them into contiguous pieces and assigning one piece to each process. Specifying `STATIC` for *mode* performs the same function as specifying `SIMPLE`, but `SIMPLE` is preferred. Default is `SIMPLE`.

The `MP_SCHEDTYPE` clause interacts with the `CHUNK` clause as follows:

- If both the `MP_SCHEDTYPE` and `CHUNK` clauses are omitted, `SIMPLE` scheduling is assumed.
- If `MP_SCHEDTYPE=INTERLEAVE` or `MP_SCHEDTYPE=DYNAMIC` and the `CHUNK` clause is omitted, `CHUNK=1` is assumed.
- If `MP_SCHEDTYPE` is set to one of the other values, `CHUNK` is ignored.
- If the `MP_SCHEDTYPE` clause is omitted, but `CHUNK` is set, `MP_SCHEDTYPE=DYNAMIC` is assumed.

D.3.1.6 `NEST` Clause

The `NEST` clause allows you to exploit nested concurrency. This `DOACROSS` clause is used most often on Origin series systems. For more information on this clause, see Section 5.2.2, page 151.

D.3.1.7 `REDUCTION` Clause

The `REDUCTION` clause specifies variables involved in a reduction operation. In a *reduction operation*, the compiler keeps local copies of the variables and combines them when it exits the loop.

This clause has the following format:

```
REDUCTION var [ , var ]...
```

var Specify one or more variable names for *var*. Each *var* must be a scalar individual variable, not an array. A *var* can be an array element (for example `REDUCTION(A(I,J))`).

One element of an array can be used in a reduction operation while other elements of the array are used in other ways. To allow for this, if an element of an array appears in the `REDUCTION` list, the entire array can also appear in the `SHARED` list.

The four types of reductions supported are `sum`(+), `product`(*), `min`(), and `max`(). Note that `min` and `max` reductions must use the `MIN`(3I) and `MAX`(3I) intrinsic functions to be recognized correctly.

The compiler confirms that the reduction expression is legal by making some simple checks. The compiler does not, however, check all statements in the `DO` loop for illegal reductions. You must ensure that the reduction variable is used correctly in a reduction operation.

Example:

```
!$DOACROSS LOCAL(I), REDUCTION(A(1))
      DO I = 2,N
         A(1) = A(1) + A(I)
      END DO
```

D.3.2 `CHUNK` Directive

The `CHUNK` directive breaks work up into pieces. Like the `MP_SCHEDTYPE` directive, the `CHUNK` directive acts as an implicit clause, in this case a `CHUNK` clause, for all `DOACROSS` directives in the scope. The `CHUNK` directive is in effect from the place it occurs in the source until another corresponding directive is encountered or the end of the procedure is reached.

The format of this directive is as follows:

```
!$CHUNK=int_expr
```

int_expr Specify an integer expression that represents the size of the chunk (that is, the number of iterations per chunk).

The `CHUNK` clause to the `DOACROSS` directive also divides work. For more information, see Section D.3.1.2, page 273.

D.3.3 MP_SCHEDTYPE Directive

The MP_SCHEDTYPE directive affects the way the compiler schedules work among the participating tasks in a loop. Like the CHUNK directive, the MP_SCHEDTYPE directive acts as an implicit clause, in this case an MP_SCHEDTYPE clause, for all DOACROSS directives in the scope. The MP_SCHEDTYPE directive is in effect from the place it occurs in the source until another corresponding directive is encountered or the end of the procedure is reached.

The MP_SCHEDTYPE directive specifies the scheduling type to be used for subsequent !$DOACROSS directives that are specified without an explicit scheduling type.

The format of this directive is as follows:

```
!$MP_SCHEDTYPE mode
```

mode	This directive accepts a *mode* argument as described in Section D.3.1.5, page 275.

The MP_SCHEDTYPE clause to the DOACROSS directive also divides work. For more information, see Section D.3.1.5, page 275.

D.3.4 !$ Directive

The !$ directive, which is really only a prefix, precedes code that should be recognized only when multiprocessing is enabled. Multiprocessing is enabled when either -pfa or -mp is specified on the f90(1) command line.

These directive lines are considered comment lines except when multiprocessing. A line beginning with !$ is treated as a conditionally compiled Fortran statement.

