About This Guide

This guide tells you, a programmer using MIPSpro™ Fortran 90 or MIPSpro POWER Fortran 90, about the implementation details of these Silicon Graphics, Inc. compilers and their run−time support. This edition describes the products only for POWER CHALLENGE systems using IRIX 6.1.

What this Guide Contains

Here is an overview of the material in this book.

- Chapter 1 covers the phases of compilation and linking, and how to set up the run−time environment for a program.
- Chapter 2 tells how MIPSpro Fortran 90 implements the features that are defined as processor dependent in the language standard.
- Chapter 3 describes the programming interface between Fortran 90 and other languages, especially C and C++.
- Chapter 4 describes the scalar optimizations that are unique to the Fortran 90 compiler, and how you control and apply them.
- Chapter 5 describes the optimization support for function inlining and interprocedural analysis.
- Chapter 6 tells how to use special comments called directives to control optimization at a statement level.
- Chapter 7 describes the optimization features for parallelization and multiprocessors.
- Chapter 8 tells how to debug and run a parallelized program.
- Appendix A lists the run−time error codes that can occur.
- Appendix B covers some problems that can arise when converting Fortran 77 modules to Fortran 90.

Additional Reading

You should be aware of the following books that can be useful in your work with Fortran 90.

- The *Fortran 90 Handbook* by Adams, Brainerd, et. al. (McGraw−Hill 1992; ISBN 0−07−000406−4) is the recommended reference manual for the Fortran 90 language. This guide is designed to supplement the *Fortran 90 Handbook.*

  For each point at which the *Fortran 90 Handbook* mentions a processor dependency, this Guide has a heading explaining the Silicon Graphics implementation of that feature. In many cases, the paragraph number in the *Fortran 90 Handbook* is given following the heading.

- The *MIPS Compiling and Performance Tuning Guide* describes the common features of Silicon Graphics, Inc. compilers, including such points as:
  – an overview of the compiler system
  – the profiling and optimization facilities of the compiler system

– a general discussion of performance tuning

– the object file utilities, archiver, debugger, and other tools

- The *MIPSpro Porting and Transition Guide* describes the important differences between 32–bit and 64–bit systems, including

  – an overview of the 64–bit compiler system

  – language implementation differences

  – porting source code to the 64–bit system

  – compilation and run–time issues

- The *MIPSpro Assembly Language Programmer's Guide* describes the use of assembly language, and documents the standard instruction sequences used by all Silicon Graphics, Inc. compilers to call subroutines.
- The *CASEVision™/WorkShop User's Guide* introduces you to a powerful suite of tools for debugging and performance tuning.
- The *dbx User's Guide* lists the commands of the *dbx* debugger.

Internet Resources for Fortran 90 Users

The following sites point to resources of great value to users of Fortran 90.

- The Fortran Market™ is a World Wide Web (WWW) site that maintains links to, and information about, Fortran products, services, organizations, and consultants, and a directory of free software in Fortran 77 and Fortran 90.

  `http://www.swcp.com:80/fortran/`

- The Center for Scientific Computing in Finland maintains a WWW page with an extensive directory to software archives worldwide.

  `http://www.csc.fi/math_topics/FTP/index.html`

Conventions Used in This Guide

These are the typographical conventions used in this guide.

Purpose	Example
Names of Fortran keywords and procedures, and names defined in example code	A function such as AINT must be named in an INTRINSIC statement. The module NEW_TYPE defines type TAX_PAYER.
Names of commands and options entered on the IRIX command line	The compiler driver is *f90*. Use *elfdump –t* to list external names in an object file.
Titles of manuals	Refer to the *dbx User's Guide*.
A term defined in the hypertext glossary	This is a processor dependency.
Filenames and pathnames	The compiler automatically includes

	libftn90.so, *libftn.so*, and *libm.so* from */usr/lib64*.
Full lines of example code or commands, including variable elements you supply	`f90 -g -mips4`*sourcename*.f
Exact quotes of computer output	`off end of record`

Chapter 1

Compiling, Linking, and Running

This chapter describes how you compile Fortran 90 source modules, link them into executable units (programs or dynamic shared objects), and run them. These are the main sections:

- "Using the Driver" gives an overview of the operation of the *f90* compiler driver.
- "Linking" gives an overview of static and dynamic linking and the creation of dynamic shared objects (DSOs) from Fortran 90.
- "Driver Options" summarizes the many *f90* options in groups of related options.
- "Execution Environment" describes how you set up and execute a program, including memory allocation and file disposition.

Using the Driver

A Fortran 90 source module is converted from text to executable code in several phases. The phases are called and controlled by the *f90* command, which is conventionally called the driver for the compiler because it "drives" the phases through their execution. The phases of compilation are shown in Figure 1–1

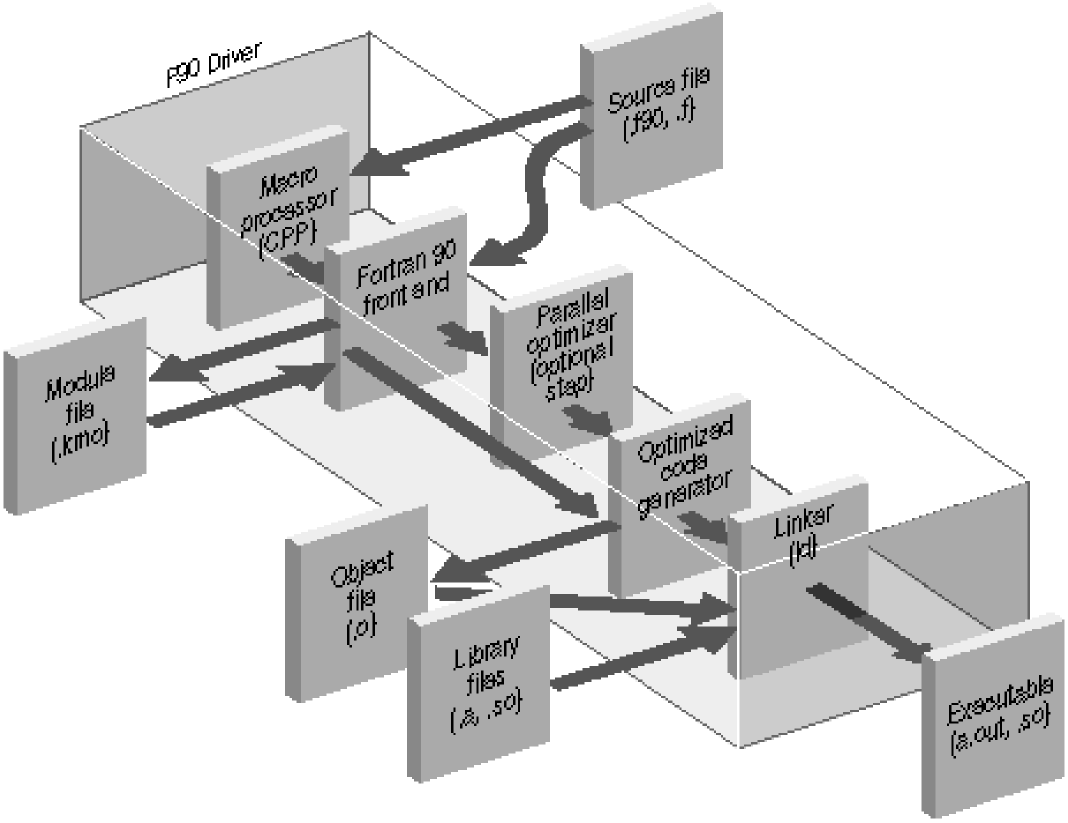

Figure 1–1 Compilation Phases

1. The first step of compilation is the *cpp* macro preprocessor. If you specify the *–nocpp* driver option, this step is skipped and the source file passes directly to the front end.
2. The Fortran 90 front end parses the syntax of the source text, reduces it to simpler forms, and detects syntax errors. This phase also writes a *module.kmo* file for each MODULE statement, and reads a *module.kmo* file for each USE statement.
3. The processed program can pass through the Parallel Optimizer if MIPSpro POWER Fortran 90 is installed.
4. The processed, parallelized program is converted to optimized machine language code in the code generator. If you specify the *–c* driver option, the machine code is written to a *name.o* file and the compilation ends.
5. The linker combines one or more object files into an executable. The executable is named *a.out* by default, but you can specify another name with the *–o* driver option.

Using Macro Processing

The *cpp* macro preprocessor can be used to include standard header files, to generate sequences of code, and to make parts of a program conditional on defined values. The following names are predefined by the driver (you can see their definitions using the *–show* driver option):

_ABI64=3 or 4	_COMPILER_VERSION=602
__DSO__	__host_mips
_LANGUAGE_FORTRAN	LANGUAGE_FORTRAN
_LANGUAGE_FORTRAN90	LANGUAGE_FORTRAN90
__mips=3 or 4	MIPSEB
_MIPSEB	_MIPS_ISA=3 or 4
_MIPS_FPSET=32	_MIPS_SZINT=32 or 64
_MIPS_SIM=_ABI64	_MIPS_SZLONG=64
_MIPS_SZPTR=64	_PIC
__sgi	_SYSTYPE_SVR4
_SVR4_SOURCE	__unix

Tip: The name LANGUAGE_FORTRAN (with and without a leading underscore) is defined by both Fortran drivers. Use it to control statements that are applicable to any Fortran program. LANGUAGE_FORTRAN77 is defined only by the *f77* driver, while the *f90* driver defines LANGUAGE_FORTRAN90. Use these names to generate code unique to one version of the language.

Linking

To link is to combine object files, resolving the connections between names that are defined in one file but called from another. Linking is discussed in greater detail in the *MIPS Compiling and Performance*

Tuning Guide.

Object Files

The object files that are linked come from one of these sources:

- Fortran 90 source files, compiled in this use of the driver
- Object files, previously compiled and saved as *name.o* files
- Archives of object files built with the *ar* command
- Dynamic shared objects (DSOs) built by the linker
- Library files, included by default or named with the *–l* driver option

A DSO is normally stored in a file with the suffix *.so*. The building of DSOs is described under "Creating Dynamic Shared Objects".

Archives are collections of object files in a single file, normally given a suffix of *.a*. The use of the archive–builder *ar* is covered in the *MIPS Compiling and Performance Tuning Guide.*

Any kind of object file—compiled *.o*, DSO *.so*, or archive *.ar*—can be specified as an input file on the *f90* command line.

Specifying the Location of Libraries

You specify the name of a library indirectly, using the *–L* and *–l* driver options. The compiler adds the prefix *lib* to the name specified with *–l* and searches for *libname.so* or *libname.a* in the directories where libraries are kept. For example, if you specify *–lblas*, the driver searches for a file *libblas.so* or *libblas.a*. If you specify *–L/usr/f90/objects*, the driver searches that directory when looking for library files.

The end result is that the library (which is simply a DSO or archive file) is included in the link phase of the compile. You can achieve the same result by giving the full name of the library as an input to *f90.*

The driver searches for libraries first in directories named by the *–L* option, and then in */usr/lib64* and in */usr/lib64/mips4*t.

The driver includes certain libraries by default:

- *libftn90.so* and *libftn.so*, Fortran run–time support
- *libm.so*, the math library
- *libc.so*, the C run–time library

Some libraries that are often included using the *–l* option are *libfpe* for floating–point exceptions, *libfGL*, the Fortran bindings to Open GL, and *libblas*, the BLAS math procedures.

Static and Dynamic Linking

When the object code of a procedure is included as part of the executable file, it is statically linked; that is, the connection between the call and the entry point is permanently fixed in the executable file.

Procedures defined in object files and archive files are always statically linked.

When the object code of a procedure is found in a DSO, the procedure is dynamically linked; that is, only the names of the procedure and the DSO are included in the executable program file. When the executable program is loaded, the DSO is also loaded, and the linkage between the call and the procedure is resolved at that time.

Most run−time libraries distributed with Fortran 90 are in DSO form, so the Fortran run−time support code and intrinsic procedures are dynamically linked.

Linking Multiple−File Programs

A large program is normally divided into many source files, each compiled separately. These files may all be written in Fortran 90; or some of them may be written in a different language—Fortran 77 or C, for example.

Exactly one of these files must be a main module. In Fortran 90 or Fortran 77, a main program is one that contains a PROGRAM statement. In C or C++, a main module contains a *main()* function definition.

To compile and link a multiple−file program, first compile each file using the −c driver option to produce an object file for that source file. Then compile all the *.o* files, using the driver for the language of the main module. (Any of the MIPSpro compiler drivers can perform the link step for any combination of object files.)

For example, imagine a program consisting of these files:

- A main module in Fortran 90 named *master.f90*
- A module of procedures in Fortran 77 named *procs.f*
- A module containing C functions, named *funcs.c*

This program could be compiled into an executable named *master* using the commands shown in Example 1−1.

Example 1−1 Compiling and Linking a Multiple−File Program

```
f90 −O3 −64 −c master.f90
f77 −O3 −64 −c procs.f
cc −O3 −64 −c funcs.c
f90 −o master master.o procs.o funcs.o
```

You perform the final step using the language for the main program because that driver includes the appropriate program−setup procedure and library code by default.

Creating Dynamic Shared Objects

The concepts and use of dynamic shared objects (DSOs) are covered in the *MIPS Compiling and Performance Tuning Guide*, and in the *dso*(5) and *rld*(1) reference pages. For example, the *MIPS Compiling and Performance Tuning Guide* covers details of how different versions of DSOs are created and used.

You can create a DSO based on one or more Fortran 90 source files, provided that they are not main program files. Then other Fortran 90 programs can link dynamically to the procedures in that DSO.

Note: The concept of a DSO and the concept of a Fortran 90 MODULE are quite distinct and should not be confused. You can link the object files from one or more MODULEs to form a DSO. This is a good method of packaging the object code of MODULEs. However, the compiler does not do this by default. A source file that defines one or more MODULEs is compiled to a single object file. You can statically link this file with other object files that need it, or you can link it to form a DSO.

The source of a small MODULE is displayed in Example B−12 Suppose this source is stored in a file *dgrtrig.f90*. It can be converted into a DSO using the commands shown in Example 1−2

Example 1−2 Compiling and Linking a DSO

```
f90 -O3 -64 -c dgrtrig.f90
ld -shared -64 dgrtrig.o -o dgrtrig.so
```

The compilation step produces the object file *dgrtrig.o*, which could be statically linked with any other program. (The compilation also produces a file *DEGREE_TRIG.kmo*, containing a summary of the module; this file is used to compile a USE DEGREE_TRIG statement that appears in any later compile.) The *ld* command in Example 1−2produces the DSO, *dgrtrig.so*.

Linking With DSOs

When a program uses a procedure that is defined in a DSO, you must take two steps.

1. Include the DSO as input when the program is linked.
2. Store the DSO where it can be found by the loader when the program is executed.

Linking a DSO as an Object File

You can simply name the DSO as an input file on the driver command line. The short program in Example 1−3uses the module that is converted to a DSO in Example 1−2

Example 1−3 Program That Uses a Module

```
use degree_trig
   print *,"sin",dgr,dsind(45.0_8)
end
```

Suppose that the DSO *dgrtrig.so* is in the current directory. Then the program in Example 1−3could be compiled as follows:

```
f90 -g -64 -o usedgr usedgr.f90 dgrtrig.so
```

While compiling the program, the compiler reads *DEGREE_TRIG.kmo* to learn the public names in the module. While linking the program, the linker resolves external references using entry point information in *dgrtrig.so*, but does not include the DSO contents in the executable file.

Tip: If you specify the full path to a DSO when linking, the path is recorded in the executable file and the loader does not need to search for the DSO when loading the program.

Linking a DSO as a Library

Another way to present object files to the linker is to include them as libraries. In order to do this,

- The DSO must have a name that begins with the letters *lib*.
- The DSO must be stored in a directory where the linker searches for library files.

For example, suppose that *dgrtrig.so* has been renamed to *libdgr.so* and stored in a directory */usr/local/f90/objects*. Now the program in Example 1−3can be compiled as follows:

```
f90 −g −64 −L/usr/local/f90/objects \
        −ldgr −o usedgr usedgr.f90
```

The *−L* option causes the linker to search for all needed libraries first in the specified directory. The *−ldgr* option tells it to include *libdgr.so* from that directory. Again, the linker resolves the external references from the contents of the DSO, but does not include the DSO in the executable file.

Loading DSOs at Execution Time

When a program that uses a DSO is loaded, the DSO files that it was linked with must be loaded also. (If they have already been loaded for use by some other program, they are not loaded again but simply shared.) In order to find them, the loader searches directories named in the LD_LIBRARY_PATH environment variable as well as other directories and environment variables documented in the *rld*(1) reference page.

Ordinarily you store the DSOs you create in one of the default directories, or in a directory named in LD_LIBRARY_PATH. For more about how DSOs are loaded, see the *MIPS Compiling and Performance Tuning Guide*.

Driver Options

This section contains an overview of the Fortran-specific driver options. The *f90*(1) reference page has a complete description of the compiler options. This discussion only covers the relationships between some of the options. In addition, you should review:

- the *MIPS Compiling and Performance Tuning Guide* for a discussion of the compiler options that are common to all MIPSpro compilers
- the *pfa*(1) and *fopt*(1) reference pages for options related to the parallel optimizer
- the *ld*(1) reference page for a description of the linker options
- chapters 5−7 of this Guide for options related to optimization features

Tip: The command *f90 −help* lists all compiler options for quick reference. Use the *−show* option to have the compiler display the exact default and nondefault options passed to each phase.

Compiling Low−Performance Programs

When you are compiling a program that does not have high performance requirements, you need only a very few compiler options. Examples of such programs include

- test cases used to explore algorithms or Fortran language features
- programs that are principally interactive
- programs you will execute under a debugger

In these cases you need only specify *−g* for debugging, the target machine architecture, and the word−length. For example, to compile a single source file to execute under *dbx* on a Power Challenge XL, you could use the following commands:

```
f90 -g3 -O0 -o testcase testcase.f
dbx testcase
```

A program compiled in this way will take little advantage of the performance features of the machine. In particular, its speed when doing heavy floating−point calculations will be far slower than the machine is capable of. For simple programs, that is usually not relevant.

Specifying Source File Format

The options summarized in Table 1−1 tell the compiler how to treat the program source file.

Table 1−1 Compile Options for Source File Format

Options	Purpose
−bytereclen	Always treat the RECL= specifier in OPEN as a count of bytes, including unformatted direct I/O
−fixedform, −col72, −col120, −extend_source, −noextend_source	Specify fixed−format input and set margin columns of source lines.
−freeform	Specify free−form input and Fortran 90 syntax.

Note: The Silicon Graphics Fortran 77 compiler supports a *−dlines* option that permits comment lines to begin with a "D" in column 1. This option is not supported by Fortran 90, and such comments will cause syntax errors.

Controlling the Macro Preprocessor

The options summarized in Table 1−2 direct the operation of the *cpp* preprocessor.

Table 1−2 Compile Options to Control *cpp*

Options	Purpose
−nocpp	Do not run *cpp*; pass source directly to front end.
−A:assertion	Add a *cpp* assertion.
−Dname, −Dname=def, −Uname	Define and undefine names to the C preprocessor.
−M, −Mupdate, −Mtarget filename	Run only the C preprocessor, requesting makefile−dependency output.

Specifying Compiler Input Files

The options summarized in Table 1−3tell the compiler what input files to use.

Table 1−3 Compile Options That Select Input Files

Options	Purpose
−I,*−Idir*	Specify location of all types of source inclusions—see note.
−nostdinc	Do not search for include files in default directories (search only directories given with −*I*options).
−Ldir	Specify location of library files.
−lname	Include *libname* in the link.
−nostdlib	Do not search for libraries in default directories (search only directories given with −*L*options).
−objectlist filename	Read *filename* for a list of object files to include in the link.

Note: The −I option is used to specify search locations for three kinds of included files:

- files included by the macro preprocessor from a #include statement
- files included by the compiler front end from an INCLUDE statement
- module (*.kmo*) files included by the compiler from a USE statement

Specifying Compiler Output Files

The options summarized in Table 1−4tell the compiler what output files to generate.

Table 1−4 Compile Options That Select Output Files

Options	Purpose
−c	Generate a single object file for each input file; do not link.
−E,*−P*	Run only the macro preprocessor and write its output to standard output (−*E*) or to *source.i* (−*P*).
−keep	Retain compiler intermediate and work files (debugging compiler problems only).
−listing	Request a listing file.
−LIST:listoption [,...]	Specify detailed options regarding the listing file.
−o	Specify name of output executable or DSO file.
−S	Write the generated object code in assembly−language form as *source.s*; do not link.

Specifying Target Machine Features

The options summarized in Table 1−5are used to specify the characteristics of the machine where the compiled program will be used. The TARG and TENV options are discussed in the *MIPS Compiling and*

Performance Tuning Guide.

Table 1−5 Compile Options for Target Machine Features

Options	Purpose
−mips3, *−mips4* (not suptd in 6.2 release)	The instruction architecture available in the target machine: use *−mips3* for MIPS R4x00 machines in 64−bit mode; use *−mips4* for MIPS R8000 and R10000 machines.
−n32, *−n64*	Specify the binary format. *−n64* (64−bit addressing) is the default.
−TARG:option,...	Specify certain details of the target CPU. Most of these options have correct default values based on the preceding options.
−TENV:option,...	Specify certain details of the software environment in which the source module will execute. Most of these options have correct default values based on other, more general values.

Specifying Memory Allocation and Alignment

The options summarized in Table 1−6 tell the compiler how to allocate memory and how to align variables in it. These options can have a strong effect on both program size and program speed.

Table 1−6 Compile Options for Memory Allocation and Alignment

Options	Purpose
−align8, −align16, −align32, −align64	Align all variables size *n* on *n*−byte address boundaries.
−d8, *−d16*	Specify the size of DOUBLE and DOUBLE COMPLEX variables.
−i2, *−i4*, *−i8*	Specify the size of INTEGER and LOGICAL variables.
−r4, *−r8*	Specify the size of REAL and COMPLEX variables.
−static	Allocate all local variables statically, not dynamically on the stack.
−Gsize, *−xgot*	Specify use of the global option table.

Specifying Debugging and Profiling

The options summarized in Table 1−7 direct the compiler to include more or less extra information in the object file for debugging or profiling.

Table 1−7 Compile Options for Debugging and Profiling

Options	Purpose
−g0, *−g2*, *−g3*, *−g*	Leave more or less symbol−table information in the object file for use with *dbx* or Workshop Pro *cvd.*
−p	Cause profiling to be enabled when the program is loaded.