The format of this directive is as follows:

```
!$ statement
```

statement	For *statement*, specify a standard Fortran statement. This feature can be used to insert debugging statements or other arbitrary code.

If *statement* is a Fortran 90 statement, the *statement* can be continued to a subsequent line in fixed source form by placing an ampersand (&) in column 6 of the continued line.

If statement is a directive, continue it using the rules for directive continuation described in Section D.2.2, page 270.

The following code demonstrates the use of the `!$` directive:

```
!$    PRINT 10
!$ 10 FORMAT('BEGIN MULTIPROCESSED LOOP')
!$DOACROSS LOCAL(I), SHARED(A,B)
        DO I = 1, 100
           CALL COMPUTE(A, B, I)
        END DO
```

D.3.5 `DOACROSS` Directive Examples

This section contains examples of `DOACROSS` directives.

Example 1. Simple `DOACROSS` directive. Consider the following code fragment:

```
     DO 10 I = 1, 100
        A(I) = B(I)
10   CONTINUE
```

By inserting a directive, it can be multiprocessed:

```
!$DOACROSS LOCAL(I), SHARED(A, B)
     DO 10 I = 1, 100
        A(I) = B(I)
10   CONTINUE
```

Here, the defaults are sufficient provided that `A` and `B` are mentioned in a nonparallel region or in another `SHARED` list. The following code will then work:

```
!$DOACROSS
     DO 10 I = 1, 100
        A(I) = B(I)
10   CONTINUE
```

Example 2. A `DOACROSS` directive with a `LOCAL` clause. Consider the following code fragment:

```
     DO 10 I = 1, N
        X = SQRT(A(I))
        B(I) = X*C(I) + X*D(I)
10   CONTINUE
```

The following code shows this fragment rewritten for multiprocessing using explicit clauses:

```
!$DOACROSS LOCAL(I, X), SHARED(A, B, C, D, N)
     DO 10 I = 1, N
        X = SQRT(A(I))
        B(I) = X*C(I) + X*D(I)
10   CONTINUE
```

The following code shows the fragment rewritten for multiprocessing using the default settings:

```
!$DOACROSS LOCAL(X)
     DO 10 I = 1, N
        X = SQRT(A(I))
        B(I) = X*C(I) + X*D(I)
10   CONTINUE
```

Example 3. A `DOACROSS` directive with a `LASTLOCAL` clause. Consider the following code fragment:

```
     DO 10 I = M, K, N
        X = D(I)**2
        Y = X + X
        DO 20 J = I, MAX
           A(I,J) = A(I,J) + B(I,J) * C(I,J) * X + Y
20   CONTINUE
10   CONTINUE
     PRINT*, I, X
```

In this example, the final values of `I` and `X` are needed after the loop completes. A correct directive is shown in the following:

```
!$DOACROSS LOCAL(Y,J), LASTLOCAL(I,X),
!$& SHARED(M,K,N,ITOP,A,B,C,D)
     DO 10 I = M, K, N
         X = D(I)**2
```

```
          Y = X + X
          DO 20 J = I, ITOP
              A(I,J) = A(I,J) + B(I,J) * C(I,J) *X + Y
20        CONTINUE
10    CONTINUE
      PRINT*, I, X
```

You can also use the defaults:

```
!$DOACROSS LOCAL(Y,J), LASTLOCAL(X)
      DO 10 I = M, K, N
          X = D(I)**2
          Y = X + X
          DO 20 J = I, MAX
              A(I,J) = A(I,J) + B(I,J) * C(I,J) *X + Y
20        CONTINUE
10    CONTINUE
      PRINT*, I, X
```

In the preceding code example, `I` is a loop index variable for the `DOACROSS` loop, so it is `LASTLOCAL` by default. Even though `J` is a loop index variable, it is not the loop index of the loop being multiprocessed and has no special status. If it is not declared, it is assigned the default value of `SHARED`, which produces an incorrect answer.

D.4 Local Common Blocks

The `-Xlocal` option to the `ld`(1) command allows named common blocks to be local to a process. Each process in the parallel job gets its own private copy of the common block. This can be helpful in converting certain types of Fortran programs into a parallel form.