For more information on debugging and profiling, see the *MIPS Compiling and Performance Tuning Guide.*

Specifying Optimization Levels

The MIPSpro Fortran 90 compiler contains three optimizer phases. One is part of the compiler "back end"; that is, it operates on the generated code, after all syntax analysis and source transformations are complete. The use of this standard optimizer, which is common to all MIPSpro compilers, is discussed in the *MIPS Compiling and Performance Tuning Guide.*

In addition, two phases of accelerators, one for scalar optimization and one for parallel array optimization, can be applied to the output of the front end. These optimizing phases are available only with the optional POWER Fortran 90 product. The options of the scalar optimizer are detailed in the *fopt* (1) reference page. The options of the parallel optimizer are detailed in the *pfa*(1) reference page.

The options summarized in Table 1−8 are used to communicate to the different optimization phases.

Table 1−8 Compile Options for Optimization Control

Options	Purpose
−O, *−O0*, *−O1*, *−O2*, *−O3*	Select basic level of optimization, setting defaults for all optimization phases.
−GCM:option,...	Specify details of global code motion performed by the back−end optimizer.
−OPT:option,...	Specify miscellaneous details of optimization.
−SWP:option,...	Specify details of pipelining done by back−end optimizer.
−sopt[*,option,...*]	Request execution of the scalar optimizer, and pass options to it.
−mp, *−pfa, −pfalist, −pfakeep*	Request automatic parallelization (POWER Fortran 90 only), and optionally retain its work files.
−WK,option,...	Pass options to the parallelizing optimizer.

The GCM, OPT, and SWP options are discussed in detail in the *MIPS Compiling and Performance Tuning Guide.*

For the options that can follow *−sopt*, refer to the *fopt*(1) reference page. For the options that can follow *−WK*, refer to the *pfa*(1) reference page. For examples of *−WK*, see Table 1−9.

When you use *−O* to specify the optimization level, the compiler assumes default options for the accelerator phases. These defaults are listed in Table 1−9. Remember, to see all options that are passed to a compiler phase, use the *−show* option.

Table 1−9 Defaults for Optimization Levels

Optimization Level	Power Fortran Defaults Passed
−O0	-WK,-roundoff=0,-scalaropt=0,-optimize=0
−O1	-WK,-roundoff=0,-scalaropt=0,-optimize=0
−O2	-WK,-roundoff=0,-scalaropt=0,-optimize=0
−O3	-WK,-roundoff=2,-scalaropt=3,-optimize=5
−sopt	-WK,-roundoff=0,-scalaropt=3,-optimize=5

In addition to optimizing options, the compiler system provides other options that can improve the performance of your programs:

- Two linker options**,** *-Gsize* and *-bestG*, control the size of the global data area, which can produce significant performance improvements. See Chapter 2 of the *Compiling, Debugging, and*

Performance Tuning Guide and the *ld*(1) reference page for more information.

- The *-jmpopt* option permits the linker to fill certain instruction delay slots not filled by the compiler front end. This option can improve the performance of smaller programs that do not require large blocks of virtual memory. See the *ld*(1) reference page for more information.

Controlling Compiler Execution

The options summarized in Table 1−10 control the execution of the compiler phases.

Table 1−10 Compile Options for Compiler Phase Control

Options	Purpose
−E, −P, −M	Execute only the C preprocessor.
−nocpp	Do not execute the C preprocessor.
−fe	Stop compilation immediately after the front−end runs (for compiler debugging only).
−sopt, −pfa	Run the parallelizing optimizer phase.
−S, −c	Stop compilation after the back−end runs, saving the output as assembly source (*−S*) or object code (*−c*).
−Yc,path	Load the compiler phase specified by *c* from the specified *path*.
−Wc,option,...	Pass the specified list of options to the compiler phase specified by *c*.

The −Yc and −Wc options take a single letter c to specify which compiler phase is meant. The letters are as follows:

p	cpp
f	front−end
b	back−end
a	assembler
l	linker
K	parallel optimizer, and Fortran 90 front end parser

For examples of using *−W* see Table 1−9.

Execution Environment

The execution environment of a Fortran 90 program includes its command−line arguments; the memory it can allocate; the files it uses; and the policies for handling execution errors.

Executing a Program

To execute a program, invoke it by name as a command. In the command environment where this happens, the run−time loader *rld* must be able to find all needed DSOs (see "Loading DSOs at Execution Time").

A program can recover the command−line arguments using the *getarg* subroutine and *iargc* function (see the *getarg*(3f) reference page). The program in Example 1−4lists its arguments using these functions.

Example 1−4 Program That Lists Command Arguments

```
external getarg, iargc
integer iargc
character(80) anarg
do j = 1, iargc()
   call getarg(j,anarg)
   write (6,"(i3,' = ',a)") j, anarg
end do
end ! program
```

Program and Memory Size Limits

In general, a program may use as much memory as necessary. There are some limits on individual allocations to 2 gigabytes (2,048 megabytes); and IRIX places global limits on run−time use of stack and allocated memory.

Allocatable Sizes

There is no limit on the size of a variable created with the ALLOCATE statement, other than the IRIX limit on total data space and total swap space.

Static and Common Sizes

When compiling with the −*static*flag, global data is allocated as part of the compiled object (*.o*) file. The total size of any single *.o* file may not exceed 2 GB. However, the total size of an executable program or a DSO linked from multiple *.o* files is not restricted.

An individual common block may not exceed 2 GB. However, you can declare multiple common blocks, each having that size.

Local Variable (Stack Frame) Sizes

An array allocated on the process stack (that is, as a local variable of a procedure) must not exceed 2 GB, but the total of all local variables can exceed that limit. The subroutine in Example 1−5has several gigabytes of local space.

Example 1−5 Subroutine With Large Local Variable Space

```
subroutine bloat(arg)
   integer arg
   integer, parameter :: ndim = 16380
   integer(8) xmat(ndim,ndim), ymat(ndim,ndim), &
              zmat(ndim,ndim)
   xmat = arg
   return
```

```
end
```

However, there is no limit on the size of an array that is passed as an argument. The function inExample 1−6takes a 34 GB argument. An argument of this size could be created with ALLOCATE.

Example 1−6 Function Taking a Large Argument

```
function bigarg(rodan)
   integer rodan(8589934592_8)
   bigarg = rodan(4294967296_8)
end
```

Checking and Setting IRIX Limits

No user can have a stack larger than the current IRIX stack allocation limit nor can allocate total data memory greater than the IRIX data limit. Both these limits can be examined using the *sh* or *csh* command *limit.*

```
% limit stacksize
stacksize 512000 kbytes
% limit datasize
datasize unlimited
```

Note: There is no warning or diagnostic message when a program exceeds the data or stack limit. The program ends with a segmentation fault.

You can use the limit command to set a smaller or larger limit, up to a system maximum. The system maximum (*rlimit_data_max* or *rlimit_stack_max*) is set using the *systune* command (refer to the *systune* (1) reference page).

Limits of Swap Storage

Regardless of the amount of space that a user is permitted by the IRIX limits, the total of all writable virtual memory for all processes in the system cannot exceed the virtual swap allocation. The total of all virtual memory actually written into cannot exceed the actual swap allocation. Thus a program that allocates a very large array may not be able to start running because it exceeds the size of virtual swap. A program may be able to allocate a very large array (which is not larger than the virtual swap allocation), but then may terminate while assigning values into the array because it has exceeded the size of actual swap space.

The current swap allocations can be checked and changed using the *swap* command; refer to the *swap*(1) reference page.

Connecting Files

This section covers the details of using files at runtime.

Preconnected Files

Table 1−11shows the standard preconnected files at program start.

Table 1–11 Preconnected Files

Unit #	Unit
5	Standard input
6	Standard output
0	Standard error

All other units are also preconnected when execution begins. Unit n is connected to a file named *fort.n*. These files need not exist, nor will they be created unless their units are used without first executing an OPEN with an explicit filename. The default connection is for sequentially formatted I/O.

Filename Syntax

In the FILE= argument of OPEN or INQUIRE, you may specify any valid IRIX path or filename. The string is case–sensitive. If the string begins with a dot or a slash, it is treated as an absolute path; otherwise it is interpreted in the context of the current working directory. (You can change the current working directory dynamically using the *chdir()* function; see "Support for IRIX Kernel Functions".)

Three predefined filenames are supported. When one of the following names is specified, the file is opened to the system I/O stream shown:

SYS$INPUT	standard input (same as logical unit 5)
SYS$OUTPUT	standard output (same as logical unit 6)
SYS$ERROR	standard error (same as logical unit 0)

By using these names you can associate an arbitrary unit number to a standard stream. However, if you access a stream alternately through different unit numbers, the results are unpredictable.

File Positions

The I/O system positions a file at start of file for both input and output. The execution of an OPEN statement followed by a WRITE on an existing file causes the file to be overwritten, erasing any data in the file.

In a program called from a parent process, units 0, 5, and 6 remain where they were positioned by the parent process.

Default File Status and Action

When the parameter STATUS="UNKNOWN" is specified in an OPEN statement, the following occurs:

- If the file does not exist, it is created and positioned at start of file.
- If the file exists, it is opened and positioned at start of file.

When a file is opened without specifying the ACTION= argument, the default action is READWRITE.

Run–Time Error Handling

When the Fortran 90 run–time system detects an error that is not handled by the program, the following action takes place:

- A message describing the error is written to the standard error unit (unit 0). See Appendix A, "Run−Time Error Codes,"for a list of the error messages.
- A core file is produced if the f77_dump_flag environment variable is set (the variable has the same name for both Fortran 77 and Fortran 90).

You can use *dbx* to locate the point of failure in the core file. To invoke *dbx* using the core file, enter this command:

`% dbx` *executable−file*

where *executable−file* is the name of the executable program. The *dbx* command *where* displays a stack trace showing the procedures that were active when the error took place.

Tip: When a program ends with a segmentation fault, a likely cause is that a memory size limit was exceeded. See "Program and Memory Size Limits".

Floating Point Exceptions

The library *libfpe* provides two methods for handling floating point exceptions.

Note: Owing to the different architecture of the MIPS R8000 and R10000 processors, library *libfpe* is not available with the current compiler. It will be provided in a future release. When porting Fortran 77 programs that depend on trapping exceptions using the facilities in *libfpe*, you will have to temporarily change the programs to do without it.

The library provides the subroutine **handle_sigfpes** and the environment variable **TRAP_FPE**. Both methods provide mechanisms for handling and classifying floating point exceptions, and for substituting new values. They also provide mechanisms to count, trace, exit, or abort on enabled exceptions. See the *handle_sigfpes*(3F) reference page for more information

Chapter 2

Implementation Details

This chapter documents the relationship of MIPSpro Fortran 90 to the ANSI standard for the Fortran 90 language. It includes these main topics:

- "Conformance to the Fortran 90 Language Standard" documents the standard compliance and the extensions supported.
- "Support for IRIX Kernel Functions" documents the IRIX functions that are available as library routines to a Fortran program.
- "Processor–Dependent Features"documents how this compiler implements features the standard calls processor–dependent.

Conformance to the Fortran 90 Language Standard

MIPSpro Fortran 90 is a complete implementation of ANSI Fortran 90. It supports all of the language features as defined by the standard, and allows only minor syntactic extensions beyond the standard.

MIPSpro Fortran 90 does not support such common language extensions as HPF (High Performance Fortran) directives. Language extensions present in Cray™ and DEC™ Fortran 90 compilers are also not supported.

The version of MIPSpro Fortran 90 and MIPSpro Power Fortran 90 described in this book support only Silicon Graphics, Inc. Power Challenge and Power Indigo2 systems (that is, systems based on the MIPS R8000 processor) running IRIX 6.0.x. The compiler generates only 64–bit executables.

Support for Multiprocessing

MIPSpro Fortran 90 supports multiprocessing with user–inserted parallel directives. These directives follow the recently adopted PCF (Parallel Computing Forum) X3H5 guidelines. (See "Using PCF Directives".)

MIPSpro POWER Fortran 90 is an extended version of MIPSpro Fortran 90 that supports the same language and directives, but adds automatic optimization for multiprocessing. This compiler analyzes a serial program for potential parallelization and when possible, generates an executable that can utilize multiple processors when executed in a Silicon Graphics, Inc. computer that has multiple CPUs in sufficient numbers.

Syntax Extensions

The following nonstandard syntax features are accepted by the compiler in order to make it simpler to port programs from Fortran 77.

- Asterisk notation can be used to signify precision of a numeric type. For example, INTEGER*8 is taken as INTEGER(8). (See "Differences in Scalar Declarations".)
- When writing a quad–precision literal floating point number, "Q" can be used as the exponent

delimiter. For example, 1Q6 is taken as 1E6_16. (See "Differences in Scalar Declarations".)

- Hollerith−form character literal values are accepted in integer expressions. (See"Character and Hollerith Literals".)
- A numeric variable can be specified as the filename value in an OPEN statement. (See "Numeric Variable for FILE".)
- Special format modes are permitted in the OPEN statement. (See "Special File Formats".)

None of these extensions adds functionality to the language; they only make it easier to port existing programs that depend on Fortran 77 extensions. Because they are nonportable, you should avoid using them when writing new code.

Support for IRIX Kernel Functions

Some commonly−used IRIX system functions have been made available to Fortran programs. System functions are facilities that are provided by the IRIX system kernel directly, as opposed to functions that are supplied by library code. For an overview of system functions, see the *intro*(2) reference page.

Table 2−1summarizes the functions in the Fortran run−time library. In general, the interface routine has the same name as the system function when called from a C program.

For details on any function, use the command *man*; for example, to get details on the *fcntl()* function, use *man fcntl.* This can yield as many as three reference pages with the same name; for example, *man chmod* produces *chmod*(1) for the user command, *chmod*(2) for the C function, and *chmod*(3f) describing the Fortran interface.

Table 2−1 Summary of System Interface Library Routines

Function	Purpose
abort	abnormal termination
access	determine accessibility of a file
acct	enable/disable process accounting
alarm	execute a subroutine after a specified time
barrier	perform barrier operations
blockproc	block processes
blockprocall	block processes
brk	change data segment space allocation
chdir	change default directory
chmod	change access mode of a file
chown	change ownership of a file
chroot	change root directory for a command
close	close a file descriptor
creat	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	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	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
fstat64	get file status—64−bit integers
ftell	reposition a file on a logical unit
ftell64	reposition a file on a logical unit for 64−bit architecture
gerror	get system error message text
getarg	return command line arguments
getc	get a character from a logical unit
getcwd	get pathname of current working directory
getdents	read directory entries
getegid	get effective group ID
gethostid	get unique identifier of current host
getenv	get value of environment variables
geteuid	get effective user ID
getgid	get user or group ID of the caller
gethostname	get current host ID
getlog	get user's login name
getpgrp	get process group ID
getpid	get process ID
getppid	get parent process ID
getsockopt	get options on sockets
getuid	get user or group ID of caller
gmtime	return system time
iargc	return number of command line arguments
idate	return date or time in numerical form
ierrno	get system error message number
ioctl	control device
isatty	determine if unit is associated with tty
itime	return date or time in numerical form
kill	send a signal to a process
link	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

lstat64	get file status—64−bit integers
ltime	return system time
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_rcle_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	open a file
oserror	get/set system error
pause	suspend process until signal
perror	write system error message to stderr (unit 0)
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
read	read from a file descriptor
readlink	read value of symbolic link
rename	change the name of a file
rmdir	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
setblockproccntall	set semaphore count
setgid	set group ID
sethostid	set current host ID
setoserror	set system error
setpgrp	set process group ID

setsockopt	set options on sockets
setuid	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	suspend execution for an interval
socket	create an endpoint for communication TCP
sproc	create a new share group process
stat	get file status
stat64	get file status—64−bit integers
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	return system time (available as both function and subroutine)
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	get and set file creation mask
umount	dismount a filesystem
unblockproc	unblock processes
unblockprocall	unblock processes
unlink	remove a directory entry
uscalloc	shared memory allocator
uscalloc64	shared memory allocator in 64−bit environment
uscas	compare and swap operator (default integer arguments)
uscas32	compare and swap operator (INTEGER(4) arguments)

uscasinfo	compare and swap information word of shared arena
usclosepollsema	detach file descriptor from a pollable semaphore
usclrerror	disable messages from shared arena (usinit reference page)
usclrtrace	disable tracing in shared arena (usinit reference page)
usconfig	semaphore and lock configuration operations
uscpsema	acquire a semaphore
uscsetlock	unconditionally set lock
usctllock	lock control operations
usctlsema	semaphore control operations
usdetach	release shared arena
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
usopenpollsem	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
usseterror	enable messages from memory arena (usinit reference page)
ussetlock	set lock
ussettrace	start tracing in memory arena (usinit reference page)
ustestlock	test lock
ustestsema	return value of semaphore
usunsetlock	unset lock
usvsema	free a resource to a semaphore
uswsetlock	set lock
wait	wait for a process to terminate
write	write to a file

Processor−Dependent Features

The Fortran 90 language standard leaves some details of language semantics to be defined by the compiler. The Fortran 90 language standard and the *Fortran 90 Handbook* (see "Additional Reading") refer to each such detail as a processor dependency.

For each point at which the *Fortran 90 Handbook* mentions a processor dependency, this Guide has a heading explaining the Silicon Graphics implementation of that feature. In many cases, the paragraph number in the *Fortran 90 Handbook* is given following the heading.

Note: The behavior of processor−dependent features is by definition nonportable. Fortran 90 systems from other vendors may not behave the same way. Furthermore, future implementations of Silicon Graphics, Inc. Fortran 90 may behave differently in subtle ways. You should never base the correctness of a program on the behavior of a processor dependency.

Standard and Intrinsic Modules

(This topic amplifies topic 1.7 in the *Fortran 90 Handbook.*)

No standard or intrinsic modules are shipped with MIPSpro Fortran 90. One module has been proposed as a standard to ISO (ISO/IEC 1539−2:1994(E)). It is a module that implements varying−length strings of characters. The Fortran 90 source of an exemplary implementation of this module is available on the World Wide Web. See the sources mentioned under "Internet Resources for Fortran 90 Users".

Supported Data Types

(This topic amplifies topics 2.9, 3.1, 4.1, 4.3, and 13.3 in the *Fortran 90 Handbook.*)

MIPSpro Fortran 90 supports the following data types:

- INTEGER with kind−parameters of 1, 2, 4, and 8
- REAL with kind−parameters of 4, 8, and 16

  DOUBLE PRECISION is a synonym for REAL(8)
- COMPLEX with kind−parameters of 4, 8, and 16

  DOUBLE COMPLEX is a synonym for COMPLEX(8)
- LOGICAL with kind−parameters of 1, 2, 4, and 8
- CHARACTER

Except for COMPLEX, the supported kind−parameters (1, 2, 4, 8 and 16) represent the size of a scalar value in bytes and the default alignment of one scalar item in memory. For COMPLEX, the kind−parameter represents the size and alignment of each of the components of the number. (This can be slightly confusing when migrating from Fortran 77. In Fortran 77, a complex number composed of two REAL*8 numbers is a COMPLEX*16 variable. But in Fortran 90, a complex number composed of two REAL(8) numbers is a COMPLEX(8) variable.).

Numeric Precision

(This topic amplifies topics 4.2.2, 4.3, and 5.1 in the *Fortran 90 Handbook.*)

INTEGER and LOGICAL values are stored as signed binary integers with the size and alignment specified by the kind–parameter. The possible integer values are:

INTEGER(1)	-128…127
INTEGER(2)	-32,768…32,767
INTEGER(4)	$-2^{31} \ldots 2^{31}$ -1
INTEGER(8)	$-2^{63} \ldots 2^{63}$ -1

REAL(4) and REAL(8) values are stored in IEEE 754 format. REAL(16) values are not stored in IEEE 754 form, but rather are stored as the diffference between two values (see the *math*(3m) reference page). COMPLEX values consist of two REAL values of the specified kind in adjacent locations. The precisions supported are shown in Table 2–2

Table 2–2 Floating–Point Precision

Data Type	Exponent	Fraction	Approximate Minimum	Approximate Maximum
REAL(4)	8 bits	24 bits	1.17e–38	3.4e+38
REAL(8)	11 bits	53 bits	2.22e–308	1.797e+308
REAL(16)	11 bits	107 bits	1.805e–276	1.797e+308

Character Values and Literals

(This topic amplifies topics 3.3, 4.3, and 5.1 in the *Fortran 90 Handbook.*)

The CHARACTER type supports only 8–bit ASCII values. Any 8–bit value can be stored in a CHARACTER variable using the CHAR intrinsic function. Only one kind–parameter is supported for CHARACTER data: the parameter 1, signifying 8–bit ASCII. For example, the lines

```
CHARACTER(80,1) MSG
MSG = 1_"Enter File Name"
```

show the declaration of a CHARACTER variable with an explicit kind and the assignment of a literal with an explicit kind.

Literal character values are represented in a source expression using the standard Fortran 90 syntax as described in section 4.3.5.4 of the *MIPSpro Fortran 90 Handbook.* Any character that can be entered from the keyboard can be contained within a literal string, with the exception of the ASCII newline character (control–J, 10 decimal), which signals the end of the line and the end of the logical statement.

INCLUDE Processing

(This topic amplifies topic 3.5 in the *Fortran 90 Handbook.*)

The Fortran 90 INCLUDE statement causes a specified file to be read by the compiler. When the argument of INCLUDE specifies an absolute path—that is, if it begins with a slash or a dot—the specified file is included.

```
INCLUDE "../hdrs/header1"
```

When the argument of INCLUDE specifies only a filename or a relative pathname, the compiler searches for the specified file in directories specified by the −I option and in standard locations for included files (see "Specifying Compiler Input Files").

```
INCLUDE "hdrs/header1"
```

There is a limit of 98 on the depth of nesting included files. That is, the compiler supports a total of 99 source files concurrently open: the original source file and up to 98 included files.

Assumed−Shape and Deferred−Shape Arrays

(This topic amplifies topics 5.2, 5.3, and 12.5.3 in the *Fortran 90 Handbook.*)

Fortran 90 supports arrays whose actual shape is not known until run time. Assumed−shape arrays are dummy arguments whose shape is not declared in the procedure—they take the shape of the actual parameter passed. Deferred−shape arrays are array variables with the POINTER or ALLOCATABLE attribute—their shapes are specified when memory is allocated for them.

When a program accesses an element of an assumed−shape or deferred−shape array, the access is not direct, but indirect through a descriptor block in memory. As a result, two to four times as many machine instructions are needed to retrieve an element from one of these arrays than from an array of declared size.

In addition, there is no interface for passing an assumed−shape or deferred−shape array as an argument to a function written in C (as discussed in "Unsupported Array Parameters").

Treatment of the SAVE Attribute

(This topic amplifies topic 5.6.4 in the *Fortran 90 Handbook.*)

The SAVE attribute is implemented in different ways for different types of variables. This topic describes the one implementation of the compiler. However, this information is subject to change in future implementations. It would be unwise to write code that depended directly on this information.

Global Variables

Variables declared at the global level of a source unit, but not within a MODULE, are typically allocated as part of the main procedure stack frame. This stack frame persists throughout the program execution.

When SAVE is specified for a global variable, the variable is allocated instead in a block storage section—either *.bss* or *.sbss*, depending on its length. This segment persists throughout the program. A side−effect of allocation in block storage is that these sections are initialized to binary zero when they are created at program load time. (Clearly, to rely on this effect would be to make your program nonportable.) For more information on the management of block storage sections, see the *MIPSpro Assembly Language Programmer's Guide.*

MODULE Global Variables

Variables declared at the outer level of a MODULE are typically allocated as COMMON segments whose names are formed from the module name. This ensures that only one instance of a module variable exists in the executable program, no matter how many object files refer to that module in a USE statement.

The use of SAVE with MODULE variables does not change this typical allocation, since common segments persist throughout the execution of the program.

Procedure Local Variables

Variables declared local to any procedure are allocated as part of the procedure's stack frame. They are released when the procedure returns.

Note: This means that in the current implementation, any function can be called recursively. However, Silicon Graphics can change the implementation of local variables in a future implementation. Always specify RECURSIVE with a function that you intend to be recursive.

A local variable with the SAVE attribute is not allocated in the procedure stack frame, but rather in a block storage segment. This means that there is only one fixed instance of the variable for the procedure. It retains its most recent value from one call of the procedure to another, or from one level of a recursive function to the next.

If different procedures have local variables with the same name and the SAVE attribute, each procedure has a unique block−storage allocation for its variable.

Nonstandard Intrinsic Procedures

(This topic amplifies topic 5.7.2 in the *Fortran 90 Handbook.*)

MIPSpro Fortran 90 does not provide any intrinsic functions or subroutines other than the ones standardized in Fortran 90. For information on porting a Fortran 77 program that relies on nonstandard intrinsics, see "Differences in Intrinsic Functions".

MIPSpro Fortran 90 does provide a large number of library functions for access to IRIX kernel functions (see "Support for IRIX Kernel Functions"). These are not "intrinsic" since their names are not recognized automatically by the compiler.

Status From ALLOCATE and DEALLOCATE

(This topic amplifies topic 6.5.1 in the *Fortran 90 Handbook.*)

The status returned from ALLOCATE or DEALLOCATE is 0 when the operation is performed, and 1 when it fails.

When the operation fails owing to a logical error such as deallocating a variable that has not been allocated, no other information is available. When an failure is due to a system condition such as a lack of memory, the system error number is recorded. You can retrieve the system error number with the *ierrno()* function, or display an error message on unit 0 using *perror()* (see the *perror*(3f) reference page).

Example 2−1 Displaying a Message on Allocation Failure

```
ALLOCATE (AV(SZ),STAT=STAT)
IF (0 /= STAT) THEN
   IF (0 /= IERRNO()) THEN
```

```
      CALL PERROR("Allocating vector")
   ELSE
      WRITE (6,*)"Unknown error allocating vector"
   END IF
END IF
```

Partial Evaluation of Array Constructors

The *MIPSpro Fortran 90 Handbook* notes that the Fortran 90 standard permits the processor to skip the evaluation of parts of an array constructor under certain conditions. It is true that, in certain limited cases, MIPSpro Fortran 90 does omit evaluation of zero−sized array constructor elements. However, the rules for when this is done are complex and subject to change. Furthermore, the standard does not specify the order in which expressions and sub−expressions within a constructor are evaluated.

In general, you should assume that any expression within an array constructor will be evaluated. You should never write code that relies on the compiler not evaluating a constructor expression, or on its evaluating the constructor in any particular order.

Status From I/O Statements

(This topic amplifies topic 9.2.2 in the *Fortran 90 Handbook*.)

The values left in an IOSTAT variable following input are as follows:

- end of file produces −1
- end of record (nonadvancing input) produces −2

Processor−Dependent Error Codes

(This topic amplifies topic 9.2.3 in the *Fortran 90 Handbook*.)

For all I/O status values greater than zero (error codes), see Appendix A, "Run−Time Error Codes." Regarding a specific case mentioned in the *MIPSpro Fortran 90 Handbook*, a request for input beyond the record size when PAD="NO" causes return of an I/O status code of 177, if an IOSTAT variable is supplied.

Default Output Record Lengths

(This topic amplifies topic 9.5.5 in the *Fortran 90 Handbook*.)

When a file is opened for sequential output and no specific RECL= argument is given, there is no maximum record length. The length of each record is determined by the data written to it.

The value of the RECL= specifier is interpreted as a count of bytes for formatted I/O and for sequential unformatted I/O. The value returned by INQUIRE(IOLENGTH) is always in terms of bytes.

In the single case of unformatted direct I/O, RECL= is interpreted as a count of 32−bit words (this behavior is for compatibility with some heritage Fortran 77 programs). You can change this behavior, forcing RECL= to be interpreted as a byte count in every case, by specifying the *−bytereclen* driver option (see "Specifying Source File Format").

Implementation of Sign Edit Codes

(This topic amplifies topic 10.9.4 in the *Fortran 90 Handbook.*)

The treatment of the Sign edit code "S" is processor dependent. MIPSpro Fortran 90 does not display a positive sign for "S" editing.

Arguments to the Main Program

(This topic amplifies topic 11.2.1 in the *Fortran 90 Handbook.*)

Although according to the Fortran 90 standard a main program has no provision for receiving arguments, the main program is invoked from the IRIX command line, and it can access the command−line arguments using the *getarg()* library function (see the *getarg*(3) reference page). That function may not be available in systems from other vendors.

Implementation of MODULE and USE

(This topic amplifies topic 2.1 and 11.6.6 in the *Fortran 90 Handbook.*)

MIPSpro Fortran 90 implements the MODULE statement automatically. When the compiler front−end encounters a MODULE statement, it checks the syntax of the contents of the module that follow.

When the syntax is valid, the compiler writes a *.kmo* file in the current directory. This is a small, printable file containing a summary of the accessible identifiers declared in the module. The name of the file is the name given in the MODULE statement, in uppercase letters.

When the compiler encounters a USE statement, it searches for a *modname.kmo* file (see "Specifying Compiler Input Files" for the search path). The contents of the file establish the names provided in that module.

As a result of these rules, a MODULE must be compiled before any USE statement for the module can be compiled.

The object code contained in a MODULE is generated when the MODULE is compiled. Procedures defined within the MODULE are named *modname$procname_* in the object file; that is, the names are qualified by the module name. In other respects, these procedures generate code no different from that of other procedures in the same object file. The object code of a MODULE is stored and linked just like any other unit of object code. Compiled modules are not stored in any special location or format. (For hints on converting a MODULE to a DSO, see "Creating Dynamic Shared Objects".)

To prepare and distribute a functional package based on a MODULE,

1. compile the module source, yielding a *.kmo* file and an object file
2. optionally, link the object file as a DSO
3. distribute only the *.kmo* and the object file (or DSO)

Using this approach, you do not have to distribute the source code of the module, only the summary *.kmo* file. The binding between accessible names in the module and the code that uses them is established at link time (or at load time for a DSO). As a result, you can distribute updated object files and the client programs do not have to be recompiled—as long as the names and data types of the accessible names are not changed.

Use of the Keyword RECURSIVE

(This topic amplifies topic 12.1.3 in the *Fortran 90 Handbook*.)

The current release of MIPSpro Fortran 90 allocates procedure local variables on the process stack (see "Treatment of the SAVE Attribute"). However, this is not a defined programming interface. You should always use the keyword RECURSIVE for any function that can be called recursively.

Array References in Statement Functions

(This topic amplifies topic 12.3.4 in the *Fortran 90 Handbook*.)

The *Fortran 90 Handbook* indicates (section 12.3.4, rule 2) that there is ambiguity in the language standard regarding the use of array arguments to intrinsic functions (within the context of a statement function). MIPSpro Fortran 90 does not permit the use of array arguments to intrinsics within a statement function.

Implementation of RANDOM_NUMBER

(This topic amplifies topic A.83 in the *Fortran 90 Handbook*.)

The pseudorandom number generator used to implement the RANDOM_NUMBER function was originally described in "Toward a Universal Random Number Generator" by George Marsaglia and Arif Zaman (Florida State University Report: FSU−SCRI−87−50 (1987)). It was later modified by F. James and published in "A Review of Pseudo−Random Number Generators." It is thought to be the best available random generator. It passes all tests for random number generators and has a period of 2^{144}.

The algorithm is a combination of a Fibonacci sequence (with lags of 97 and 33, and operation "subtraction plus one, modulo one") and an arithmetic sequence using subtraction.

The RANDOM_SEED function takes two seed integers,

- 0 <= *ij* <= 31328
- 0 <= *kl* <= 30081

A change in either *ij* or *kl* produces a new sequence of length approximately 10^{30}. For example, if several groups are working on parts of the same calculation, each group could be assigned its own *ij* seed number, leaving it with 30,081 choices for *kl*.

To test the random generator, use the code shown in Example 2−2

Example 2−2 Test of RANDOM_NUMBER

```
integer seed(2)
   real first20k(20000), next6(6)
```

```
   seed(1) = 1802
   seed(2) = 9373
   call random_seed(put=seed(1:2))
   call random_number(harvest=first20k)
   call random_number(harvest=next6)
   do j=1,6
      write (6,"(F10.1)") (4096*4096*next6(j))
   end do
end
```

The output from this test should be the following numbers:

```
 6533892.0
14220222.0
 7275067.0
 6172232.0
 8354498.0
10633180.0
```

Chapter 3

Linkage to Other Languages

Occasionally it is not possible to implement all of an application in Fortran 90. You may need to call external procedures written in C, C++, or some other language—or you may need to call a Fortran 90 procedure from one of those languages. This chapter focuses on the interface between Fortran and the most common other language, C.

External and Public Names

(This topic amplifies topic 3.1.1 in the *Fortran 90 Handbook*.)

When your Fortran 90 program defines the body of a procedure, the compiler places the name of the procedure, as a character string, in the object module it generates. This is a public name, which is accessible to other modules.

When your Fortran 90 program declares a procedure as EXTERNAL, the compiler places the name of the procedure, as a character string, in the generated object module. This is an external name, which is needed by the module but not defined in it. Names of common blocks and names of data and procedures declared within modules are also external names. (You can display the public and external names defined in a module using the *nm* utility, as discussed in the *MIPS Compiling and Performance Tuning Guide*.)

It is up to the IRIX linker, *ld*, 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 linker.

How Fortran 90 Handles External and Public Names

The Fortran compiler forces all input source text (other than the contents of character literals) to lowercase as the first step of compilation. As a result, it changes the names of procedures and named common blocks while it translates the source file. As recorded in the object file, these names are changed in 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

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 problem when linking modules compiled by Fortran 90 or Fortran 77, since the same convention is used for both the public and external names. Therefore the names match and the linker can resolve them.

The names of procedures defined within a MODULE are qualified with the module name, also in lowercase. If SUBROUTINE MATMUL is defined in MODULE MATRIX, its public name string is *matrix$matmul_* ; that is, the module name, ‘$,’ the procedure name, and an underscore.

Note: Some Fortran 77 compilers from Silicon Graphics support the *−U* compiler option, telling the compiler to not force all uppercase input to lowercase. As a byproduct, it becomes possible to put mixed−case public names in the object file. This driver option is not supported by MIPSpro Fortran 90.

Note: Some Fortran 77 compilers from Silicon Graphics take the use of the '$' character as the final letter of a procedure name as a signal to suppress the underscore in the public name. However, the '$' is not allowed at any position of a name in MIPSpro Fortran 90. There is no way to suppress the final underscore in an external name.

Naming Fortran Subprograms From C

In order to call a Fortran 90 subprogram from a C module you must spell the name the way the Fortran compiler spells it—using all lowercase letters and a trailing underscore. A subprogram declared as follows:

```
SUBROUTINE HYPOT()
```

must be declared in a C function as follows (lowercase with a trailing underscore):

```
extern int hypot_()
```

Note: Since the public name of a procedure in a MODULE contains a "$" character, and since C does not allow "$" in identifiers, it is not possible to call directly from a C program to a MODULE procedure.

Naming C Functions From Fortran

The public names of C functions can have uppercase or mixed−case names, and they have terminal underscores only when the programmer writes them that way. However, there is no way by which you can make the MIPSpro Fortran 90 compiler generate an external name containing uppercase letters or lacking an underscore. As a result, you cannot link a Fortran 90 module to some procedures in other languages. The linker reports an unresolved name.

In order to call a C function from a Fortran program, you must 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 supply a function name that Fortran 90 can call. The only solution is to write a C function that takes the same arguments, but that has a name composed of lowercase letters and ending in 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* utility to do it automatically (see "Making C Wrappers With mkf2c").

Correspondence of Fortran and C Data Types

When you exchange data values between Fortran 90 and C, either as parameters, as function results, or as elements of common blocks, you must make sure that the two languages agree on the size, alignment, and subscript of each data value.

Corresponding Scalar Types

The correspondence between Fortran and C scalar data types is shown in Table 3−1 This table assumes the default precisions. Use of compiler options such as *−i2* or *−r8* affects the meaning of the words LOGICAL, INTEGER, and REAL.

Table 3−1 Corresponding Fortran and C Data Types

Fortran Data Type	Corresponding C Type
BYTE, INTEGER(1), LOGICAL(1)	signed char
CHARACTER(1)	unsigned char
INTEGER(2), LOGICAL(2)	short
INTEGER[a], INTEGER(4), LOGICAL[a], LOGICAL(4)	int or long
INTEGER(8), LOGICAL(8)	long long
REAL[a], REAL(4)	float
DOUBLE PRECISION, REAL(8)	double
REAL(16)	long double
COMPLEX[a], COMPLEX(kind=4)	typedef struct{float real, imag; } cpxk4;
DOUBLE COMPLEX, COMPLEX(kind=8)	typedef struct{double real, imag;} cpxk8;
COMPLEX(kind=16)	typedef struct{long double re, im;} cpxk16;
CHARACTER(*n*) (*n*>1)	typedef char fstr_*n*[*n*];

[a] Assuming default kind−parameter

Corresponding Character Types

The Fortran CHARACTER(1) data type corresponds to the C type unsigned char. However, the two languages differ in the treatment of strings of characters.

A Fortran CHARACTER(*n*) (for *n*>1) variable 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).

Since there is no terminal null byte, most of the string library functions familiar to C programmers (*strcpy*(), *strcat*(), *strcmp*(), and so on) cannot be used with Fortran string values. The *strncpy*(), *strncmp*(), *bcopy*(), and *bcmp*() functions can be used because they depend on a count rather than a delimiter.

Corresponding Array Elements

Fortran and C use different arrangements for the elements of an array in memory. Fortran uses column−major order (when iterating sequentially through memory, the leftmost subscript varies fastest), whereas C uses row−major order (the rightmost subscript varies fastest to generate sequential storage locations). In addition, Fortran array indices are by default origin−1, and can be declared as any origin, while C indices are always origin−0.

To use a Fortran array in C,

1. reverse the order of dimension limits when declaring the array
2. reverse the sequence of subscript variables in a subscript expression
3. adjust the subscripts to origin−0 (usually, decrement by 1)

The correspondence between Fortran and C subscript values is depicted in Figure 3−1 You derive the C subscripts for a given element by decrementing the Fortran subscripts and using them in reverse order; for example, Fortran (99,9) corresponds to C [8][98].

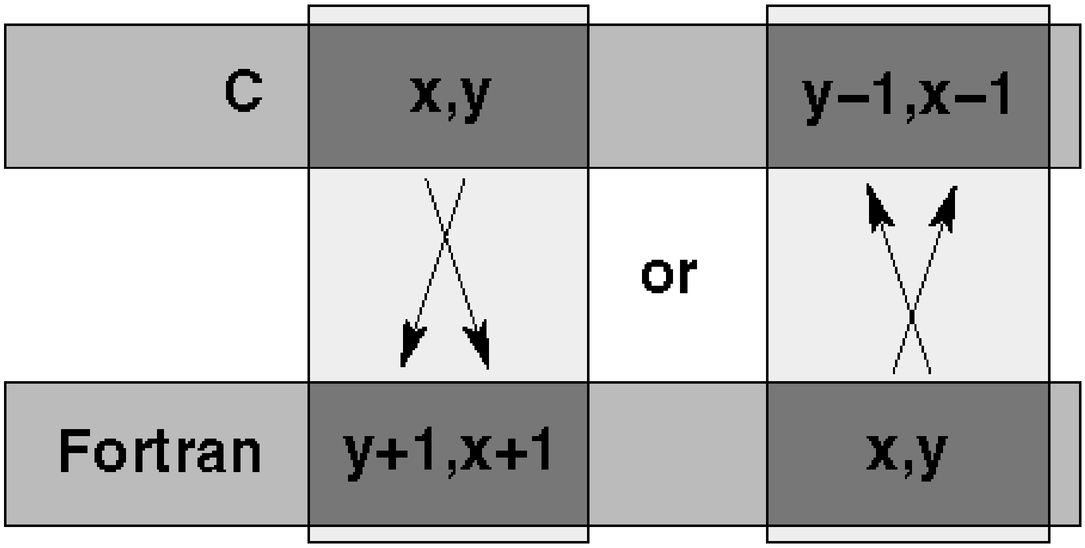

Figure 3−1 Correspondence Between Fortran and C Subscripts

Note: A Fortran array can be declared with some other lower bound than the default of 1. If the Fortran subscript is origin−0, no adjustment is needed. If the Fortran lower bound is greater than 1, the C subscript is adjusted by that amount.

Unsupported Array Parameters

Fortran 90 supports assumed−shape and deferred−shape arrays (see "Assumed−Shape and Deferred−Shape Arrays") as well as array slices. You cannot pass any of these types of array to a non−Fortran procedure. The reason is that Fortran 90 represents such arrays in memory using a descriptor record containing indirect pointers and other data. The format of this record is not part of the published programming interface to MIPSpro Fortran 90, as it is subject to change.

If you attempt to pass an assumed−shape or deferred−shape array, or an array slice, to a non−Fortran function, the function does not receive the address of array elements in memory as it would when an array is passed. Instead it receives the address of a descriptive record of undocumented contents, resulting in unpredictable behavior.

How Fortran Passes Subprogram Parameters

The MIPSpro Fortran 90 compiler generates code to pass parameters according to simple, uniform rules; and it generates subprogram code that expects parameters to be passed according to these rules. When calling non−Fortran functions, you must know how parameters will be passed; and when calling Fortran subprograms from other languages, you must cause the other language to pass parameters correctly.

Note: You should be aware that all compilers for a given version of IRIX use identical conventions for passing parameters. These conventions are documented at the machine instruction level in the *MIPSpro Assembly Language Programmer's Guide*, which also details the differences in the conventions used in different releases.

Normal Treatment of Parameters

Every parameter passed to a subprogram, regardless of its data type, is passed as the address of the actual parameter value in memory. This simple rule is extended for two special cases:

- The length of each CHARACTER(*n*) parameter (for *n*>1) is passed as an additional INTEGER(4) value, following the explicit parameters.
- When a function returns type CHARACTER(*n*) (for *n*>1), the address of the space to receive the result is passed as the first parameter to the function, and the length of the result space is passed as the second parameter, preceding all explicit parameters.

Example 3−1 Example Subroutine Call

```
COMPLEX(8) cp8
CHARACTER(16) creal, cimag
EXTERNAL CPXASC
CALL CPXASC(creal,cimag,cp8)
```

The code generated from the CALL in Example 3−1 prepares the following five argument values:

1. The address of *creal*
2. The address of *cimag*
3. The address of *cp8*
4. The length of *creal*, an integer value of 16
5. The length of *cimag*, an integer value of 16

Example 3−2 Example Function Call

```
CHARACTER(8) symbl,picksym
CHARACTER(100) sentence
INTEGER nsym
symbl = picksym(sentence,nsym)
```

The code generated from the function call in Example 3−2prepares the following five argument values:

1. The address of temporary space to hold the function result (after the function call, the contents of the temporary are copied to variable *symbl*)
2. The length of *symbl*, an integer value of 8
3. The address of *sentence*, the first explicit parameter
4. The addrss of *nsym*, the second explicit parameter
5. The length of *sentence*, an integer value of 100

Calling Fortran From C

There are two types of callable Fortran subprograms: subroutines and functions. In C terminology, both types of subprogram are external functions. The difference is the use of the function return value from each.

Calling Fortran Subroutines From C

From the standpoint of a C module, a Fortran subroutine is an external function returning int. The integer return value is normally ignored by a C caller (it is the alternate return statement number, if any).

Example 3−3shows a simple Fortran 90 subroutine that takes adds arrays of complex numbers.

Example 3−3 Example Fortran Subroutine With COMPLEX Parameters

```
SUBROUTINE ADDC32(Z,A,B,N)
COMPLEX(32) Z(1),A(1),B(1)
INTEGER N,I
DO 10 I = 1,N
    Z(I) = A(I) + B(I)
10 CONTINUE
RETURN
END
```

Example 3−4shows a sketch of how the Fortran 90 subroutine could be called from C.

Example 3−4C Declaration and Call With COMPLEX Parameters

```
typedef struct{long double real, imag;} cpx32;
extern int
  addc32_(cpx32*pz,cpx32*pa,cpx32*pb,int*pn);
cpx32 z[MAXARRAY], a[MAXARRAY], b[MAXARRAY];
```

```
...
    int n = MAXARRAY;
    (void)addc32_(&z, &a, &b, &n);
```

The Fortran subroutine in Example 3–3is named in Example 3–4using lowercase letters and a terminal underscore—the way the Fortran 90 compiler spells the public name in the object file. The subroutine is declared as returning an integer. This return value is ignored but, for clarity, the actual call is cast to void to show that the return value is ignored intentionally, not by accident.

The trivial subroutine in Example 3–5takes adjustable–length character parameters.