The common block must be a named common block (blank common cannot be made local), and it must not be initialized by `DATA` statements.

To create a local common block, use the special loader option `-Xlocal` followed by a list of common block names. The external name of a common block known to the loader has a trailing underscore and is not surrounded by slashes. For example, the following command makes the common block `/foo/` a local common block in the resulting `a.out` file. You can specify multiple `-Xlocal` options if necessary.

```
% f90 -mp a.o -Wl,-Xlocal,foo_
```

You can use the `!$COPYIN` directive to copy values from the master thread's version of the common block into the slave thread's version. This directive has the following format:

```
!$COPYIN item [, item] ...
```

item	Specify one or more members of a local common block. Each *item* can be a variable, an array, an individual element of an array, or the entire common block.

Note: The `!$COPYIN` directive cannot be executed from inside a parallel region.

The following example propagates the values for `x` and `y`, all the values in the common block `foo`, and the `I`th element of array `A`:

```
!$COPYIN X,Y, /FOO/, A(I)
```

These items must be either common blocks or members of common blocks. The directive is translated into executable code, so in this example, `I` is evaluated at the time this statement is executed.

D.5 PCF Directives

In addition to the simple loop-level parallelism offered by the `DOACROSS` directive, the compiler supports a set of directives that allows you to specify a more general model of parallelism. This model is based on the work done by the Parallel Computing Forum (PCF), which itself formed the basis for the proposed ANSI-X3H5 standard.

The main concept in this model is the *parallel region*, which can be any arbitrary section of code (not just a `DO` loop). Within the parallel region, there are special *work-sharing constructs* that can be used to divide the work among separate processes or threads. All master and slave threads synchronize at the bottom of a work-sharing construct. None of the threads continue past the end of a construct until they all have completed execution within that construct.

The parallel region can also contain a *critical section* construct, where exactly one process executes at a time. Within a critical section, only one thread executes at a time, and threads do not synchronize at the bottom of a critical section.

The master thread executes the user program until it reaches a parallel region. It then spawns one or more slave threads that begin executing code at the beginning of a parallel region. Each thread executes all the code in the region until a work sharing construct is encountered. Each thread then executes some portion of the work sharing construct, and then resumes executing the parallel region code. At the end of the parallel region, all the threads synchronize, and the master thread continues execution of the user program.

For information on interthread communication with library routines, see Appendix A, page 239.

The compiler recognizes the PCF directives when multiprocessing is enabled with either the `-mp` or the `-pfa` option to the `f90`(1) command. The PCF directives are as follows:

- `BARRIER`
- `CRITICAL SECTION`, `END CRITICAL SECTION`
- `PARALLEL`, `END PARALLEL`
- `PARALLEL DO`
- `PDO`, `END PDO`
- `PSECTION[S]`, `SECTION`, and `END PSECTION[S]`
- `SINGLE PROCESS`, `END SINGLEPROCESS`

The following sections describe the syntax of the PCF directives.

Note: Generated code from the PCF directives is sometimes slower than the generated code from the special case parallelism offered by the `DOACROSS` directive. PCF directive code is slower because of the extra synchronization required. When a `DOACROSS` loop executes, there is a synchronization point at entry and another at exit. When a parallel region executes, there is a synchronization point at entry to the region, another at each entry to a work-sharing construct, another at each exit from a work-sharing construct, and one at exit from the region. Thus, several separate `DOACROSS` loops typically execute faster than a single parallel region with several `PDO` directives. Limit your use of the parallel region construct to those few cases that actually need it.

D.5.1 `BARRIER` Directive

The `BARRIER` directive ensures that each process waits until all processes reach the barrier before proceeding.

This directive has the following format:

```
!$PAR BARRIER
```

D.5.2 CRITICAL SECTION and END CRITICAL SECTION Directives

The CRITICAL SECTION and END CRITICAL SECTION directives ensure that the enclosed block of code is executed by only one process (thread) at a time. Another process attempting to gain entry to the critical section must wait until the previous process has exited. Threads do not synchronize at the bottom of a critical section.

The critical section construct can appear anywhere in a program, including inside and outside a parallel region and within a DOACROSS loop.