Example 3–5 Example Fortran Subroutine With String Parameters

```
SUBROUTINE PRT(BEF,VAL,AFT)
CHARACTER*(*)BEF,AFT
REAL VAL
PRINT *,BEF,VAL,AFT
RETURN
END
```

The C program in Example 3–6prepares CHARACTER*16 values and passes them to the subroutine in Example 3–5

Example 3–6C Program that Passes String Parameters

```
typedef char fstr_16[16];
extern int
   prt_(fstr_16*pbef, float*pval, fstr_16*paft,
                int lbef, int laft);
main()
{
    float val = 2.1828e0;
    fstr_16 bef,aft;
    strncpy(bef,"Before..........",sizeof(bef));
    strncpy(aft,"..........After",sizeof(aft));
    (void)prt_(bef,&val,aft,sizeof(bef),sizeof(aft));
}
```

Observe that the subroutine call requires five parameters: the addresses of the three explicit parameters, and the lengths of the two string parameters. In Example 3–6, the string length parameters are generated using *sizeof()*, which returns the memory size of the typedef *fstr_16*.

When the Fortran code does not require a specific length of string, the C code that calls it can pass an ordinary C character vector, as shown in Example 3–7. In this example, the string length parameter length values are calculated dynamically using *strlen()*.

Example 3–7C Program That Passes Different String Lengths

```
extern int
prt_(char*pbef, float*pval, char*paft, int lbef, int laft);
```

```
main()
{
        float val = 2.1828e0;
        char *bef = "Start:";
        char *aft = ":End";
        (void)prt_(bef,&val,aft,strlen(bef),strlen(aft));
}
```

Calling Fortran Functions From C

A Fortran function returns a scalar value as its explicit result. This corresponds exactly to the C concept of a function with an explicit return value. When the Fortran function returns any type shown inTable 3−1 other than CHARACTER(*n*) (*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.

The trivial function shown in Example 3−8accepts and returns COMPLEX(8) values.

Example 3−8 Fortran Function Returning COMPLEX(8)

```
COMPLEX(kind=8) FUNCTION FSUB8(INP)
COMPLEX(kind=8) INP
FSUB8 = INP
END
```

Although a COMPLEX value is declared as a structure in C, it can be used as the return type of a function. The C program in Example 3−9shows how the function in Example 3−8is declared and called.

Example 3−9C Program That Receives COMPLEX Return Value

```
typedef  struct{ double real, imag; } cpx8;
extern cpx8 fsub8_( cpx8 * inp );
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");
}
```

Observe that the parameters to a function, like the parameters to a subroutine, are passed as pointers, but the value returned is a value, not a pointer to a value.

Note: In IRIX 5.3 and earlier 32−bit systems, you can*not* call a Fortran function that returns COMPLEX (although you can call one that returns any other arithmetic scalar type). The register conventions used by

32−bit compilers prior to IRIX 6.0 do not permit returning a structure value from a Fortran function to a C caller.

Example 3−10 Fortran Function Returning CHARACTER(16)

```
FUNCTION FS16(J,K,S)
   CHARACTER(16) S
   INTEGER J,K
   FS16 = S(J:K)
RETURN
END
```

The function in Example 3−10 has a CHARACTER(16) return value. When a Fortran function returns a CHARACTER*n (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 parameters of the function, preceding the explicit parameters. This is demonstrated in Example 3−11.

Example 3−11 C Program That Receives CHARACTER(16) Return

```
typedef char fstr_16[16];
extern void
fs16_ (fstr_16 *pz,int lz,int *pj,int *pk,fstr_16*ps,int ls);
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 Example 3−11, the address and length of the function result are the first two parameters of the function. (Since type *fstr_16* is an array, its name, *oup*, evaluates to the address of its first element.) The next three parameters are the addresses of the three named parameters. The final parameter is the length of the string parameter.

Calling C From Fortran

In general, 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 parameters (see "How Fortran Passes Subprogram Parameters"). When the C function expects parameters passed using other conventions, you normally need to build a "wrapper" for the C function using the *mkf2c* command.

Normal Calls to C Functions

The C function in Example 3−12is written to use the Fortran conventions for its name (lowercase with final underscore) and for parameter passing.

Example 3−12C Function Written to be Called from Fortran

```
/*
|| C functions to export the facilities of strtoll()
|| to Fortran 77 programs.  Effective Fortran declaration:
||
|| INTEGER*8 FUNCTION ISCAN(S,J)
|| 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);
```

```
      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 program in Example 3−13demonstrates a call to the function in Example 3−12

Example 3−13 Fortran Program to Call a C Function

```
EXTERNAL ISCAN
INTEGER(8) ISCAN
INTEGER(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
```

Using Fortran COMMON 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 "How Fortran 90 Handles External and Public Names" (that is, forced to lowercase and extended with an underscore). The name of the "blank common" is _BLNK__ (one leading underscore and two final ones).

In order to refer to the contents of a common block, take these steps:

- Declare a C structure whose fields have the appropriate data types to match the successive elements of the Fortran common block. (See Table 3−1for corresponding data types.)
- Declare the common block name as an external structure of that type.

A sketch of the method is shown in Example 3−14

Example 3−14 Common Block Usage in Fortran and C

```
      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...
}
```

There are two important restrictions on this capability.

First, you cannot map a common block that contains pointer−based variables. The data object that represents a pointer−based variable is not documented, so you cannot know what kind of C data type to use at that point in the C structure declaration.

Second, if the common block contains a variable of Fortran 90 derived type (a structure), you must be sure that the derived type is declared with the SEQUENCE attribute. Otherwise, you cannot be sure that its fields will appear in the expected sequence in memory.

Using Fortran Arrays in C Code

As described under "Corresponding Array Elements", a C program must take special steps to access arrays created in Fortran. The Fortran fragment in Example 3−15prepares a matrix in a common block, then calls a C subroutine to modify the array.

Example 3−15 Fortran Program Sharing an Array in Common with C

```
INTEGER IMAT(10,100),R,C
COMMON /WITHC/IMAT
R = 74
C = 6
CALL CSUB(C,R,746)
PRINT *,IMAT(6,74)
END
```

The C function in Example 3−16stores 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.

Example 3−16 C Subroutine to Modify a Common Array

```
extern struct { int imat[100][10]; } withc_;
int csub_(int *pc, int *pr, int *pval)
{
    withc_.imat[*pr-1][*pc-1] = *pval;
    return 0; /* all Fortran subrtns return int */
}
```

Calls to C Using %LOC and %VAL

Using the special intrinsic functions %VAL and %LOC you can pass parameters in ways other than the standard Fortran conventions described under ‘"How Fortran Passes Subprogram Parameters".

Using %VAL

%VAL is used in parameter lists to cause parameters to be passed by value rather than by reference. Suppose that you need to call a C function having the following prototype:

```
int takesint_ (int p1, char *p2, int len)
```

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 Fortran 90 code similar to that in Example 3−17.

Example 3−17 Fortran Call Using %VAL

```
character(80) sentence
integer(4) j
call takesint(%VAL(j),sentence)
```

The use of %VAL(*j*) causes the contents of *j* to be passed, rather than the address of *j*.

Using %LOC

%LOC returns the address of its argument. It can be used with %VAL to prevent passing the lengths of character values as hidden parameters. In other words, the argument %LOC(%VAL(*char_var*)) passes only the address of *char_var*; it does not pass the implicit length argument.

Making C Wrappers With mkf2c

The program *mkf2c* provides an alternate interface for C routines called by Fortran. (Some details of *mkf2c* are covered in the *mkf2c*(1) reference page.)

The *mkf2c* program reads a file of C function prototype declarations and generates an assembly language module. This module contains one callable entry point for each C function. The entry point, or "wrapper," accepts parameters in the Fortran calling convention, and passes the same values to the C function using the C conventions.

A simple case of using a function as input to *mkf2c* is

```
simplefunc (int a, double df)
{ /* function body ignored */ }
```

For this function, *mkf2c* (with no options) generates a wrapper function named *simple_* (truncated to 6 characters, made lowercase, with an underscore appended). The wrapper function expects two parameters, an integer and a REAL*8, passed according to Fortran conventions; that is, by reference. The code of the wrapper loads the values of the parameters into registers using C conventions for passing parameters by value, and calls *simplefunc()*.

Parameter Assumptions by mkf2c

Since *mkf2c* processes only the C source, not the Fortran source, it treats the Fortran parameters based on the data types specified in the C function header. These treatments are summarized in Table 3–2

Table 3–2 How *mkf2c* Treats Function Arguments

Data Type in C Prototype	Treatment by Generated Wrapper Code
unsigned char	Load CHARACTER(1) from memory to register, no sign extension.
char	Load CHARACTER(1) from memory to register; sign extension only when *–signed* is specified.
unsigned short, unsigned int	Load INTEGER(2) or INTEGER(4) from memory to register, no sign extension.
short	Load INTEGER(2) from memory to register with sign extension.
int, long	Load INTEGER(4) from memory to register with sign extension.
long long	Load INTEGER(8) from memory to register with sign extension.
float	Load REAL(4) from memory to register, extending to double unless *–f* is specified.
double	Load REAL(8) from memory to register.
long double	Load REAL(16) from memory to register.
char *name*[], *name*[*n*]	Pass address of CHARACTER(*n*) and pass length as integer parameter as Fortran does.
char *	Copy CHARACTER(*n*) value into allocated space, append null byte, pass address of copy.

Character String Treatment by mkf2c

In Table 3–2, notice the different treatments for an argument declared as a character array and one declared as a character address (even though these two declarations are semantically the same in C).

When the C function expects a character address, *mkf2c* generates the code to dynamically allocate memory and to copy the Fortran character value, for its specified length, to memory. This creates a null–terminated string. In this case,

- the address passed to C points to allocated memory

- the length of the value is not passed as an implicit argument
- there is a terminating null byte in the value
- changes in the string are *not* reflected back to Fortran

A character array is passed by *mkf2c* as a Fortran CHARACTER**n* value. In this case,

- the address prepared by Fortran is passed to the C function
- the length of the value is passed as an implicit argument (see "Normal Treatment of Parameters")
- the character array contains no terminating null byte
- changes in the array by the C function *are* visible to Fortran

Since the C function cannot declare the extra string−length parameter (if it declared the parameter, *mkf2c* would process it as an explicit argument), the C programmer has a choice of ways to access the string length. When the Fortran program always passes character values of the same size, the length parameter can simply be ignored. If its value is needed, the *varargs* macro can be used to retrieve it.

Suppose the C function prototype is specified as follows:

```
void func1 (char carr1[],int i, char *str, char carr2[]);
```

In this case, *mkf2c* passes a total of six parameters to C. The fifth parameter is the length of the Fortran value corresponding to *carr1*. The sixth is the length of *carr2*. The C function can use the *varargs* macros to retrieve these hidden parameters. *mkf2c* ignores the *varargs* macro *va_alist* appearing at the end of the parameter name list.

When *func1* is changed to use varargs, the C source file is as shown in Example 3−18

Example 3−18 C Function Using *varargs*

```
#include "varargs.h"
void
func1 (char carr1[],int i,char *str,char carr2[],va_alist);
{}
```

The C routine would retrieve the lengths of *carr1* and *carr2*, placing them in the local variables *carr1_len* and *carr2_len*, using code like the fragment shown in Example 3−19.

Example 3−19 C Code to Retrieve Hidden Parameters

```
va_list ap;
int carr1_len, carr2_len;
va_start(ap);
carr1_len = va_arg (ap, int)
carr2_len = va_arg (ap, int)
```

Restrictions of mkf2c

When it does not recognize the data type specified in the C function, *mkf2c* issues a warning message and generates code to simply pass the pointer set up by Fortran. It does this in the following cases:

- any nonstandard data type name, for example a data type that might be declared using typedef or a data type defined as a macro
- any structure argument
- any argument with multiple indirection (two or more asterisks, for example *char***)

Since *mkf2c* does not support structure−valued arguments, it does not support passing COMPLEX**n* values or derived types. Nor does *mkf2c* have any means of passing assumed−shape or deferred−shape arrays.

Using mkf2c and extcentry

mkf2c understands only a limited subset of the C grammar. This subset includes common C syntax for function entry point, C−style comments, and function bodies. However, it does not include constructs such as typedefs, external function declarations, or C preprocessor directives. The presence of these things in the input to *mkf2c* can confuse it.

To ensure that only the constructs understood by *mkf2c* are included in wrapper input, you need to place special comments around each function for which Fortran−to−C wrappers are to be generated (see example below).

Once these special comments, **/* CENTRY */** and **/* ENDCENTRY */**, are placed around the code, use the program *excentry*(1) before *mkf2c* to generate the input file for *mkf2c*.

Example 3−20illustrates the use of *extcentry*. It shows the C file *foo.c* containing the function **foo**, which is to be made Fortran callable.

Example 3−20 Source File for Use With *extcentry*

```
typedef unsigned short grunt [4];
struct {
   long 1,11;
   char *str;
} bar;
main ()
{
   int kappa =7;
   foo (kappa,bar.str);
}
/* CENTRY */
foo (integer, cstring)
int integer;
char *cstring;
{
   if (integer==1) printf("%s",cstring);
} /* ENDCENTRY */
```

The special comments **/* CENTRY */** and **/* ENDCENTRY */** surround the section that is to be made Fortran callable. To generate the assembly language wrapper *foowrp.s* from the above file *foo.c*, use the following set of commands:

```
% extcentry foo.c foowrp.fc
% mkf2c foowrp.fc foowrp.s
```

The programs *mkf2c* and *extcentry* are found in the directory */usr/bin*.

Makefile Considerations

make(1) contains default rules to help automate the control of wrapper generation. The following example of a makefile illustrates the use of these rules. In the example below, an executable object file is created from the files *main.f* (a Fortran main program) and *callc.c*:

```
test:  main.o callc.o
   f90 -o test main.o callc.o
callc.o: callc.fc
clean:
   rm -f *.o test *.fc
```

In this program, main calls a C routine in *callc.c*. The extension *.fc* has been adopted for Fortran−to−call−C wrapper source files. The wrappers created from *callc.fc* will be assembled and combined with the binary created from *callc.c*. Also, the dependency of *callc.o* on *callc.fc* will cause *callc.fc* to be recreated from *callc.c* whenever the C source file changes. (The programmer is responsible for placing the special comments for *extcentry* in the C source as required.)

Note: Options to *mkf2c* can be specified when *make* is invoked by setting the *make* variable F2CFLAGS. Also, do not create a *.fc* file for the modules that need to have wrappers created. These files are both created and removed by *make* in response to the *file.o:file.fc* dependency.

The *makefile* above controls the generation of wrappers and Fortran objects. You can add modules to the executable object file in one of the following ways:

- If the file is a native C file whose routines are not to be called from Fortran using a wrapper interface, or if it is a native Fortran file, add the *.o* specification of the final make target and dependencies.
- If the file is a C file containing routines to be called from Fortran using a wrapper interface, the comments for *extcentry* must be placed in the C source, and the *.o* file placed in the target list. In addition, the dependency of the *.o* file on the *.fc* file must be placed in the makefile. This dependency is illustrated in the example makefile above, where *callf.o* depends on *callf.fc*.

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 of the CPU, and in particular you can change the rounding mode. When running Fortran 90 programs that contain quad−precision operations, you must run the compiler in round−to−nearest mode. Because this mode is the default, you usually do not need to be concerned with setting it. You usually need to set this mode when writing programs that call your own assembly routines. Refer to the *swapRM* manual page for further details.

Chapter 4

Controlling Scalar Optimization

All MIPSpro compilers share a common optimizer phase, controlled by the use of the *−On* driver option. MIPSpro Fortran 90 also has source−level scalar optimization, controlled with source assertions and the *−WK* driver option.

Use of the common optimizer with any language is documented in the *MIPS Compiling and Performance Tuning Guide*. This chapter covers optimizer features that are unique to Fortran 77 and Fortran 90, and the use of source−level scalar optimizations, that is, optimizations that are equally effective on uniprocessor and multiprocessor machines.

Advanced source−level optimizations are described in Chapter 5 and Chapter 6. The parallel−optimization features unique to Power Fortran 90 are introduced in Chapter 7.

Overview of Scalar Optimization

You can use the compiler to perform various scalar optimizations by specifying any of the options listed in Table 4−1 on the command line. Specify the options in a comma−separated list with no intervening blanks, following the *-WK* option, as follows:

`% f90` *f77options* `-WK,`*option*`[,`*option*`] ...` *file*

The *−WK* option passes its arguments to the optimization phases that are invoked by the Fortran front end.

Defaults are set for the options in Table 4−1 by the *−On* option. These defaults are reflected in the pfa(1) and fopt(1) reference pages, and are usually correct. You use specific values of these options only when dealing with special performance−tuning issues unique to your application.

You can manage many of the optimizations listed in Table 4−1 using compiler directives (see Chapter 6, "Using Directives and Assertions").

Table 4−1 Optimization Options

Long Name	Short Name	Default Value
-aggressive=*letter*	-ag=*letter*	option off
-arclimit=*integer*	-arclm=*integer*	5000
-[no]assume=*list*	-[n]as=*list*	CEL
-cacheline=*integer*	-chl=*integer*	4
-cachesize=*integer*	-chs=*integer*	256
-[no]directives=*list*	-[n]dr=*list*	ackpv
-dpregisters=*integer*	-dpr=*integer*	16
-each_invariant_if_growth=*integer*	-eiifg=*integer*	20
-fpregisters=*integer*	-fpr=*integer*	16
-fuse	-fuse	option on with -scalaropt=2 or -optimize=5
-max_invariant_if_growth=*integer*	-miifg=*integer*	500
-optimize=*integer*	-o=*integer*	depends on -O option
-recursion	-rc	option on

-roundoff=*integer*	-r=*integer*	depends on -O option
-scalaropt=*integer*	-so=*integer*	depends on -O option
-setassociativity=*integer*	-sasc=*integer*	1
-unroll=*integer*	-ur=*integer*	4
-unroll2=*weight*	-ur2=*weight*	100

Controlling the Optimization Level

Controlling Scalar Optimizations

The *-scalaropt=n* option (or *-so=n*) controls the level of scalar optimizations that the compiler performs. Valid values for *n* are:

0	Disables all scalar optimizations.
1	Enables simple scalar optimizations—dead code elimination, global forward substitution of variables, and conversion of IF−GOTO to IF−THEN−ELSE.
2	Enables the full range of scalar optimizations—floating of invariant IF statements out of loops, array expansion, loop fusion, loop peeling, induction variable recognition, and loop rerolling and unrolling (if *-roundoff* is greater than zero; see "Controlling Variations in Round Off"
3	Enables memory management transformations if *-roundoff=3* (see "Performing Memory Management Transformations" on page 68). Performs dead−code elimination during output conversion.

There is no default value for *−scalaropt* If you do not specify it, this option can still be in effect through the *-O* option.

Optimization level can also be influenced with a directive in the source file; see "Setting Optimization Level".

Controlling General Optimizations

The *-optimize=n* option (or *-o=n*) sets the optimization level. Each optimization level is cumulative (that is, level 5 performs everything up to and including level 5). You can also modify the optimization level on a loop−by−loop basis by using a directive within the source file (see "Setting Optimization Level"). Valid values for *n* are:

0	Disables optimization.
1	Performs only simple optimizations. Enables induction variable recognition.
2	Performs lifetime analysis to determine when last−value assignment of scalars is necessary.
3	Recognizes triangular loops and attempts loop interchanging to improve memory referencing. Uses special case data dependence tests. Also, recognizes special index sets called wraparound variables.

4	Generates two versions of a loop, if necessary, to break a data dependence arc.
5	Enables array expansion and loop fusion.

Although higher optimization levels increase performance, they also increase compilation time. There is no default value for this option. If you do not specify it, this option can still be in effect through the *-O* option.

The output of following example is described for *-optimize=1*, *-optimize=2*, and *-optimize=5* to illustrate the range of this option. (This example also uses *−pfa* and *-minconcurrent=0.*)

```
    ASUM = 0.0
    DO 10 I = 1,M
       DO 10 J = 1,N
         ASUM = ASUM + A(I,J)
         C(I,J) = A(I,J) + 2.0
10  CONTINUE
```

At *-optimize=1*, the compiler sees the summation in ASUM as an intractable data dependence between iterations and does not try to optimize the loop. At *-optimize=2* (perform lifetime analysis and do not interchange around reduction) the compiler is able to recognize the reduction and automatically introduce a reduction parallelization:

```
      ASUM = 0.
C$DOACROSS SHARE(M,N,A,C),LOCAL(I,J),REDUCTION(ASUM)
      DO 3 I=1,M
        DO 2 J=1,N
          ASUM = ASUM + A(I,J)
          C(I,J) = 2. + A(I,J)
      2 CONTINUE
      3 CONTINUE
```

Reduction and other parallelizations are discussed in Chapter 7; see especially "Using the REDUCTION Clause" and "Dealing With Reduction".

Specifying *-optimize=5* (loop interchange around reduction to improve memory referencing) produces the following:

```
      ASUM = 0.
C$DOACROSS SHARE(N,M,A,C),LOCAL(J,I),REDUCTION(ASUM)
     DO 3 J=1,N
     DO 2 I=1,M
          ASUM = ASUM + A(I,J)
          C(I,J) = 2. + A(I,J)
    2 CONTINUE
    3 CONTINUE
```

Controlling Global Assumptions

The *-assume=list* option (or *-as=list*) controls certain global assumptions of a program. You can also control most of these assumptions with various assertions (see "Using Assertions"). The default is *-assume=cel*.

The *list* can contain the following letters:

a	Assumes procedure argument aliasing is used, meaning that different dummy arguments could refer to the same object. This practice is forbidden by the Fortran 77 and Fortran 90 standards. This option provides a method of dealing with programs that use argument aliasing anyway.
b	Assumes array subscripts may go outside the declared array bounds.
c	Places constants used in subroutine or function calls in temporary variables.
e	Assumes that two or more variable names defined in EQUIVALENCE statements may be used to refer to the same memory location inside one DO loop nest.
l	Uses temporary variables within an optimized loop and assigns the last value to the original scalar, if the compiler determines that the scalar can be reused before it is assigned.

By default, the compiler assumes that a program conforms to the Fortran 90 standard, that is,*-assume=el*, and includes *-asssume=c* to simplify some analysis and inlining. When your program conforms strictly to language standards, you can specify the *-noassume* option.

The following command compiles the Fortran program *source.f90*, assuming it allows subscripts out of bounds:

```
% f90 -WK,-assume=b source.f
```

Specifying Recursion

The *-recursion* (or *-rc*) option tells the compiler that subroutines and functions in the source program can be called recursively (that is, a subroutine or function calls itself, or it calls another routine that calls it). The presence of recursion affects storage allocation decisions.

This option is enabled by default. To disable it, specify *-norecursion* (or *-nrc*). You can control this assumption on a procedure basis by using a directive within the source file (see "Ignoring Data Dependence Conflicts").

Unsafe transformations can occur unless the *-recursion* option is enabled for each recursive routine that the compiler processes.

Setting the Listing Level

If you request one, the optimization phases will produce a listing showing how your source text was modified. This enables you to check line by line to see which loops were unrolled, how temporary variables were inserted, and so on. The basic option for receiving a listing is *−listoptions=letters* (or *−lo=letters*) The letters that can be given include:

C	display calling tree
N	display program unit names as processed

O	annotated listing of original program
S	summary statistics
T	annotated listing of output program

For example, the command

```
f90 ...options... -O5 -WK,-lo=CT testmodule.f90
```

produces a file named *testmodule.L* containing a listing of the calling tree as deduced by the optimizer, and an annotated listing of the modified program.

Enabling Loop Fusion

The *-fuse* option enables loop fusion, an optimization that transforms two adjacent loops into a single loop. The use of data−dependence tests allows fusion of more loops than is possible with standard techniques. You must also specify *-scalaropt=2* or *-optimize=5* to enable loop fusion.

Setting Invariant IF Floating Limits

When a loop contains an IF statement whose condition cannot change from one iteration to another (a loop−invariant if), the compiler performs the same test for every iteration. The code can often be made more efficient by floating the IF statement out of the loop and putting the THEN and ELSE sections into their own loops. This process is called invariant IF floating.

The *-each_invariant_if_growth* and the *-max_invariant_if_growth* options control limits on invariant IF floating. This process generally involves duplicating the body of the loop, which can increase the amount of code considerably.

The *-each_invariant_if_growth=n* option (or *-eiifg=n*) controls the rewriting of IF statements nested within loops. This option specifies a limit on the number of executable statements in a nested IF statement. If the number of statements in the loop exceeds this limit, the compiler does not rewrite the code. If there are fewer statements, the compiler improves execution speed by interchanging the loop and IF statements. This process becomes complicated when there is other code in the loop, since a copy of the other code must be included in both the THEN and ELSE loops.

Valid values for *n* are from 0 to 100; the default is 20. The code in Example 4−1illustrates a loop−invariant IF.

Example 4−1 Skeleton of a Loop Containing an Invariant IF

```
DO I = ...
    section-1
    IF ( ) THEN
      section-2
    ELSE
      section-3
    ENDIF
    section-4
```

```
ENDDO
```

Under Invariant−IF Floating,Example 4−1is modified to Example 4−2

Example 4−2 Skeleton of Code With Invariant IF Floated Out

```
IF ( ) THEN
    DO I = ...
      section−1
      section−2
      section−4
    ENDDO
ELSE
    DO I = ...
      section−1
      section−3
      section−4
    ENDDO
ENDIF
```

When sections 1 and 4 are large, the extra code generated can cost a program more time, through cache contention, extra paging, and so on, than it gains from the reduced number of Boolean expressions. The *-each_invariant_if_growth* option provides a maximum size (in number of lines of executable code) of the duplicate sections, above which the compiler will try to float an invariant IF statement outside a loop.

You can limit the total amount of additional code generated in a program unit through invariant IF floating by specifying the *-max_invariant_if_growth* option. This option (or *-miifg=n*) specifies an upper bound on the total number of additional lines of code the compiler can generate in each program unit through invariant−IF floating. This limit is applied on a per−subroutine basis. For example, if a subroutine is 400 lines long and *-miifg=500*, the compiler can add at most 100 lines in the process of invariant IF floating. The default for *n* is 500.

Note: Other compiler optimizations can add or delete lines, so the final number of lines might differ from the value specified with *-miifg*.

Both limits can be controlled on a loop−by−loop basis with a directive within the source (see "Setting Invariant IF Floating Limits").

Controlling Variations in Round Off

The *-roundoff=n* option (or *-r=n*) controls the amount of variation in roundoff error produced by optimization. If an arithmetic reduction is accumulated in a different order than is used in the scalar program, the roundoff error is accumulated differently and the final result might differ from the output of the original program. Although the difference is usually insignificant, certain restructuring transformations performed by the compiler must be disabled to obtain exactly the same answers as the scalar program.

The values you can specify for *n* are cumulative. For example, *-roundoff=3* performs what is described for level 3 in addition to what is listed for the previous levels. Valid values for *n* are:

0 Suppresses any transformations that change roundoff error.

1 Performs expression simplification (which might generate various overflow or underflow errors) for expressions with operands between binary and unary operators, expressions that are inside trigonometric intrinsic functions returning integer values, and after forward substitution. Enables strength reduction. Performs intrinsic function simplification for *max* and *min*. Enables code floating if *-scalaropt* is at least 1. Allows loop interchanging around serial arithmetic reductions, if *-optimize* is at least 4. Allows loop rerolling, if *-scalaropt* is at least 2.

2 Allows loop interchanging around arithmetic reductions if *-optimize* is at least 4. For example, the floating point expression A/B/C is computed as A/(B*C).

3 Recognizes REAL (float) induction variables if *-scalaropt* greater than 2 or *-optimize* is at least 1. Enables sum reductions. Enables memory management optimizations if *-scalaropt=3* (see "Performing Memory Management Transformations" for details about memory management transformations).

Consider the code fragment in Example 4–3

Example 4–3 Code Fragment With Summation

```
      ASUM = 0.0
      DO 10 I = 1,M
        DO 10 J = 1,N
            ASUM = ASUM + A(I,J)
            C(I,J) = A(I,J) + 2.0
 10   CONTINUE
```

When *-roundoff=1*, the compiler does not transform the summation reduction. The compiler distributes the loop, as shown in Example 4–4

Example 4–4 Code Fragment Distributed

```
     ASUM = 0.
       DO 2 J=1,N
     DO 2 I=1,M
         ASUM = ASUM + A(I,J)
 2       CONTINUE
         DO 3 J=1,N
         DO 3 I=1,M
         C(I,J) = A(I,J) + 2.
   3     CONTINUE
```

When *-roundoff=2* and *-optimize=5* (reduction variable identification and loop interchange around arithmetic reduction), Example 4–3 is transformed as shown in Example 4–5

Example 4–5 Code Fragment Transformed with –roundoff=2

```
      ASUM = 0.
         DO 10 J=1,N
       DO 2 I=1,M
           ASUM = ASUM + A(I,J)
           C(I,J) = A(I,J) + 2.
     2 CONTINUE
  10  CONTINUE
```

When *-roundoff=3* and *-optimize=5*, the compiler recognizes REAL induction variables. Example 4−6 shows such a loop.

Example 4−6 Code Fragment With REAL Induction Variable

```
      ASUM = 0.0
      X = 0.0
      DO 10 I = 1,N
         ASUM = ASUM + A(I)*COS(X)
         X  =  X + 0.01
10    CONTINUE
```

When *-roundoff=3* and *-optimize=5*, this loop is transformed in the way shown in Example 4−7.

Example 4−7 Transformed Loop With REAL Induction Variable

```
      ASUM = 0.
      X = 0.
      DO 10 I=1,N
        ASUM = ASUM + A(I) * COS ((I − 1) * 0.01)
10    CONTINUE
```

Using Vector Intrinsic Functions

The nine intrinsic functions ASIN, ACOS, ATAN, COS, EXP, LOG, SIN, TAN, and SQRT have a scalar (element by element) version and a special version optimized for vectors. When you use −O3 optimization, the compiler uses the vector versions if it can. On the MIPS R8000 and R10000 processors, the vector function is significantly faster than the scalar version, but has a few restrictions on its use.

Finding Vector Intrinsics

To apply the vector intrinsics, the compiler searches for loops of the following form:

```
real a(10000), b(10000)
do j = 1, 1000
   b(2*j) = sin(a(3*j))
enddo
```

The compiler can recognize the eight functions ASIN, ACOS, ATAN, COS, EXP, LOG, SIN, and TAN when they are applied between elements of named variables in a loop (SQRT is not recognized automatically). The compiler automatically replaces the loop with a single call to a special, vectorized version of the function.

The compiler cannot use the vector intrinsic when the input is based on a temporary result or when the output replaces the input. In the following example, only certain functions can be vectorized:

```
real a(400,400), b(400,400), c(400,400), d( 400,400 )
   call xx(a,b,c,d)
   do j = 100,300,2
      do i = 100, 300,3
         a(i,j) = 1.23*i + a(i,j)
         b(i,j) = sin(a(i,j) + 1.0)
         a(i,j) = log(a(i,j))
         c(i,j) = sin(c(i,j)) / cos(d(i,j))
         d(i+30,j-10) = tan( d(j,i) )
      enddo
   enddo
   call xx(a,b,c,d)
   end
```

In the preceding function,

- the first SIN call is applied to a temporary value and cannot be vectorized
- the LOG call can be vectorized
- results from the second SIN call and first COS call are used in temporary expressions and cannot be vectorized
- the TAN call can be vectorized

Limitations of the Vector Intrinsics

The vector intrinsics are limited in the following ways:

- The SQRT function is not used automatically in the current release.
- The single−precision COS, SIN, and TAN functions are valid only for arguments whose absolute value is less than or equal to 2**28.
- The double−precision COS, SIN, and TAN functions are valid only for arguments whose absolute value is less than or equal to PI*2^{19}.

The vector functions assume that the input and output arrays either coincide completely, or do not overlap. They do not check for partial overlap, and will produce unpredictable results if it occurs.

Disabling Vector Intrinsics

If you need to disable use of vector intrinsics while still compiling at *−O3* level, you can do so. Specify the option *−OPT:vector_intrinsics=OFF*

```
f90 -64 -mips4 -O3 -OPT:vector_intrinsics=OFF trig.f
```

Calling Vector Intrinsic Functions Directly

The *MIPSpro Fortran 77 Programmer's Guide* gives a method of calling the vector intrinsic functions directly. This method cannot be used from Fortran 90 (because the C functions that implement vector intrinsics do not have names ending in an underscore). However, you can write a "wrapper" in C or in Fortran 77 that calls the vector functions, and call this from Fortran 90.

Using Aggressive Optimization

The *-aggressive=letter* option (or *-ag=letter*) performs optimizations that are normally forbidden. In order to use this option, your program must be a single file so that the compiler can analyze all of it simultaneously.

The only available value for *letter* is *a*, which instructs the compiler to add padding to Fortran COMMON blocks. This optimization provides favorable alignments of the virtual addresses. This option does not have a default value.

```
% f90 -WK,-ag=a program.f
```

Unfortunately, it is not always possible to add padding to a COMMON. Fortran allows different routines to have different definitions of COMMON. Therefore, when using this option the entire program must be in a single source file, so the compiler can check for equivalent COMMON definitions.

Controlling Internal Table Size

The *-arclimit=integer* option (or *-arclm=integer*) sets the size of the internal table that the compiler uses to store data dependence information. The default value for *integer* is 5000.

The compiler dynamically allocates the dependence data structure on a loop–nest–by–loop–nest basis. If a loop contains too many dependence relationships and cannot be represented in the dependence data structure, the compiler will stop analyzing the loop. Increasing the value of *-arclimit* allows the compiler to analyze larger loops.

Note: Most users do *not* need to change this value. Also, the number of data dependencies (and the time required to do the analysis) is potentially nonlinear in the length of the loop. Very long loops (several hundred lines) may be impossible to analyze regardless of the value of *-arclimit*.

Performing Memory Management Transformations

Memory management transformations are based on information about the characteristics of the hardware memory and cache.

The compiler recognizes the following memory management command line options when specified with the *–WK* option:

–cacheline	Specifies the width of the memory channel between cache and main memory.

−cachesize	Specifies the data cache size.
−fpregisters	Specifies number of available floating−point registers.
−dpregisters	Specifies number of available double precision registers.
−setassociativity	Specifies which memory management transformation to use.

The *-cacheline=n* option (or *-chl=n*) specifies the width of the memory channel, in bytes, between the cache and main memory. The default value for *n* is 4. The correct value for the Power Challenge and Power Indigo2 systems is 128.

The *-cachesize=n* option (or *-chs=n*) specifies the size of the data cache, in kilobytes, for which to optimize. The default value for *n* is 256 kilobytes. You can obtain the cache size for a given machine with the *hinv*(1) command.

The *-setassociativity=n* option (or *-sasc=n*) provides information on the mapping of physical addresses in main memory to cache pages. The default value for *integer*, 1, says that a datum in main memory can be put in only one place in the cache. If this cache page is already in use, its contents must be rewritten or flushed so that the newly−accessed page can be copied into the cache. The recommended value is 1 for all machines except the POWER CHALLENGE and POWER Onyx series, where you should set it to 4.

The *-dpregisters=n* option (or *-dpr=n*) specifies the number of double−precision registers each CPU has. The *-fpregisters=n* option (or *-fpr=n*) specifies the number of single precision (that is, ordinary floating point) registers each CPU has.

You should specify the same value for both *-dpregisters* and *-fpregisters*. The default values for *n* are 16 for both options. When compiled in 32−bit mode, Silicon Graphics recommends that you do not specify 16, although that is what the hardware supports. It is better to specify a smaller value such as 12, to provide extra registers in case the compiler needs them. In 64−bit mode, where the hardware supports 32 registers, specify 28.

Chapter 5

Inlining and Interprocedural Analysis

This chapter contains the following sections:

- "Overview Inlining and IPA" defines inlining and interprocedural analysis (IPA) and summarizes the driver options that control these processes.
- "Specifying Functions for Inlining or IPA" explains how to specify which function calls will be inlined or analyzed.
- "Specifying Occurrences for Inlining and IPA" explains how to manage the depth and time cost of inlining and IPA.
- "Conditions That Prevent Inlining and IPA" lists several conditions that prevent inlining and interprocedural analysis.

Overview Inlining and IPA

Inlining is the process of replacing a function reference with a copy of the code of the function. This eliminates the overhead of the function call, and can assist other optimizations by making more evident the relationships between the function arguments and returned value, and the surrounding code. However, it also expands the size of the generated object code.

Interprocedural analysis (IPA) is the process of inspecting called functions to get information on relationships between arguments, returned values, and global data. IPA can provide many of the benefits of inlining without replacing the function reference.

You can control inlining and IPA from the command line and also by using directives in your source code. The driver options for inlining and IPA are summarized in Table 5−1 As with all special optimizations, you specify them as sub−options of the-*WK* option.

Table 5−1 Inlining and IPA Options

Long Option Name	Short Option Name	Default Value
-inline[=*list*]	-inl[=*list*]	option off
-ipa[=*list*]	-ipa[=*list*]	option off
-inline_and _copy	-inlc	option off
-inline_looplevel=*integer*	-inll=*integer*	2
-ipa_looplevel=*integer*	-ipall=*integer*	2
-inline_depth=*integer*	-ind=*integer*	2
-inline_man	-inm	option off
-ipa_man	-ipam	option off
-inline_from_files=*list*	-inff=*list*	option off
-ipa_from_files=*list*	-ipaff=*list*	option off
-inline_from_libraries=*list*	-infl=*list*	option off
-ipa_from_libraries=*list*	-ipa=*list*	option off
-inline_create=*name*	-incr=[=*name*]	option off

-ipa_create=*name*]	-ipacr=[=*name*]	option off

Specifying Functions for Inlining or IPA

To request inlining of all eligible function calls, specify *–inline* To request analysis of all eligible function calls, specify *–ipa* However, full inlining or full IPA can be time–consuming and may not yield good results.

Often there are specific functions that are particularly good candidates for inlining or IPA, either because of their contents or because of their frequency of use. You can use the *–inline*and *–ipa*options to specify these functions. When you do, calls to other functions are not analyzed.

The *-inline=list* option (or *-inl=list*) specifies a list of functions that should be expanded inline. The *-ipa=list* option specifies a list of routines that should be analyzed. The names in *list* are separated by colons.

The following command performs inline expansion on the two routines *saxpy* and *daxpy* from the file *foo.f* :

```
f90 –WK,–inline=saxpy:daxpy foo.f
```

The compiler looks for the routines in the current source file, unless you specify an *-inline_from* or *-ipa_from* option. Refer to "Specifying Where to Search for Routines" for details.

Specifying Where to Search for Routines

In order to copy or to analyze a function, the compiler must have access to the text of the function body. If you do not specify otherwise, the compiler searches for the function in the current source file.

The options listed in Table 5–2tell the compiler where to search for the routines specified with the *-inline* or *-ipa* options. If you do not specify either option, the compiler searches the current source file by default.

Table 5–2 Inlining and IPA Search Command Line Options

Long Option Name	Short Option Name
-inline_from_files =*list*	- **inff=***list*
-ipa_from_files=*list*	- **ipaff=***list*
-inline_from_libraries=*list*	-infl=*list*
-ipa_from_libraries=*list*	-ipafl=*list*

In each case, *list* consists of names of files or directories separated by colons. When you specify a directory, the compiler uses all appropriate files in that directory. For example

```
f90 ... –WK,–inline_from_files=subs.f90:../common
```

Note: These options by themselves do not initiate inlining or IPA. They only specify where to look for the routines. Use them in conjunction with the appropriate *-inline* or *-ipa* option.

If you specify a nonexistent file or directory, the compiler issues an error. If you specify multiple *-inline_from* or *-ipa_from* options, the compiler concatenates their lists. All lists are searched in the order that they appear on the command line.

The compiler recognizes two special abbreviations when specified in *list*:

- "–" means current source file (as listed on the command line or specified in an *-input=file* command line option)
- "." means the current working directory

The following command specifies inline expansion on the current source file, *calc.f90*, followed by *subs.f90*.

```
% f90 -WK,-inline,-inline_from_files=-:subs.f90 calc.f90
```

When executed, the compiler searches the current source file *calc.f* and *input.f* for all eligible routines to expand. (It searches for all eligible routines because the *-inline* option was specified without a *list*.)

The compiler resolves routine name references by a searching for them in the order that they appear in *-inline_from*/*-ipa_from* options on the command line. Libraries are searched in their original lexical order.

The compiler recognizes the type of file from its extension, or lack of one, as described in Table 5–3 (The creation and use of libraries is the subject of the next topic.)

Table 5–3 Filename Extensions

Extension	Type of File
.f, *.F*, *.for*, *.FOR*	Fixed–format source (Fortran 77 or Fortran 90)
.f90, *.F90*	Free–format source
.i	Fortran source run through cpp
.klib	Library created with -inline_create or -ipa_create option
Other	Directory

Creating Libraries of Inline Functions

Normally, inlining or IPA is done directly from a Fortran source file. However, when the same set of functions is called from many different programs, it is more efficient to create a pre–analyzed library of the routines.

Use the *-inline_create=name* option (or *-incr=name*) to create a library of prepared function texts for later use. The *-ipa_create=name* option (or *-ipacr=name*) is the analogous option for IPA. The created library contains preprocessed information about the functions in a source file. The compiler can use this quickly for inlining or for IPA.

Note: Libraries created for inlining contain complete information and can be used for both inlining and IPA. Libraries created for IPA contain only summary information and can be used only for IPA.

The compiler assigns *name* to the library file it creates. Since the compiler recognizes input libraries by their file suffix, you should specify the file suffix *.klib*, for example: *prog.klib*.

Creating a library is done separately from compiling. Create a library using the *-inline_create* option (or the *-ipa_create* option for IPA only). For example, the following command line creates a library called *prog.klib* based on the functions in source program *prog.f90*:

```
f90 ... -WK,-inline_create=prog.klib prog.f90
```

When you specify this option the compiler creates only the library; it does not compile the source program. The following command compiles *samp.f90*, taking information about all eligible functions from *prog.klib.*

```
f90 ... −WK,−inl,−inlf=prog.klib samp.f90
```

When creating a library, you can specify only one *-inline_create* (*-ipa_create*) option. Therefore, you can create only one library at a time. The compiler overwrites any existing file with the same name as the library.

If you do not specify the *-inline* (*-ipa*) option along with the *-inline_create* (*-ipa_create*) option, the compiler includes all routines from the inlining universe in the library, if possible. If you specify *-inline=list* or *-ipa=list*, the compiler includes only the named routines in the library.

Tip: You do not have to generate your inlining or IPA library from the same source that will actually be linked into the running program. This capability can cause errors if misused, but it can also be useful. For example, you can write a library of hand−optimized assembly language routines, then construct an IPA library using Fortran routines that mimic the behavior of the assembly code. Thus, you can do parallelism analysis with IPA correctly, yet actually call the hand−optimized assembly routines.

Using the Inline−and−Copy Option

The *-inline_and_copy* (or *-inlc*) option functions like the *-inline* option, except that the compiler copies the unoptimized text of a routine into the transformed code file each time the routine is called or referenced. Use this option when inlining routines that are called from the file in which they are located. This option has no special effect when the routines being inlined are being taken from a library or separate source file.

When a routine has been inlined everywhere it is used, leaving it unoptimized saves compilation time. When a program involves multiple source files, the unoptimized routine is still available in case another source file contains a reference to it.

Note: The *-inline_and_copy* algorithm assumes that all CALLs and references to the routine precede the routine itself in the source file. If the routine is referenced after the text of the routine and the compiler cannot inline that particular call site, it invokes the unoptimized version of the routine.

Specifying Occurrences for Inlining and IPA

The loop level, depth, and manual options allow you to specify specific instances of the routines specified with the *-inline* or *-ipa* options to process.

Using Loop Level

The *-inline_looplevel=integer* (or *-inll=integer*) and *-ipa_looplevel=integer* (or *-ipall=integer*) options enable you to limit inlining and interprocedural analysis to routines that are referenced in deeply nested loops, where the reduced call overhead or enhanced optimization is multiplied.

Because inlining increases the size of the code, the extra paging and cache contention can actually slow down a program. Restricting inlining to routines used in DO loops multiplies the benefits of eliminating subroutine and function call overhead for a given amount of code space expansion. (If inlining appears to have slowed an application code, try using IPA, which has little effect on code space and the number of temporary variables.)

To determine which loops are most deeply nested, the compiler constructs a call graph to account for nesting of loops farther up the call chain. *integer* is defined relative to the most deeply nested leaf of the call graph. For example, if you specify 1 for *integer*, the compiler expands calls in only the most deeply nested loop. If you specify 2, the compiler expands routines in the deepest and second−deepest nested loops. Specifying a large number for *integer* enables inlining or IPA at any nesting level up to and including the integer value. If you do not specify *-inline/ipa_looplevel*, the loop level is 2.

Consider the code skeleton in Example 5−1.

Example 5−1 Skeleton of Nested Loops

```
PROGRAM MAIN
  ..
 CALL A    ------> SUBROUTINE A

  ..
 DO
  DO
   CALL B -----> SUBROUTINE B
  ENDDO             DO
 ENDDO                DO
                        CALL C -------> SUBROUTINE C
                      ENDDO
                    ENDDO
```

The CALL B is inside a doubly−nested loop and therefore, is more profitable for the compiler to expand than the CALL A. The CALL C is quadruply nested, so inlining C yields the greatest gain of the three.