These directives have the following format:

```
!$PAR CRITICAL SECTION [ (lock_variable) ]

!$PAR END CRITICAL SECTION
```

lock_variable	Specify an integer variable that is initialized to zero. The parentheses are required. If you do not specify *lock_variable*, the compiler automatically supplies a global lock. Multiple critical section constructs inside the same parallel region are considered to be independent of each other unless they use the same explicit *lock_variable*.

D.5.3 PARALLEL and END PARALLEL Directives

The PARALLEL and END PARALLEL directives enclose a parallel region that includes work-sharing constructs and critical sections. It signifies the boundary within which slave threads execute. A user program can contain any number of parallel regions.

These directives have the following format:

```
!$PAR PARALLEL [clause [,clause]...]

!$PAR END PARALLEL
```

clause Specify one of the following clauses:

- `IF` (*logical_expression*)
- `LOCAL` *var*[`,` *var*] `...`
- `SHARED` *var*[`,` *var*] `...`

The `IF`, `LOCAL`, and `SHARED` clauses have the same meaning as for the `DOACROSS` directive. Also as with the `DOACROSS` directive, the keyword `LOCAL` is preferred to `PRIVATE` and the keyword `SHARED` is preferred to `SHARE`. For more information on these clauses and their syntax, see Section D.3.1, page 271.

The preferred form of the directive has no commas between the clauses.

In the following code, all threads enter the parallel region and call routine `FOO`:

```
          SUBROUTINE EX1(INDEX)
          INTEGER I
!$PAR PARALLEL LOCAL(I)
          I = MP_MY_THREADNUM()
          CALL FOO(I)
!$PAR END PARALLEL
          END
```

D.5.4 `PARALLEL DO` Directive

The `PARALLEL DO` directive indicates that the iterations of the subsequent `DO` loop should be executed by different processes. This directive produces the same effect as the `DOACROSS` directive, and it is conceptually the same as a parallel region containing exactly one `PDO` construct and no other code. Each thread inside the enclosing parallel region executes separate iterations of the loop within the parallel `DO` construct. This directive must not appear within a parallel region.

This directive has the following format:

```
!$PAR PARALLELDO [clause [,clause ] ...]
```

clause For *clause*, enter one or more of the DOACROSS clauses described in Section D.3.1, page 271.

D.5.5 PDO and END PDO Directives

The PDO and END PDO directives surround a loop and indicate that the iterations of the enclosed loop should be executed by different processes. These directives must be enclosed within a parallel region delimited by PARALLEL and END PARALLEL directives.

Within a parallel region, each thread inside the region executes a separate iteration of a loop within a PDO construct.

These directives have the following format:

```
!$PAR PDO [clause [, clause]...]
```

```
[!$PAR END PDO [NOWAIT]]
```

clause Specify one of the following clauses:

- AFFINITY
- CHUNK=*int_expr*
- LASTLOCAL *var*
- LOCAL *var* [, *var*] ...
- MP_SCHEDTYPE=*mode*
- (ORDERED). Specifying the (ORDERED) clause is equivalent to specifying MP_SCHEDTYPE=DYNAMIC and CHUNK=1. The parentheses are required.

Each clause has the same meaning as for the DOACROSS directive. Also as with the DOACROSS directive, the keyword LASTLOCAL is preferred to LAST LOCAL and the keyword LOCAL is preferred to PRIVATE.

The (ORDERED) clause is not a supported DOACROSS clause.

For more information on the AFFINITY clause and its syntax, see Section 5.2.2.1, page 152. For more information on the other clauses and their syntax, see Section D.3.1, page 271.

It is legal to declare a data item as LOCAL in a PDO directive even if it was declared as SHARED in the enclosing parallel region.

The END PDO directive is optional. If specified, this directive must appear immediately after the end of the DO loop. The optional NOWAIT clause specifies that each process should proceed directly to the code immediately following the directive. If you do not specify NOWAIT, the processes wait until all have reached the directive before proceeding.

Note: Localized ALLOCATABLE or POINTER arrays are not supported on the PDO directive.