For *-inline_looplevel=1*, only the functions called in the most deeply−nested call sites are inlined (the call to C in Example 5−1). *-inline_looplevel=2* inlines only routines called at the most deeply nested level and one loop less deeply nested. *-inline_looplevel=3* would be required to inline subroutine B, because its call is two loops less nested than the call to subroutine C. A value of 3 or greater causes the compiler to inline C into B, and then to inline the new B into the main program.

The calling tree written to the listing file includes the nesting depth level of each call in each program unit and the aggregate nesting depth (the sum of the nesting depths for each call site, starting from the main program). You can use this information to identify the best routines for inlining. (See "Setting the Listing Level".)

A routine that passes the *-inline_looplevel* test is inlined everywhere it is used, even places that are not in deeply nested loops. If some, but not all, invocations of a routine are to be expanded, use the C*$* INLINE or C*$* IPA directives just before each CALL/reference to be expanded (refer to "Fine−Tuning Inlining and IPA").

Depth of Nested Inlining

When a routine is expanded inline, it can contain references to other routines. The compiler must decide whether to recursively expand these references (which might themselves contain yet other references, and so on).

The *-inline_depth=integer* option (or *-ind=integer*) restricts the depth to which the compiler continues to attempt inlining already inlined routines. Valid values for *integer* are:

1−10	Specifies a depth to which inlining is limited. The default is 2.
0	Uses the default value (2).
−1	Limits inline expansion to only those routines that do not reference other routines (that is, only leaf routines are inlined). The compiler does not support any other negative values.

Recursive inlining can be quite expensive in compilation time. Exercise discretion in its use.

Note: There is no corresponding *-ipa_depth* option.

Enabling Manual Control

The *-inline_man* (or *-inm*) option enables recognition of the C*$* INLINE directive. This directive, described in "Fine−Tuning Inlining and IPA", allows you to select individual instances of routines to be inlined. The *-ipa_man* (or *-ipam*) option is the analogous option for the C*$* IPA directive.

Conditions That Prevent Inlining and IPA

This section lists conditions that prevent the compiler from inlining and analyzing subroutines and functions, whether from a library or source file. Many constructs that prevent inlining will also stop or restrict interprocedural analysis.

These are the conditions that inhibit inlining:

- Dummy and actual parameters are mismatched in type or class.
- Dummy parameters are missing.
- Actual parameters are missing and the corresponding dummy parameters are arrays.
- An actual parameter is a non−scalar expression (for example, A+B, where A and B are arrays).
- The number of actual parameters differs from the number of dummy parameters.
- The size of an array actual parameter differs from the array dummy parameter and the arrays cannot be made linear.
- The calling routine and called routine have mismatched COMMON declarations.
- The called routine has EQUIVALENCE statements (some of these can be handled).
- The called routine contains NAMELIST statements.
- The called routine has dynamic arrays.
- The CALL to be expanded has alternate return parameters.

Inlining is also inhibited when the routine to be inlined

- is too long (the limit is about 600 lines)
- contains a SAVE statement
- contains variables that are live−on−entry, even if they are not in explicit SAVE statements
- contains a DATA statement (DATA implies SAVE) and the variable is live on entry
- contains a CALL with a subroutine or function name as an argument
- contains a C*$*INLINE directive
- contains unsubscripted array references in I/O statements
- contains POINTER statements

Chapter 6

Using Directives and Assertions

This chapter contains the following sections:

- "Overview of Directives" explains the concept of directives and assertions and how they are coded.
- "Using Directives" summarizes all the directives, and describes the use of each.
- "Overview of Assertions" summarizes the supported assertions and describes how you can use them to inform the compiler.

Overview of Directives

The optimizations described in Chapter 4, "Controlling Scalar Optimization," and in Chapter 5, "Inlining and Interprocedural Analysis" are controlled by driver options. However, you can use specially−formatted comment lines called directives and assertions to control these optimizations over small units of program code. In this way, you can save time by having the compiler apply optimizations to only the segments of code that can benefit (such as deeply nested loops), or you can prevent the compiler from modifying code that should be left unchanged.

A directive is a statement that tells the compiler to treat the following source text in a particular way, for instance, by applying or not applying inlining to it. An assertion is a statement that tells the compiler something about the following source text, for example that it does or does not use equivalenced identifiers to access the same memory location. Both directives and assertions are written as specially−formatted comment lines. Because they are syntactically comments, directives and assertions are automatically ignored by any compiler that does not support them.

Note: In free−format source, a directive or assertion begins with the two characters "!$." In fixed−format source, a directive or assertions begins with "C$." In either case, the directive must begin in the first column of the source line in order to be recognized.

Recognizing Directives and Assertions

By default, the compiler recognizes all Silicon Graphics directives and assertions. You can use the *-WK,-directives* driver option to selectively enable/disable certain directives and assertions.

The *-directives=list* option (or *-dr=list*) specifies which type of directives to accept. *list* can contain any combination of the following letters:

a	Accept Silicon Graphics C*$* ASSERT assertions.
c	Accept Cray™ CDIR$ directives.
k	Accept Silicon Graphics C*$* and C$PAR directives.
s	Accepts directives based on the PCF (Parallel Computing Forum) X3H5 guidelines.
v	Accepts VAST™ CVD$ directives.

The default value for *list* is *acksv*. For example, *-WK,-directives=k* enables Silicon Graphics directives only, whereas *-WK,-directives=kas* enables Silicon Graphics directives and assertions and Sequent directives. To disable all of the above options, enter *-nodirectives* or *-directives* (without any values for *list*) on the command line.

In addition to specifying *-WK,-directives=a* in *list*, you can control whether the compiler accepts assertions using the C*$* ASSERTIONS and C*$* NOASSERTIONS directives (see "Using Assertions").

Using Directives

Directives enable, disable, or modify a feature of the compiler. Essentially, directives are driver options specified within the input file instead of on the command line. Unlike driver options, directives have no default setting. To invoke a directive, you must either toggle it on or set a desired value for its level.

Directives and Driver Options

Directives allow you to enable, disable, or modify a feature of the compiler in addition to, or instead of, driver options. Directives placed on the first line of the input file are called global directives. The compiler interprets them as if they appeared at the top of each program unit in the file. Use global directives to ensure that the program is compiled with the correct driver options. 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. (Set the global value using a driver option or a global directive.)

Some driver options act like global directives. Other driver options override directives. Many directives have corresponding driver options. If you specify conflicting settings to the driver and a directive, the compiler chooses the most restrictive setting. For Boolean options, if either the directive or the driver has the option turned off, it is considered off. For options that require a numeric value, the compiler uses the minimum of the driver setting and the directive setting.

Supported Directives

Table 6−1lists the directives supported by the compiler. In addition to the standard Silicon Graphics directives, the compiler supports the Cray and VAST directives listed in the table. The compiler maps these directives to corresponding Silicon Graphics assertions (see "Overview of Assertions" for details.)

Table 6−1 Directives Summary

Directive	Compatibility
C*$*ARCLIMIT(*n*)	Silicon Graphics
C*$*[NO]ASSERTIONS	Silicon Graphics
C*$* EACH_INVARIANT_IF_GROWTH(*n*)	Silicon Graphics
C*$* [NO]INLINE	Silicon Graphics
C*$* [NO]IPA	Silicon Graphics
C*$* MAX_INVARIANT_IF_GROWTH(*n*)	Silicon Graphics
C*$* OPTIMIZE(*n*)	Silicon Graphics
C*$* ROUNDOFF(*n*)	Silicon Graphics
C*$* SCALAR OPTIMIZE(*n*)	Silicon Graphics
C*$* UNROLL(*integer*[,*weight*])	Silicon Graphics

CDIR$ NO RECURRENCE	Cray
CVD$ [NO] DEPCHK	VAST
CVD$ [NO]LSTVAL	VAST

Keep in mind that directives begin with "C" (the comment marker) in fixed format source, and with "!" free−format source; but they always begin in column 1 in either case.

Controlling Internal Table Size

The C*$* ARCLIMIT(*integer*) directive sets the minimum size of the internal table that the compiler uses for data dependence analysis. The greater the value for *integer*, the more information the compiler can keep on complex loop nests. The maximum value and default value for *integer* is 5000.

When you specify this directive globally, it has the same effect as the *-arclimit* driver option (refer to "Controlling Internal Table Size" for details).

Setting Invariant IF Floating Limits

The C*$* EACH_INVARIANT_IF_GROWTH and the C*$* MAX_INVARIANT_IF_GROWTH directives control limits on the floating of invariant IF statements. This process generally involves duplicating the body of the loop, which can increase the amount of code considerably. Refer to"Setting Invariant IF Floating Limits" for details about invariant IF floating.

The C*$* EACH_INVARIANT_IF_GROWTH(*integer*) directive limits the total number of additional lines of code generated through invariant IF floating in a loop. You can control this limit globally with the *-each_invariant_if_growth* driver option (see "Setting Invariant IF Floating Limits").

You can limit the maximum amount of additional code generated in a program unit through invariant IF floating with the C*$* MAX_INVARIANT_IF_GROWTH(*integer*) directive. Use the *-max_invariant_if_growth* driver option to control this limit globally (see "Setting Invariant IF Floating Limits").

These directives are in effect until the end of the routine or until reset by a succeeding directive of the same type. Examine the loop in Example 6−1.

Example 6−1 Using Invariant−IF Directives

```
!*$*EACH_INVARIANT_IF_GROWTH(integer)
!*$*MAX_INVARIANT_IF_GROWTH(integer)
   DO I = ...
!*$*EACH_INVARIANT_IF_GROWTH(integer)
!*$*MAX_INVARIANT_IF_GROWTH(integer)
      DO J = ...
!*$*EACH_INVARIANT_IF_GROWTH(integer)
!*$*MAX_INVARIANT_IF_GROWTH(integer)
         DO K = ...
            section−1
            IF ( ) THEN
```

```
            section-2
         ELSE
            section-3
         ENDIF
            section-4
      ENDDO
   ENDDO
ENDDO
```

In floating the invariant IF out of the loop nest, the compiler honors the constraints set by the innermost directives first. If those constraints are satisfied, the invariant IF is floated from the inner loop. The middle pair of directives is tested and the invariant IF is floated from the middle loop as long as the restrictions established by these directives are not violated. The process of floating continues as long as the directive constraints are satisfied.

Setting Optimization Level

The C*$* OPTIMIZE(*integer*) directive sets the optimization level in the same way as the *-optimize* driver option. As you increase *integer*, the compiler performs more optimizations, and therefore takes longer to compile. Valid values for *integer* are:

0	Disables optimization.
1	Performs only simple optimizations. Enables induction variable recognition.
2	Performs lifetime analysis to determine when last–value assignment of scalars is necessary.
3	Recognizes triangular loops and attempts loop interchanging to improve memory referencing. Uses special case data dependence tests. Also, recognizes special index sets called wraparound variables.
4	Generates two versions of a loop, if necessary, to break a data dependence arc.
5	Enables array expansion and loop fusion.

Setting Variations in Round Off

The C*$* ROUNDOFF(*integer*) directive controls the amount of variation in round off error produced by optimization in the same way as the *-roundoff* driver option. Valid values for *integer* are:

0	Suppresses any transformations that change roundoff error.
1	Alter loops only if the order of computation remains the same.
2	Allows loop interchanging around arithmetic reductions if *-optimize* is at least 4. For example, the floating point expression A/B/C is computed as A/(B*C).
3	Recognizes REAL (float) induction variables if *-scalaropt*greater than 2 or *-optimize* is at least 1. Enables sum reductions. Enables memory management optimizations if *-scalaropt=3*(see "Performing Memory Management

Transformations" for details about memory management transformations).

Controlling Scalar Optimizations

The C*$* SCALAR OPTIMIZE(*integer*) directive controls the amount of standard scalar optimizations that the compiler performs. Unlike the *-WK,-scalaropt* driver option, the C*$* SCALAR OPTIMIZE directive sets the level of loop–based optimizations (such as loop fusion) only, and not straight–code optimizations (such as dead–code elimination). Valid values for *integer* are:

0 Suppresses any transformations that change roundoff error.

1 Performs expression simplification (which might generate various overflow or underflow errors) for expressions with operands between binary and unary operators, expressions that are inside trigonometric intrinsic functions returning integer values, and after forward substitution. Enables strength reduction. Performs intrinsic function simplification for *max* and *min*. Enables code floating if *-scalaropt* is at least 1. Allows loop interchanging around serial arithmetic reductions, if *-optimize* is at least 4. Allows loop rerolling, if *-scalaropt* is at least 2.

2 Allows loop interchanging around arithmetic reductions if *-optimize* is at least 4. For example, the floating point expression A/B/C is computed as A/(B*C).

3 Recognizes REAL (float) induction variables if *-scalaropt* greater than 2 or *-optimize* is at least 1. Enables sum reductions. Enables memory management optimizations if *-scalaropt=3* (see "Performing Memory Management Transformations" for details about memory management transformations).

Fine–Tuning Inlining and IPA

Chapter 5, "Inlining and Interprocedural Analysis," explains how to use inlining and IPA on an entire program. You can fine–tune inlining and IPA using the C*$*[NO] INLINE and C*$*[NO] IPA directives.

The compiler ignores these directives by default. They are enabled when you specify any inlining or IPA driver option, respectively, on the command line. The *-inline_manual* and *-ipa_manual* driver options enable these directives without activating the automatic inlining/algorithms.

The C*$* [NO] INLINE directive behaves like the *-inline* driver option, but allows you to specify which occurrences of a routine are actually inlined. The format for this directive is

```
C*$*[NO]INLINE [(name[,name ... ])] [HERE|ROUTINE|GLOBAL]
```

The possible options are:

name Specifies the routines to be inlined. If you do not specify a name, this directive will affect all routines in the program.

HERE Applies the INLINE directive only to the next line; occurrences of the named routines on that next line are inlined.

ROUTINE Inlines the named routines everywhere they appear in the current routine.

GLOBAL Inlines the named routines throughout the source file.

If you do not specify HERE, ROUTINE, or GLOBAL, the directive applies only to the next statement.

The C*$*NOINLINE form overrides the *-inline* driver option and so allows you to selectively disable inlining of the named routines at specific points.

In Example 6–2, the C*$*INLINE directive inlines the first call to *beta* but not the second.

Example 6–2 Using Directives to Control Inlining

```
       do i =1,n
!*$*INLINE (beta) HERE
          call beta (i,1)
       enddo
       call beta (n, 2)
```

Using the specifier ROUTINE rather than HERE in the example would inline both calls. This routine must be compiled with the *-inline_man* driver option for the compiler to recognize the C*$* INLINE directive.

The C*$* [NO] IPA directive is the analogous directive for interprocedural analysis. The format for this directive is

C*$*[NO]IPA [(*name* [,*name*...])] [HERE|ROUTINE|GLOBAL]

Overview of Assertions

Assertions provide the compiler with additional information about the source program. Sometimes assertions can improve optimization results.

Assertions can be unsafe because the compiler cannot verify the accuracy of the information provided. If you specify an incorrect assertion, the compiler–generated code might produce different results than the original serial program. If you suspect unsafe assertions are causing problems, use the *-WK,-nodirectives* driver option or the C*$* NO ASSERTIONS directive to tell the compiler to ignore all assertions.

Table 6–2 lists the supported assertions and the scope of their effect.

Table 6–2 Assertions and Their Duration

Assertion	Scope
C*$* ASSERT [NO] ARGUMENT ALIASING	Until reset
C*$* ASSERT [NO] BOUNDS VIOLATIONS	Until reset
C*$* ASSERT [NO] EQUIVALENCE HAZARD	Until reset
C*$* ASSERT NO RECURRENCE	Next loop
C*$* ASSERT RELATION (*name.xx. name*)	Next loop
C*$* ASSERT [NO] TEMPORARIES FOR CONSTANT ARGUMENTS	Next loop

Using Assertions

As with a directive, the compiler treats an assertion as a global assertion if it comes before all comments and statements in the file. That is, the compiler treats the assertion as if it were repeated at the top of each program unit in the file.

Some assertions (such as C*$* ASSERT RELATION) include variable names. If you specify them as global assertions, a program uses them only when those variable names appear in COMMON blocks or are dummy argument names to the subprogram. You cannot use global assertions to make relational assertions about variables that are local to a subprogram.

Many assertions, like directives, are active until another assertion resets them or the end of the program unit or file. Other assertions are active within a program unit, regardless of where they appear in that program unit.

Certain Cray and VAST *directives* function like Silicon Graphics *assertions.* The compiler maps these directives to the corresponding assertions. These directives are described along with the related assertions later in this chapter.

There is no guarantee that a specified assertion will have an effect. The compiler notes the information provided by the assertion and uses the information if it will help.

The C*$*[NO]ASSERTIONS directive instructs the compiler to accept or ignore assertions. The C*$* NO ASSERTIONS version is in effect until the next C*$* ASSERTIONS directive or the end of the program unit.

If you specify the *-directives* driver option without the assertions parameter (that is, *a*), the compiler will ignore assertions regardless of whether or not the file contains the C*$* ASSERTIONS directive. Refer to "Recognizing Directives and Assertions" for details on the *-directives* driver option.

Assertions About Data Dependence

In order to perform optimizations, the compiler must analyze the dependence relations between variables. Without full information on the dependence relations, the compiler cannot safely change the source in any way that would alter the order of execution.

In many cases the compiler can infer dependences from the source text. But in some cases there is not enough information. You can add information with assertions.

Known and Assumed Dependence

When the compiler can detect a possible dependence between two variables, but cannot determine the exact nature of the dependence, it creates an assumed dependence. The loop in Example 6−3illustrates simple forward dependence.

Example 6−3 Loop Containing Only Forward Dependence

```
      DO 10 i=2,n
10    X(i) = X(i) + X(i−1) * 0.5
```

In this example, the compiler can determine that between *X(i)* and *X(i−1)*there is a forward dependence, and that the distance is one. With this information the compiler could, for example, do loop unrolling. The loop in Example 6−4is similar, but has an assumed dependence as well.

Example 6–4 Loop Containing Forward and Assumed Dependence

```
      DO 10 i=2,n
10    X(i) = X(i) + X(i-1) * X(m)
```

The compiler cannot be sure if there is a dependence between *X*(*i*) and *X*(*m*). If it is always true that *m*>*n*, there is no dependence, and the code in Example 6–4can be unrolled, or parallelized, the same as the code in Example 6–3 However, if under some (or all) circumstances, $1 \le \mu < \nu$, then for some iterations of the loop *X*(*m*) will be different than in other iterations. It is no longer safe to unroll the loop or to parallelize it.

When assumed dependence blocks an optimization, you can supply information using an assertion of the true relation, or you can assert there are no dependences.

Asserting a Relationship

The assertion C*$* ASSERT RELATION(*name.xx.name*) specifies the relationship between two variables or between a variable and a constant. *name* is the variable or constant, and *xx* is any of the following: GT, GE, EQ, NE, LT, or LE. This assertion applies only to the next DO statement.

The C*$* ASSERT RELATION assertion includes one or two variable names. When specified globally, this assertion will only be used when the names appear in COMMON blocks or are dummy arguments to a subprogram. You cannot use global assertions to make relational assertions about variables that are local to a subprogram.

This assertion can be used to eliminate the assumed dependence in Example 6–4 If you know that *m* is always greater than *n* (and hence there is no dependence between *X*(*i*) and *X*(*m*)), you can use the assertion as shown in Example 6–5

Example 6–5 Asserting a Relationship

```
C*$* ASSERT RELATION (M .GT. N)
      DO 10 i=2,n
10    X(i) = X(i) + X(i-1) * X(m)
```

With this extra information the compiler can unroll or parallelize this loop.

Note: Many relationships of this type can be cheaply tested for at run time. The compiler attempts to answer questions of this sort by generating an IF statement that explicitly tests the relationship at run time. Occasionally, the compiler needs assistance, or you might want to squeeze that last bit of performance out of some critical loop by asserting some relationship rather than repeatedly checking it at run time.

Ignoring Data Dependence Conflicts

The assertion C*$* ASSERT NO RECURRENCE(*variable*) tells the compiler to ignore all data dependence conflicts caused by *variable* in the DO loop that follows it. For example, the following code tells the compiler to ignore all dependence arcs caused by the variable X in the loop:

This assertion can be used to eliminate the assumed dependence in Example 6–4 When you know that there is no dependences on *X*(*m*), you can tell the compiler so using this assertion, as shown in Example 6–6

Example 6–6 Loop with Dependences Denied

```
C*$* ASSERT NO RECURRENCE (X)
      DO 10 i=2,n
10    X(i) = X(i) + X(i-1) * X(m)
```

The assertion shown in Example 6–6 causes the compiler to ignore the assumed dependence, and also any other dependences involving *X*—including the real dependence between *X(i)* and *X(i−1)*. The name *M* could be used instead, so as to retain information about the real dependence.

The C*$* ASSERT NO RECURRENCE assertion applies only to the next DO loop. It cannot be specified as a global assertion.

In addition to the C*$* ASSERT NO RECURRENCE assertion, the compiler supports the Cray CDIR$ NORECURRENCE assertion and the VAST CVD$ NODEPCHK directive, which perform the same function.

Assertions About Aliasing

The compiler can deal with two kinds of aliasing. An alias is a situation in which the same memory location is accessible under two or more different and apparently unrelated names.

Using Equivalenced Variables

The C*$* ASSERT [NO] EQUIVALENCE HAZARD assertion tells the compiler that your code does not use equivalenced variables to refer to the same memory location inside one loop nest. Normally, EQUIVALENCE statements allow your code to use different variable names to refer to the same storage location. The *-WK,−assume=e* driver option acts like the global C*$* ASSERT EQUIVALENCE HAZARD assertion (see "Controlling Global Assumptions"). The C*$* ASSERT EQUIVALENCE HAZARD assertion is active until you reset it or until the end of the program.

In Example 6–7, if arrays E and F are equivalenced, but you know that the loop does not access overlapping sections, the use of C*$* ASSERT NO EQUIVALENCE HAZARD allows the compiler to parallelize the loop.

Example 6–7 Asserting Nonequivalence

```
      EQUIVALENCE ( E(1), F(101) )
C*$* ASSERT NO EQUIVALENCE HAZARD
      DO 10 I = 1,N
         E(I+1) = B(I)
         C(I) = F(I)
10    CONTINUE
```

Under optimization, Example 6–7 is converted to Example 6–8. (The C$DOACROSS directive is covered in Chapter 7.)

Example 6–8 Result of Asserting Nonequivalence

```
      EQUIVALENCE (E(1), F(101))
```

```
C*$* ASSERT NO EQUIVALENCE HAZARD
C$DOACROSS SHARE(N,E,B,C,F),LOCAL(I)
      DO 10 I=1,N
         E(I+1) = B(I)
         C(I) = F(I)
10    CONTINUE
```

Using Argument Aliasing

The C*$* ASSERT [NO] ARGUMENT ALIASING assertion allows the compiler to make assumptions about procedure dummy arguments. It is possible to call a procedure, specifying the same variable or array element in two or more positions of the argument list. Within the procedure, two or more dummy argument names, which apparently refer to different memory locations, actually refer to the same location.

In addition, when a procedure accesses a global variable, and you pass the same variable as an argument, an alias is created. In this case, the global variable can be thought of as an implicit argument.

According to the Fortran 77 standard, you can alias a dummy variable only if you do not modify (that is, write to) the aliased variable under either name.

The subroutine in Example 6–9violates the standard, because variable *A* is aliased in the subroutine (through dummy arguments *C* and *D*) and variable *X* is aliased (through global name *X* and dummy argument *E*).

Example 6–9 Two Kinds of Argument Aliasing

```
COMMON X,Y
REAL A,B
CALL SUB (A, A, X)
...
SUBROUTINE SUB(C,D,E)
COMMON X,Y
X =  ...
C =  ...
END
```

The driver option *-assume=a* acts like a global C*$* ASSERT ARGUMENT ALIASING assertion (see "Controlling Global Assumptions"). A C*$* ARGUMENT ALIASING assertion is active until it is reset or until the next routine begins.

Asserting Array Bounds Violations

The C*$* ASSERT [NO] BOUNDS VIOLATIONS assertion indicates that array subscript bounds may be violated during execution. If your program does not violate array subscript bounds, do not specify this assertion. When specified, this assertion is active until reset or until the end of the program. For formal parameters, the compiler treats a declared last dimension of (1) the same as (*).

The *-WK,-assert=b* driver option acts like a global C*$* ASSERT BOUNDS VIOLATIONS assertion.

In Example 6–10the compiler assumes the first loop nest conforms to the standard, and it therefore can optimize both loops. For example, the loops can be interchanged to improve memory referencing because no *A*(*I*,*J*) will overwrite an *A*(*I'*,*J*+*1*).

In the second nest, the assertion warns the compiler that the loop limit of the first array index (*I*) might violate the declared array bounds. The compiler plays it safe and optimizes only the rightmost array index.

Example 6–10 Asserting Array Bounds Safety

```
      DO 100 I = 1,M
         DO 100 J = 1,N
            A(I,J) = A(I,J) + B (I,J)
100   CONTINUE
C*$*ASSERT BOUNDS VIOLATIONS
      DO 200 I = 1,M
         DO 200 J = 1,N
            A(I,J) = A(I,J) + B (I,J)
200   CONTINUE
```

Note: The compiler always assumes that array references are within the array in order to make the rightmost index concurrentizable.

After optimization, the code of Example 6–10is converted to the form shown in Example 6–11.

Example 6–11 Result of Asserting Array Bounds Safety

```
!$DOACROSS SHARE(N,M,A,B),LOCAL(J,I)
   DO J=1,N
      DO I=1,M
         A(I,J) = A(I,J) + B (I,J)
      END DO
!*$*ASSERT BOUNDS VIOLATIONS
   DO I=1,M
!$DOACROSS SHARE(N,I,A,B),LOCAL(J)
      DO J=1,N
         A(I,J) = A(I,J) + B (I,J)
      END DO
   END DO
```

Asserting Safety of Constant Arguments

Sometimes the compiler does not perform certain transformations when their effects on the rest of the program are unclear. For example, usually the IF–to–intrinsic transformation changes code like that in Example 6–12

Example 6–12 Code Using Intrinsic Equivalent

```
SUBROUTINE X(I,N)
   IF (I .LT. N) I = N
END
```

The compiler recognizes the equivalence to intrinsic MAX, as in Example 6−13

Example 6−13 IF−to−Intrinsic Conversion

```
SUBROUTINE X(I,N)
   I = MAX(I,N)
END
```

Suppose the actual parameter passed as *I* is a constant, such as the following,

```
CALL X(1,N)
```

In that case, it would appear that the value of the constant 1 was being reassigned. In order to avoid this possibility, without additional information the compiler does not perform transformations within the subroutine.

Most compilers automatically put constant actual arguments into temporary variables to protect against this case. The C*$*ASSERT TEMPORARIES FOR CONSTANT ARGUMENTS assertion or the *-WK,-assume=c* driver option (the default) informs the compiler that constant parameters are protected. The NO version directs the compiler to avoid transformations that might change the values of constant parameters.

Chapter 7

Optimizing for Multiprocessors

This chapter contains these sections:

- "Overview of Parallel Optimization" provides an overview of parallel processing and a preview of this chapter.
- "Parallel Execution" discusses the fundamentals of parallel execution.
- "Writing Simple Parallel Loops" explains how to use the C$DOACROSS directive to parallelize single DO loops.
- "Analyzing Data Dependencies for Multiprocessing" describes how to analyze loops to determine whether they can be parallelized.
- "Breaking Data Dependencies" explains how to rewrite a loop with data dependencies so that some or all of the loop can run in parallel.
- "Adjusting the Work Quantum" describes how to determine whether the work performed in a loop is greater than the overhead associated with multiprocessing the loop.
- "Cache Effects" explains how to write loops that account for the effect of cache memory on performance.
- "Run−Time Control of Multiprocessing"tells of library functions and environment variables that give explicit run−time control over the degree of multiprocessing.
- "DOACROSS Implementation" discusses how multiprocessing is implemented in a DOACROSS routine.
- "Using PCF Directives" describes how to use the PCF directives to take advantage of a general model of parallelism.

Overview of Parallel Optimization

Using MIPSpro POWER Fortran 90—an extended version of MIPSpro Fortran 90 with extra features for optimization—you can compile your program so that, when you run it in a Silicon Graphics multiprocessor with available CPUs, your program will recruit the power of additional CPUs to work in parallel on the data. You request parallel optimization in one of two ways.

Automatic Parallelization

If you specify the −*pfa*driver option (see "Specifying Optimization Levels"), the compiler analyzes the program and attempts to parallelize every loop. You can see the result in the listing file (see"Setting the Listing Level"). In some cases the compiler will needlessly parallelize loops that do not contain enough work to justify parallel execution; and in other cases the compiler will not be able to parallelize loops owing to data dependencies.

Explicit Parallelization

You can also direct the compiler to parallelize specific loops, or specific procedures, or show it how to handle specific data dependencies. You do this by omitting the *–pfa* driver option, and instead writing directives in the program.

You have the choice of two different models for explicit parallelization:

- A simple model is based on the use of the DOACROSS directive. With it you enable specified DO–loops to execute in parallel, so that multiple iterations of the loop execute concurrently.
- A more general model is based on the Parallel Computing Forum (PCF) directives. With them, you can parallelize both looping and nonlooping sections of code, and you can specify critical sections and single–process sections of code.

This chapter discusses techniques for analyzing your program and converting it to multiprocessing operations. Chapter 8, "Compiling and Debugging Parallel Fortran," gives compilation and debugging instructions for parallel processing.

Parallel Execution

The basic idea of parallel optimization is that you can distribute parts of the program's work to two or more independent threads. Each thread executes asynchronously on a different CPU. By doing more than one piece of work at a time, your program finishes sooner than if all the work were done by one process on one CPU.

Process Structure

The processes that participate in the parallel execution of a program are arranged in a master/slave organization. The original process, created when the program is first loaded, is the master. It creates zero or more slave processes to assist it. The master process and each of the slave processes are called a thread of execution, or simply a thread.

Note: The term "thread" is used here as a convenient term for an independent executable entity within a program. Do not assume that it means any particular implementation of threads, for example "POSIX threads."

When the master process reaches a parallelized section of the program—which is usually a loop—the master assigns some of the work to each slave. The slaves and master execute concurrently, each on a different part of the loop or different section of code. As each slave completes its portion of the loop, it waits for further signals from the master, while the master resumes normal execution.

By default, the number of threads is set equal to the number of CPUs on the particular machine.You can control the number of threads used, either by setting environment variables before running the program, or from within the program by calling library routines.

Program Design

For multiprocessing to work correctly, the code executed by one thread must not depend on results produced by another thread. This is because you cannot predict the timing relationship between one thread and another. In particular, when a loop is parallelized, each iteration of the loop must produce the same answer regardless of when any other iteration of the loop is executed. Not all loops have this property. Loops without it cannot be correctly executed in parallel. (The same is true of loop unrolling. See "Assertions About Data Dependence".)

In the more general model of parallel execution, you specify sections of code that can run in parallel. When there are data dependencies between them, you can resolve these by delimiting a critical section that can be executed by only one process at a time.

Dynamic Scheduling

Since a long−running program spends most of its time within loops, parallel optimization focuses on the DO loop. The compiler tries to arrange it so that different iterations of the DO loop execute in parallel on multiple CPUs. For example, suppose a DO loop consisting of 20000 iterations will run on a machine with four available CPUs. Using the SIMPLE scheduling method (described in following topics), the first 5000 iterations run on one CPU, the next 5000 on another, and so on. The total execution time of that loop will be 1/4th the time for the nonparallel loop, plus the overhead time it takes to recruit, initialize, and release the added CPUs.

The multiprocessing code adjusts itself at run time to the number of CPUs actually present on the machine. Thus, if a 200−iteration loop is moved to a machine with only two available CPUs, it is automatically scheduled as two blocks of 100 iterations each, without any need to recompile or relink. In fact, multiprocessing code can be run on single−processor machines.

Parallel Directives

You control the parallelization of your program by writing directives in it. You have a choice of two families of directives (or you can mix them in the same program).

To provide compatibility for existing parallel programs, Silicon Graphics supports the directives for parallelism used by Sequent Computer Corporation. These directives allow you to parallelize specified DO−loops, while leaving the details to the Fortran compiler. To use this model, see"Writing Simple Parallel Loops".

You can also use the proposed Parallel Computing Forum (PCF) standard (ANSI−X3H5 91−0023−B Fortran language binding) directives. With these directives you can specify more general kinds of parallelization, and you can coordinate between the threads. To use the PCF model, see "Using PCF Directives". (The section "Writing Simple Parallel Loops" has important conceptual material you should read first.)

The directives are compiled with your source program. In addition, you manage parallel execution at run time using environment variables that are tested by the run−time library. Also, there are a number of special library routines that permit more direct, run−time control over the parallel execution (refer to "Run−Time Control of Multiprocessing"for more information.)

Writing Simple Parallel Loops

Six multiprocessing directives are used to parallelize specified loops:

C$DOACROSS	Specify multiprocessing parameters
C$&	Continue a C$DOACROSS directive to multiple lines
C$	Identify a line of code executed only when multiprocessing is active.
C$MP_SCHEDTYPE	Specify the way a loop is divided across CPUs
C$CHUNK	Specify the units of work into which a loop is divided.
C$COPYIN	Load a local copy of a COMMON block from the master process's version.

Note: In fixed−format source, directives start with "C," as comments must. In free−format source, they start with "!" instead, but are otherwise the same. In all cases, directives must start in the first column to be recognized (see "Recognizing Directives and Assertions").

When you compile a program with MIPSpro Fortran 90 (without the "Power" option), directives related to multiprocessing are treated as comments. This allows the identical source to be compiled with a single−processing compiler or by Fortran without the multiprocessing option.

The C$COPYIN directive is described under "Using Local COMMON Blocks". The other directives are described in the topics that follow.

Syntax of C$DOACROSS

The essential compiler directive for multiprocessing is C$DOACROSS. This directive marks a DO loop to be run in parallel, and specifies the details of how that loop is to be executed. The directive applies only to the following statement, which must be a DO loop. The C$DOACROSS directive has the form

`C$DOACROSS [`*clause* `[ [,]` *clause* `...]`

where valid values for each optional *clause* are

`IF (`*logical_expression*`)`
`{LOCAL | PRIVATE} (`*item*`[,`*item* `...] )`
`{SHARED | SHARE} (`*item*`[,`*item* `...])`
`{LASTLOCAL | LAST LOCAL} (`*item*`[,`*item* `...])`
`REDUCTION (`*item*`[,`*item* `...])`
`MP_SCHEDTYPE=`*mode*
`{CHUNK=`*integer_expression* `| BLOCKED(`*integer_expression*`)}`

The preferred form of the directive (as generated by WorkShop Pro MPF) uses the optional commas between clauses. The C$& directive can be used to extend C$DOACROSS to multiple lines, so each clause can be written on one line.

```
!$DOACROSS IF (N.GT.10000), CHUNK=100,
!$& MP_SCHEDTYPE=DYNAMIC
```

Using the IF Clause

You use the IF clause to decide at run time whether the loop is actually executed in parallel. The logical expression is tested at run−time. If its value is .TRUE., the loop is executed in parallel. If the expression is .FALSE., the loop is executed serially. Typically, the expression tests the number of times the loop will execute to be sure there is enough work in the loop to justify the overhead of parallel execution (see "Adjusting the Work Quantum"). In some cases, it tests the number of threads available (see "Using mp_numthreads and mp_set_numthreads").

Using the LOCAL, LASTLOCAL, and SHARE Clauses

The LOCAL, SHARE, and LASTLOCAL clauses specify lists of variables that need special treatment when used within the loop controlled by C$DOACROSS. Only the names of variables can appear in these clauses. An array variable is listed by name only, without any subscripts. Names of variables in COMMON blocks can not appear in a LOCAL list (but see "Using Local COMMON Blocks"). A variable can appear in only one of these clauses.

The LOCAL, SHARE, LASTLOCAL and REDUCTION lists are discussed at more length under "Analyzing Data Dependencies for Multiprocessing".

Using the LOCAL Clause

The LOCAL clause gives a list of variables that can be localized to each slave thread. Each iteration of the loop receives a private, uninitialized copy of a LOCAL variable. You should specify a variable as LOCAL when its value is calculated and used in the course of a single iteration of the loop and its value does not depend on any other iteration of the loop.

PRIVATE is a synonym for LOCAL, but LOCAL is preferred.

Using the LASTLOCAL Clause

The LASTLOCAL clause, like the LOCAL clause, specifies variables that can be localized to each iteration of the loop. In addition, the compiler generates code to copy the final value of the variable from the local copy of whichever slave process executes the logically−final iteration, and saves this value in the named variable for use in the serial code following the loop.

The loop iteration variable is given the LASTLOCAL attribute by default. However, if you do not need the value of the iteration variable after the loop, you can save a little time by specifying it as LOCAL instead.

The phrase LAST LOCAL is a synonym for LASTLOCAL, but LASTLOCAL is preferred.

Using the SHARE Clause

The SHARE clause specifies variables that must be common to all slave processes. When a variable is declared as SHARE, all iterations of the loop can safely share a single copy of the variable. You should declare a variable SHARE when:

- it is not modified in the loop
- it is an array in which each iteration of the loop accesses a different element

All variables except the loop−iteration variable are SHARE by default. The word SHARED is a synonym for SHARE, but SHARE is preferred.

Using the REDUCTION Clause

The REDUCTION clause specifies variables involved in a reduction operation. In a reduction operation, the compiler keeps local copies of the variables but combines them when it exits the loop. For an example and more discussion, see "Dealing With Reduction". For the relationship between reduction analysis and optimization levels, see "Controlling General Optimizations".

An element of the REDUCTION list must be a scalar variable and cannot be an array. However, it can be an individual element of an array specified by subscript.

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 SHARE list.

The four types of reductions supported are sum, product, min, and max. Sum and product reductions are recognized through the use of the + and * operators. The min and max reductions are recognized through the use of the MIN and MAX intrinsic functions.

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.

Using the CHUNK and MP_SCHEDTYPE Clauses

The CHUNK and MP_SCHEDTYPE clauses affect the way the compiler divides the work among the slave threads. These clauses do not affect the correctness of the loop. They are useful for tuning the performance of critical loops. See "Balancing the Load With Interleaving" for more details.

For the MP_SCHEDTYPE=*mode* clause, *mode* can have one of the following five values:

SIMPLE or STATIC	Divide the loop by the available CPUs and give each slave thread a contiguous group of iterations.
GSS or GUIDED	Dynamically vary the amount of work per thread, allocating smaller units as the loop approaches the end.
RUNTIME	Use environment variables to manage scheduling.
DYNAMIC	Slave threads compete for CHUNK−sized assignments.
INTERLEAVE or INTERLEAVED	Parcel out CHUNK−sized assignments to CPUs in rotation.

SIMPLE scheduling is the default unless CHUNK is specified; then DYNAMIC is the default. You can use different modes in different loops.

The CHUNK clause supplements the DYNAMIC and INTERLEAVE modes only. Instead of the CHUNK option, you can specify the *-WK,-chunk=n* driver option.

Using MP_SCHEDTYPE=SIMPLE

The simple scheduling method (MP_SCHEDTYPE=SIMPLE) divides the iterations of the loop among processes by dividing iterations into as many contiguous pieces as there are slave threads, assigning one piece to each process. The SIMPLE method has the lowest overhead, since each slave thread receives one assignment of work, after which it is finished. Use it when every loop iteration takes the same amount of time. When some iterations take longer than others, one slave can fall behind. The completion time of the loop is the completion of the most heavily−loaded CPU.

STATIC is a synonym for SIMPLE, but SIMPLE is preferred.

Using MP_SCHEDTYPE=GSS

The Guided Self−Scheduling algorithm (GSS) divides the iterations of the loop into pieces whose size varies depending on the number of iterations remaining. The initial pieces are not sufficient to finish the loop. When a slave thread finishes its piece, it returns for another piece of work.

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 slave entries into the critical section. Use GSS when there are relatively few slave CPUs and they are shared with other programs.

GUIDED is a synonym for GSS, but GSS is preferred.

Using MP_SCHEDTYPE=DYNAMIC

In dynamic scheduling, the iterations of the loop are broken into pieces of the size specified with the CHUNK clause. (CHUNK=1 is the default.)

As each slave process finishes a piece, it enters a critical section to take the next available piece. The smaller the CHUNK, the more of these entries that will occur, increasing overhead.

Using MP_SCHEDTYPE=INTERLEAVE

The interleave method breaks the iterations into pieces of the size specified by the CHUNK option, and execution of those pieces is interleaved among the processes. For example, if there are four processes and CHUNK=2, then the first process will execute iterations 1-2, 9-10, 17-18, …; the second process will execute 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. (This scheduling type is discussed further under "Balancing the Load With Interleaving".)

INTERLEAVED (with a final "D") is a synonym for INTERLEAVE, but the latter is preferred.

Using MP_SCHEDTYPE=RUNTIME

You can defer the choice of the scheduling method until run time using MP_SCHEDTYPE=RUNTIME. In this case, the scheduling routine examines values in environment variables to select one of the other methods. See "Environment Variables for RUNTIME Scheduling" for more details. Use this when you want to experiment with the performance of different scheduling types without recompiling.

C$DOACROSS Examples

Example 7–1shows a simple loop marked for parallel execution. The default scheduling is SIMPLE. By default, variable *I* is LASTLOCAL while *A* and *B* are SHARE.

Example 7–1 Simple Parallel Loop

```
C$DOACROSS
   DO 10 I = 1, 100
      A(I) = B(I)
10 CONTINUE
```

If you know that the value of *I* is not required following the loop, it can be made LOCAL. Example 7–2 shows the loop with the use of the variables specified.

Example 7–2 Simple Parallel Loop With LOCAL, SHARE

```
C$DOACROSS LOCAL(I), SHARE(A, B)
   DO 10 I = 1, 100
      A(I) = B(I)
10 CONTINUE
```

The loop in Example 7–3uses a variable *X* that is set and used locally to each iteration of the loop. Time will be saved by making this variable LOCAL as shown, so that each slave thread has its own copy. Since the loop variable *I* is not used after the loop, it is marked LOCAL also. This loop illustrates a parallel call to a function, SQRT. For more discussion, see "Parallel Procedure Calls".

Example 7–3 Parallel Loop With LOCAL and Function Call

```
C$DOACROSS LOCAL(I, X)
   DO 10 I = 1, N
      X = SQRT(A(I))
      B(I) = X*C(I) + X*D(I)
10 CONTINUE
```

In the loop shown in Example 7–4 the final values of *I* and *X* are needed after the loop completes. This gives a reason for the use of LASTLOCAL. Also, Example 7–4illustrates the use of the C$& directive to continue the directive.

Example 7–4 Parallel Loop With LASTLOCAL

```
C$DOACROSS LOCAL(Y,J), LASTLOCAL(I,X),
C$& SHARE(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
```

Note that in Example 7–4, *J* is listed as LOCAL. It is important to see that the inner loop using *J* is not a parallel loop (it has no C$DOACROSS directive). Each slave thread executes all the iterations for this inner loop within each iteration of the outer loop that it handles.

The variable *J* would be SHARE by default, but this would produce incorrect results. As multiple slave threads attempted to execute copies of the inner loop, they would interfere with each other's assignments to *J*. Hence it is important to specify *J* as LOCAL so that each slave has an independent copy.

Using C$

The C$ directive is considered a comment line except when multiprocessing. A line beginning with C$ is treated as a conditionally compiled Fortran statement. The rest of the line contains a standard Fortran statement. The statement is compiled only if multiprocessing is turned on. In this case, the "C$" or "!$" prefix is treated as blanks. You can use these directives to insert debugging statements, or to insert arbitrary code into the multiprocessed version. In Example 7–5, a diagnostic PRINT statement executes only when the program executes in multiprocessing mode.

Example 7–5 Use of "C$" Conditional Code

```
!$     PRINT *,'BEGIN MULTIPROCESSED LOOP'
!$DOACROSS LOCAL(I), SHARE(A,B)
    DO I = 1, 100
      CALL COMPUTE(A, B, I)
   END DO
```

Using C$MP_SCHEDTYPE and C$CHUNK

The C$MP_SCHEDTYPE=*mode* directive acts as default MP_SCHEDTYPE clause for following C$DOACROSS directives. The *mode* value is any of the modes listed in the section called "Using the CHUNK and MP_SCHEDTYPE Clauses". Any following C$DOACROSS directive that does not have an explicit MP_SCHEDTYPE clause is given the value specified in the last directive prior to the line, rather than the normal default.

The C$CHUNK=*integer_expression* directive affects the CHUNK clause of a C$DOACROSS in the same way that the C$MP_SCHEDTYPE directive affects the MP_SCHEDTYPE clause for all C$DOACROSS directives in scope. Both directives are in effect from the place they occur in the source until another corresponding directive is encountered or the end of the procedure is reached.

You can also invoke this functionality from the command line during a compile. The *-mp_schedtype=schedule_type* and *-chunk=integer* command line options have the effect of implicitly putting the corresponding directives as the first lines in the file.