The code in the following example is equivalent to a DOACROSS loop. In fact, the compiler recognizes this as a special case and generates the same (more efficient) code as for a DOACROSS directive.

```
          SUBROUTINE EX2(A,N)
          REAL A(N)
!$PAR PARALLEL LOCAL(I) SHARED(A)
!$PAR PDO
          DO I = 1, N
            A(I) = A(I) + 1.0
          END DO
!$PAR END PARALLEL
          END
```

D.5.6 PSECTION[S], SECTION, and END PSECTION[S] Directives

The PSECTION[S] and END PSECTION[S] directives delimit a parallel section construct and distribute code blocks to processes. These directives have an effect that is similar to the Fortran 90 SELECT construct. Each block of code is parceled out in turn to a separate thread.

The SECTION directive indicates a starting line for an individual section within a parallel section.

These directives must be enclosed within a parallel region delimited by PARALLEL and END PARALLEL directives.

These directives have the following format:

```
!$PAR PSECTION[S] [LOCAL var[, var] ...]

[!$PAR SECTION]

!$PAR END PSECTION[S] [NOWAIT]
```

var Specify a variable name for *var*. The `LOCAL` keyword has the same meaning as it does on the `DOACROSS` directive. The `LOCAL` keyword is preferred to `PRIVATE`. For more information on `LOCAL`, see Section D.3.1, page 271.

It is legal to declare a data item as `LOCAL` in a parallel sections construct even if it was declared as `SHARED` in the enclosing parallel region.

The optional `NOWAIT` clause specifies that each process should proceed directly to the code immediately following the directive. If you do not specify `NOWAIT`, the processes wait until all have reached the `END PSECTION` directive before proceeding.

Parallel sections can contain critical section constructs, but they cannot contain any of the following types of constructs:

- A `DO` loop that is preceded by a `PDO` directive
- A `DO` loop that is preceded by a `PARALLEL DO` or a `DOACROSS` directive
- Code delimited by `SINGLEPROCESS` and `END SINGLEPROCESS` directives

Each code block is executed in parallel (depending on the number of processes available). The code blocks are assigned to threads one at a time, in the order specified. Each code block is executed by only one thread.

For example, consider the following code:

```
          SUBROUTINE EX3(A,N1,B,N2,C,N3)
          REAL A(N1), B(N2), C(N3)
!$PAR PARALLEL LOCAL(I) SHARED(A,B,C)
!$PAR PSECTIONS
!$PAR SECTION
          DO I = 1, N1
            A(I) = 0.0
          END DO
!$PAR SECTION
          DO I = 1, N2
```

```
            B(I) = 0.5
          END DO
!$PAR SECTION
          CALL NORMALIZE(C,N3)
          DO I = 1, N3
            C(I) = C(I) + 1.0
          END DO
!$PAR END PSECTION
!$PAR END PARALLEL
          END
```

The first thread to enter the parallel section construct executes the first block, the second thread executes the second block, and so on. This example has only three sections, so if more than three threads are in the parallel region, the fourth and higher threads wait at the `!$PAR END PSECTION` directive until all threads are finished. If the parallel region is being executed by only two threads, whichever thread finishes its block first continues and executes the remaining block.

This example uses `DO` loops, but a parallel section can be any arbitrary block of code. Parallel constructs have significant overhead. Make sure the amount of work performed is enough to outweigh the extra overhead.

The sections within a parallel section construct are assigned to threads one at a time, from the top down. There is no other implied ordering to the operations within the sections. In particular, a later section cannot depend on the results of an earlier section, unless some form of explicit synchronization is used. If there is such explicit synchronization, you must be sure that the lexical ordering of the blocks is a legal order of execution.

D.5.7 `SINGLEPROCESS` and `END SINGLEPROCESS` Directives

The `SINGLEPROCESS` and `END SINGLEPROCESS` directives enclose a block of code that should be executed by only one process. These directives must be enclosed within a parallel region delimited by `PARALLEL` and `ENDPARALLEL` directives.

These directives have the following format:

```
!$PAR SINGLEPROCESS  [LOCAL var [, var] ...]

!$PAR END SINGLEPROCESS [NOWAIT]
```

var Specify a variable name for *var*. The LOCAL keyword has the same meaning as it does on the DOACROSS directive. The LOCAL keyword is preferred to PRIVATE. For more information on LOCAL, see Section D.3.1, page 271.

It is legal to declare a data item as LOCAL in a single process construct even if it was declared as SHARED in the enclosing parallel region.

The optional NOWAIT clause specifies that each process should proceed directly to the code immediately following the directive. If you do not specify NOWAIT, the processes wait until all have reached the END SINGLEPROCESS directive before proceeding.

This construct is semantically equivalent to a parallel section construct with only one section. The single process construct provides a more descriptive syntax.

The first thread to reach a single process section executes the code in that block. All other threads wait at the end of the block until the code has been executed.

Notice the use of the repetition of the IF test in the first parallel loop:

```
             IF (A(I,J) .GT. CUR_MAX) THEN
!$PAR CRITICAL SECTION
               IF (A(I,J) .GT. CUR_MAX) THEN
```

This practice is called *test&test&set*. It is a multiprocessing optimization. The following straightforward code segment is incorrect:

```
          DO I = 1, N
            IF (A(I,J) .GT. CUR_MAX) THEN
!$PAR CRITICAL SECTION
                  INDEX_X = I
                  INDEX_Y = J
                  CUR_MAX = A(I,J)
!$PAR END CRITICAL SECTION
            ENDIF
          ENDDO
```

Because many threads execute the loop in parallel, there is no guarantee that once inside the critical section, CUR_MAX still has the same value it did in the IF test outside the critical section (some other thread may have updated it). In particular, CUR_MAX may now have a value that is larger than A(I,J). Therefore, the critical section must be locked before testing the value of CUR_MAX. Changing the previous code into the following code works correctly, but suffers from a serious performance penalty: the critical section lock must be acquired and released (an expensive operation) for each element of the array:

```
            DO I = 1, N
!$PAR CRITICAL SECTION
                IF (A(I,J) .GT. CUR_MAX) THEN
                  INDEX_X = I
                  INDEX_Y = J
                  CUR_MAX = A(I,J)
                ENDIF
!$PAR END CRITICAL SECTION
         ENDDO
```

Because the values are rarely updated, this process involves a lot of wasted effort. It is almost certainly slower than just executing the loop serially.

Combining the two methods, as in the original example, produces code that is both fast and correct. If the IF test outside of the critical section fails, you can be certain that the values will not be updated and can proceed. You can expect that the outside IF test will account for the majority of cases. If the outer IF test passes, then the values might be updated, but you cannot always be certain. To ensure correctness, you must perform the test again after acquiring the critical section lock.

You can prefix one of the two identical IF tests with !$ to reduce overhead in the non-multiprocessed case.

Lastly, note the difference between the single process and critical section constructs. If several processes arrive at a critical section construct, they execute the code one at a time. However, they will all execute the code. If several processes arrive at a single process construct, only one process executes the code. The other processes bypass the code and wait at the end of the construct for the chosen process to finish.

D.5.8 Restrictions on the PCF Directives

The three work-sharing constructs, PDO, PSECTION, and SINGLEPROCESS, must be executed by all the threads executing in the parallel region or by none of the threads. The following is illegal:

```
      ...
!$PAR PARALLEL
          IF (MP_MY_THREADNUM() .GT. 5) THEN
!$PAR SINGLE PROCESS
```

```
              MANY_PROCESSES = .TRUE.
!$PAR END SINGLE PROCESS
          ENDIF
           ...
```

The preceding code cannot run successfully when more than six processors are used. One or more processes will be stuck at the `!$PAR ENDSINGLEPROCESS` directive waiting for all the threads to arrive. Because some of the threads never took the appropriate branch, they will never encounter the construct. However, the following kind of simple looping is supported:

```
       ...
!$PAR PARALLEL LOCAL(I,J) SHARED(A)
          DO I= 1,N
!$PAR PDO
            DO J = 2,N
       ...
```

The distinction here is that all of the threads encounter the work-sharing construct. They all complete it, and they all loop around and encounter it again.

This restriction does not apply to the critical section construct, which operates on one thread at a time without regard to any other threads.

Parallel regions cannot be nested inside of other parallel regions, nor can work-sharing constructs be nested. However, as an aid to writing library code, you can call an external routine that contains a parallel region even from within a parallel region. In this case, only the first region is actually run in parallel. Therefore, you can create a parallelized routine without accounting for whether it will be called from within an already parallelized routine.

Index