Nesting C$DOACROSS

The compiler does not support direct nesting of C$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, a different form of nesting is allowed. A procedure that uses C$DOACROSS can be called from within a parallel region. This can be useful if a single procedure is called from several different places, sometimes from within a parallel loop and sometimes not.

Nesting in this way does not increase the parallelism. When the first C$DOACROSS loop is encountered, that loop is run in parallel. This fully occupies all the slave threads. If, in the parallel loop, a call is made to a routine that itself has a C$DOACROSS, that inner loop is executed serially.

Parallel Procedure Calls

It is possible to use functions and subroutines (intrinsic or user–defined) within a parallel loop. However, it is up to you to make sure that parallel invocations of a procedure do not interfere with one another. Intrinsic functions such as SQRT return a value that depends only on the input arguments; they do not modify global data and they do not use static storage. We say that such a function has no side effects.

Except for RANDOM_SEED and RANDOM_NUMBER, the standard Fortran 90 intrinsic functions have no side effects and can safely be called from a parallel loop. For the most part, the Fortran library functions listed in "Support for IRIX Kernel Functions"*do* have side effects and can *not* safely be included in a parallel loop.

For user–written procedures, it is the responsibility of the programmer to ensure that the routines can be correctly multiprocessed.

Caution: Do not use the *-static* option when compiling routines called within a parallel loop. This converts procedure local variables into static variables which cannot be used in parallel threads.

Tip: You cannot call RANDOM_NUMBER within a parallel loop because the slave threads, running concurrently within the function, would interfere with each other updating the seed values. Repeated or nonrandom values could be returned. In order to use random numbers in a parallel loop, first create an array containing one number for each iteration of the loop. Then treat that array as SHARE within the loop. Example 7–6illustrates the technique.

Example 7–6 Loop Using Random Numbers

```
    RANDOM_NUMBER(HARVEST = R(1:N))
!$DOACROSS LOCAL(I) SHARE(X,R)
    DO I = 1,N
      X(I) = PERTURB(X(I),R(I))
   END DO
```

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 even 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 or writes a memory location repeatedly, as long as no other iterations 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 not modified, there is no data dependence associated with it. (Remember that a variable can be modified through the action of a procedure call.)

The Fortran compiler supports four kinds of variable usage within a parallel loop: SHARE, LOCAL, LASTLOCAL, and REDUCTION. The basic meanings of these keywords are discussed under "Using the LOCAL, LASTLOCAL, and SHARE Clauses" and "Using the REDUCTION Clause".

It is often difficult to analyze loops for data dependence information. Each use of each variable must be examined to see if it fulfills the criteria for LOCAL, LASTLOCAL, SHARE, or REDUCTION. If all of the uses of all variables conform, the loop can be parallelized. If not, the loop cannot be parallelized as it stands, but possibly can be rewritten into an equivalent parallel form (see "Breaking Data Dependencies").

MIPSpro Power Fortran 90 analyzes loops for data dependence and automatically inserts the required C$DOACROSS directives when it determines that a loop has data independence. When Power Fortran 90 cannot determine whether the loop is independent, it produces a listing file detailing where the problems lie. You can use directives to specify the use of variables, assisting the compiler (see "Assertions About Data Dependence").

Simple Independence

The loop in Example 7−7 demonstrates simple data independence.

Example 7−7 Loop With Data Independence

```
      DO 10 I = 1,N
10    A(I) = X + B(I)*C(I)
```

Each iteration writes to a different location in *A*, and none of the variables appearing on the right−hand side is modified. This loop can be correctly run in parallel. All the variables are SHARE except for *I*, which is either LOCAL or LASTLOCAL, depending on whether its last value is used later.

Simple Dependence

The loop in Example 7−8 refers to *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 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.

Example 7−8 Loop With Stride−1 Dependence

```
      DO 20 I = 2,N
20    A(I) = B(I) - A(I-1)
```

The loop in Example 7−9 looks like the one in Example 7−8 The difference is that the stride of the DO loop (the fixed increment between each iteration) is now 2 rather than 1. Now *A*(*I*) is always an even−numbered element of *A*, while *A*(*I*−1) is always an odd−numbered element that never receives an assignment under the expression *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. There is no dependence. The loop can be run in parallel. Arrays *A* and *B* can be declared SHARE, while variable *I* should be declared LOCAL or LASTLOCAL.

Example 7−9 Loop With Stride−2 Dependence

```
      DO 20 I = 2,N,2
20    A(I) = B(I) - A(I-1)
```

Complicated Independence

At first glance, the loop in Example 7−10 looks like it cannot be run in parallel because it uses both *W*(*I*) and *W*(*I*−*K*). Closer inspection reveals that because the value of *I* varies between *K*+1 and *2*K*, the value of *I*−*K* goes from 1 to *K*. This means that the *W*(*I*−*K*) term varies from *W*(1) up to *W*(*K*), while the *W*(*I*) term varies from *W*(*K+1*) up to *W*(*2*K*). So *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. Variables *W*, *B*, and *K* can be declared SHARE, while variable *I* should be declared LOCAL or LASTLOCAL.

Example 7−10 Loop With Apparent Dependence

```
DO I = K+1, 2*K
   W(I) = W(I) + B(I,K) * W(I-K)
END DO
```

This example points out 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, though tedious. Fortunately, in practice most array indexing expressions are simple.

An Inconsequential Data Dependence

There is a data dependence in Example 7−11 because it is possible that at some point *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.

Example 7−11 Loop With Inconsequential Dependence

```
INDEX = SELECT(N)
DO I = 1, N
   A(I) = A(INDEX)
END DO
```

In this special case, you can ignore the dependence. 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 SHARE, while variable *I* should be declared LOCAL or LASTLOCAL.

Use of Local Variable

In Example 7−12, 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 LOCAL variable within the loop.

Example 7−12 Loop With Local Variable Use

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

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 SHARE, while variable *I* should be declared LOCAL or LASTLOCAL.

Rewritable Data Dependence

In Example 7−13, the value of INDX survives the loop iteration and is carried into the next iteration. This loop cannot be parallelized as it is written.

Example 7−13 Loop With Dependence

```
INDX = 0
DO I = 1, N
   INDX = INDX + I
   A(I) = B(I) + C(INDX)
END DO
```

Making *INDX* a LOCAL variable does not work because you need the value of *INDX* computed in the previous iteration. It is possible to rewrite this loop to make it parallel (see "Breaking Data Dependencies").

Exit Branch

The loop in Example 7–14contains an exit branch; that is, under certain conditions the flow of control exits the loop. The Fortran compiler cannot parallelize loops containing exit branches. While one slave thread might discover the exit condition, other slave threads working on later iterations would continue to run.

Example 7–14 Loop With Exit Branch

```
   DO I = 1, N
      IF (A(I) .LT. EPSILON) GOTO 320
      A(I) = A(I) * B(I)
   END DO
320 CONTINUE
```

Local Array

In Example 7–15, each iteration of the loop uses the same locations in the *D* array.

Example 7–15 Loop With Local Array Use

```
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
```

However, closer inspection reveals that the entire *D* array is being used as a temporary. This can be multiprocessed by declaring *D* to be LOCAL. The compiler allows arrays (even multidimensional arrays) to be LOCAL variables with one restriction: the size of the array must be known at compile time. The dimension bounds must be constants; the LOCAL array cannot have been declared using a variable or the asterisk syntax. Arrays TOTAL_DISTANCE and *A* can be declared SHARE, while array *D* and variable *I* should be declared LOCAL or LASTLOCAL.

Breaking Data Dependencies

Many loops that have data dependencies can be rewritten so that some or all of the loop can be run in parallel. The essential idea is to locate the expressions in the loop that cannot be made parallel and try to find another way to express them that does not depend on any other iteration of the loop. If this is not possible, 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.

The first step is to analyze the loop to discover the data dependencies (see "Analyzing Data Dependencies for Multiprocessing"). You can use WorkShop Pro MPF with MIPSpro Fortran 90 to identify the problem areas.

Sometimes the dependencies in a loop cannot be broken, and you must either accept the serial execution rate or try to discover a new parallel method of solving the problem. The rest of this section is devoted to a series of examples on how to deal with common situations. These are by no means exhaustive but cover some situations that happen in practice.

Value Derived From Iteration Count

Example 7−16is the same as Example 7−13 *INDX* has a value derived from the iteration count. The programmer, almost by instinct, has written code to build this value incrementally—this is obviously the "efficient" way to do it in serial code. However, because each value of *INDX* depends on the value in a previous iteration, the loop cannot be paralellized.

Example 7−16 Loop With Dependence

```
   INDX = 0
   DO I = 1, N
      INDX = INDX + I
      A(I) = B(I) + C(INDX)
   END DO
```

In fact, *INDX* can be derived directly from the current iteration number without reference to preceding iterations. As shown in Example 7−17, this can be done using code that would be "inefficient" in a serial program, since it does an "unnecessary" multiply and divide in each loop.

Example 7−17 Loop With Dependence Removed

```
!$DOACROSS LOCAL (I, INDX)
   DO I  = 1, N
      INDX = (I*(I+1))/2
      A(I) = B(I) + C(INDX)
   END DO
```

As a result, *INDX* can be designated a LOCAL variable, and the loop can now be multiprocessed.

Indirect Indexing

The loop in Example 7−18cannot be parallelized. It is the final statement that causes problems.

Example 7−18 Loop With Indirect Indexing

```
   DO 100 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
100 CONTINUE
```

The indexes *IXX* and *IYY* are computed in a complex way and depend on the values from the *IXOFFSET* and *IYOFFSET* arrays. It cannot be said that *TOTAL*(*IXX,IYY*) in one iteration of the loop will always be different from *TOTAL*(*IXX,IYY*) in every other iteration of the loop.

Example 7−19shows that the assignment to *TOTAL* can be pulled out into a separate loop by expanding *IXX* and *IYY* into arrays that retain intermediate values.

Example 7−19 Loop With Dependency Split to Other Loop

```
!$DOACROSS LOCAL(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 100 I = 1, N
      TOTAL(IXX(I),IYY(I)) = TOTAL(IXX(I), IYY(I)) + EPSILON
100 CONTINUE
```

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 will be true if *IXOFFSET* or *IYOFFSET* are permutation vectors.

Tip: Temporary arrays such as IXX and IYY in Example 7−19can be ALLOCATABLE, allocated just before needed and released after.

Before we leave this example, note that if we 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, of course, 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—and you need to document the dependence on this assertion, so that future program maintenance does not make it invalid.

Dealing With Recurrence

Example 7−20shows a simple example of recurrence, which exists when a value computed in one iteration is immediately used by another iteration.

Example 7−20 Loop With Recurrence Relation

```
DO I = 1,N
   X(I) = X(I-1) + Y(I)
END DO
```

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 an inner loop encloses the recurrence; in that case, try to parallelize the outer loop.

Dealing With Reduction

The operation in Example 7–21is known as a reduction. Reductions occur when the elements of an array of values are combined and reduced to a single value. This example is a sum reduction because the combining operation is addition.

Example 7–21 Loop With a Sum Reduction

```
ASUM  = 0.0
DO I = 1,N
   ASUM = ASUM + A(I)
END DO
```

Since the value of *ASUM* is carried from one loop iteration to the next, this loop cannot be parallelized. However, because this example merely sums the elements of *A*(*I*), we can rewrite it to accumulate multiple, independent subtotals. Then we can do much of the work in parallel. This approach is shown in Example 7–22 (The *mp_numthreads* library call is discussed under "Using mp_numthreads and mp_set_numthreads".)

Example 7–22 Loop With Partitioned Sum Reduction

```
   NUM_THREADS = MP_NUMTHREADS()
! IPIECE_SIZE = N/NUM_THREADS rounded up
   IPIECE_SIZE = (N + (NUM_THREADS -1)) / NUM_THREADS
   DO K = 1, NUM_THREADS
      PARTIAL_ASUM(K) = 0.0
! The first thread does 1 through IPIECE_SIZE, the second
! does IPIECE_SIZE + 1 through 2*IPIECE_SIZE, and so on.
! If M is not evenly divisible by num_threads, the MIN
! expression makes the last piece small.
      I_START = K*IPIECE_SIZE - IPIECE_SIZE +1
      I_FINISH = MIN(K*IPIECE_SIZE,N)
      DO I = I_START, I_FINISH
         PARTIAL_ASUM(K) = PARTIAL_ASUM(K) + A(I)
      END DO
   END DO
! Finally, add up the partial sums
   ASUM = 0.0
   DO I = 1, NUM_THREADS
      ASUM = ASUM + PARTIAL_ASUM(I)
   END DO
```

The outer loop on *K* can be run in parallel. The array pieces for the partial sums are contiguous; that is, each inner loop processes a span of elements of *A*(*I*) that are adjacent in memory. This results in good cache utilization.

The approach illustrated in Example 7−22is needed so often that automatic support is provided by the REDUCTION clause of C$DOACROSS. All of Example 7−22can be reduced to the simple code of Example 7−23

Example 7−23 Loop With Automatic Partitioned Reduction

```
   ASUM = 0.0
!$DOACROSS LOCAL (I), REDUCTION (ASUM)
   DO I = 1, N
      ASUM = ASUM + A(I)
   END DO
```

Types of Reductions

You can use the REDUCTION clause to automatically partition reductions based on four types of reduction operations:

sum	S = S+A(I)
product	P = P*A(I)
min	L = MIN(L,A(I))
max	M = MAX(M,A(I))

Multiple reductions are supported in a single loop, as shown in Example 7−24

Example 7−24 Loop With Four Reductions

```
!$DOACROSS LOCAL(I),REDUCTION(BG_SUM,BG_PROD,BG_MIN,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 compiler recognizes the type of reduction based on the operator or intrinsic function used. The number of available threads is calculated at runtime, and the arrays of partial values are allocated automatically.

Note: A partitioned reduction may not produce results identical to the serial reduction. Because computer arithmetic has limited precision, round−off errors accumulate in different ways when you sum the values in a different order. The final answer can differ, usually only in the last few decimal places. Either answer is "correct." If the difference is significant, neither answer is trustworthy.

Inner Reduction

One further example of a reduction transformation is noteworthy. Consider the nested loops in Example 7−25

Example 7−25 Reduction Nested in Outer Loop

```
   DO I = 1, N
      TOTAL = 0.0
      DO J = 1, M
         TOTAL = TOTAL + A(J,I)
      END DO
      B(I) = C(I) * TOTAL
   END DO
```

The inner loop could be parallelized with a REDUCTION clause, and this would be a reasonable optimization in the case provided that *N* is small and *M* is large.

However, first consider the outer loop. The variable *TOTAL* fulfills the criteria for a LOCAL 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. The outer loop can be parallelized, as shown inExample 7−26

Example 7−26 Parallel Outer Loop With Inner Reduction

```
!$DOACROSS LOCAL(I,J,TOTAL) SHARE(A)
   DO I = 1, N
      TOTAL = 0.0
      DO J = 1, M
         TOTAL = TOTAL + A(J,I)
      END DO
      B(I) = C(I) * TOTAL
   END DO
```

Iterations of the outer loop are performed in parallel. Each one has its own copy of *I*, *J*, and *TOTAL*, and executes the inner loop serially.

Coding Reductions Manually

When you have a reduction that is not a simple sum, product, MIN or MAX operation, it cannot be parallelized automatically. However, you can apply the technique shown in Example 7−22 The idea of partitioning an array to permit parallel computation, and then combining the partial results, is an important technique for breaking data dependence. This situation turns up again and again in various contexts and guises.

Tip: If multiple CPUs are not available, a program such as Example 7−22will waste time in overhead operations. You can write two versions of a reduction, one serial and one parallel, and encapsulate them in subroutines. Then you can dynamically choose which to execute using the C$ directive (see"Using C$").

Adjusting the Work Quantum

A certain amount of overhead is needed to initialize a parallel loop. If the work done in the loop is small, the parallel loop can actually run slower. To avoid this, make the amount of work inside the multiprocessed region as large as possible.

Using a Loop Interchange

In the nested loops of Example 7–27you could choose to parallelize the *J* loop or the *I* loop. In general, try to parallelize the outermost DO loop because it encloses the most work. However, you cannot parallelize the *K* loop in Example 7–27because different iterations of the *K* loop read and write the same values of *A*(*I,J*).

Example 7–27 Nested Loops

```
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
```

In this example, the *I* loop is the outermost one that can be parallelized. However, using the technique called loop interchange, you can reorder the loops to make this one the outermost, as shown in Example 7–28

Example 7–28 Nested Loops, Interchanged

```
!$DOACROSS LOCAL(I, J, K) SHARE(A, B, C)
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
```

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.

Conditional Parallelism

Occasionally, the only loop available to be parallelized has a fairly small amount of work. It may be worthwhile to force certain loops to run without parallelism or to select between a parallel version and a serial version, on the basis of the length of the loop.

The first loop in Example 7–29can never execute more thanfour iterations. It is not worth parallelizing such a loop unless there is an extraordinary amount of work in the body of the loop.

Example 7–29 Loop Not Worth Parallelizing

```
   J = (N/4) * 4
   DO I = J+1, N
      A(I) = A(I) + X*B(I)
   END DO
   DO I = 1, J, 4
      A(I) = A(I) + X*B(I)
      A(I+1) = A(I+1) + X*B(I+1)
      A(I+2) = A(I+2) + X*B(I+2)
      A(I+3) = A(I+3) + X*B(I+3)
   END DO
```

The second loop in Example 7−29has been optimized using manual loop unrolling of order four. Even so, this loop is worth parallelizing if *N* is big enough. In general, the overhead of initiating a parallel loop is roughly equivalent to 1,000 floating−point operations. Let *F* be the approximate number of floating−point operations in the loop, and let *P* be the number of available CPUs. If parallelization is to save time, the approximate inequality $F-F/P>1000$ must hold.

The revision in Example 7−30uses the IF clause on the DOACROSS directive to test if time can be saved by parallelization.

Example 7−30 Loop With Conditional Parallelization

```
   J = (N/4) * 4
   DO I = J+1, N
      A(I) = A(I) + X*B(I)
   END DO
!$DOACROSS LOCAL(I),
!$ IF ((J*8)-(J*8/MP_NUMTHREADS()).GE.1000)
   DO I = 1, J, 4 ! 8 flops per iter.
      A(I) = A(I) + X*B(I)
      A(I+1) = A(I+1) + X*B(I+1)
      A(I+2) = A(I+2) + X*B(I+2)
      A(I+3) = A(I+3) + X*B(I+3)
   END DO
```

Cache Effects

The cache memory of current Silicon Graphics systems has a major effect on performance. A CPU runs at or near its theoretical maximum speed only when all data and instructions are present in the cache. Whenever the CPU must fetch data from main memory, execution slows.

The technique for the best cache performance in Fortran is quite simple: make the loop step through the array in the same way that the array is laid out in memory. This means stepping through the array:

- without skipping any elements (with a "stride" of 1)
- with the leftmost subscript varying the fastest

Note that this optimization does not depend on multiprocessing, nor is it required in order for multiprocessing to work correctly. However, when you divide work between multiple CPUs, it is easy to introduce nonsequential array access. Always try to divide work so that each slave thread works on a contiguous span of array elements.

Parallelizing a Matrix Multiply

The loops in Example 7−31are the same as in Example 7−28 In order to get the most work into the outer loop, the *I* loop was interchanged with the *K* loop.

Example 7−31 Nested Loops, Interchanged

```
!$DOACROSS LOCAL(I, J, K) SHARE(A, B, C)
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
```

Unfortunately, to get the best cache performance, the *I* loop should be innermost. This is because *I* is the leftmost index in the references to arrays *A* and *B*. As the example stands, the innermost statement touches elements of *A* with a stride of *J*, and elements of *B* with a stride of *K*, resulting in unnecessarily frequent cache misses.

At the same time, to get the best multiprocessing performance, the outermost loop should be parallelized. In this case, you can perform one additional loop interchange of the *I* and *J* loops, and get the best of both optimizations. This is illustrated in Example 7−32

Example 7−32 Nested Loops Interchanged for Cache Performance

```
!$DOACROSS LOCAL(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
```

Trade−Offs Between Optimizations

Sometimes you must choose between the possible optimizations and their costs. The loop in Example 7−33can be parallelized on *I* but not on *J*.

Example 7−33 Vector Reduction

```
DO J = 1, N
```

```
   DO I = 1, M
      A(I) = A(I) + B(J)*C(I,J)
   END DO
END DO
```

As shown in Example 7–34 you could interchange the loops to put *I* on the outside, thus getting a bigger work quantum.

Example 7–34 Parallelized Vector Reduction

```
!$DOACROSS LOCAL(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 the loop steps through the *C* array with a stride of *I*—the second subscript varies the fastest. Supposing that *C* is, in total, much larger than the cache, all of *C* will be pumped through the cache, *I* times.

Another approach is to forego the loop interchange, and to parallelize only the inner loop. The inner loop can be seen as a sum reduction into *A*(*I*). The result is shown in Example 7–35

Example 7–35 Vector Reduction With Parallel Inner Loop

```
DO J = 1, N
!$DOACROSS LOCAL(I) REDUCTION( A(I) )
   DO I = 1, M
      A(I) = A(I) + B(J)*C(I,J)
   END DO
END DO
```

However, this approach entails initiating parallel execution *J* times; so *M* needs to be large for this approach to show any improvement.

You must trade off the various possible optimizations to find the combination that is right for the particular job.

Balancing the Load With Interleaving

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. However, some iterations can take significantly longer to complete than other iterations. This can be the natural result of the algorithm used.

At the end of a parallel region, the program has to wait for all slave threads to complete their tasks. If the work is not divided evenly, time is wasted waiting for the slowest process to finish.

Example 7–36 Loop With Iterations of Different Lengths

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

The code segment in Example 7−36can be parallelized on the outer loop. Because the inner loop goes from 1 to *I*, the first iterations of the outer loop will end long before the last iterations of the outer loop. In this example, this is easy to predict, so you can change the program as shown in Example 7−37. (See also "Using mp_numthreads and mp_set_numthreads".)

Example 7−37 Loop With Balanced Iterations

```
   NUM_THREADS = MP_NUMTHREADS()
!$DOACROSS LOCAL(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, it has been broken into interleaved blocks. Thus, each execution thread receives some small values of *I* and some large values of *I* , so that each slave thread has about the same amount of work to do. Interleaving usually, but not always, cures a load balancing problem.

You can use the *MP_SCHEDTYPE* clause to automatically perform this desirable transformation.

```
C$DOACROSS LOCAL (I,J), MP_SCHEDTYPE=INTERLEAVE
      DO 20 I = 1, N
         DO 10 J = 1, I
            A (J,I) = A(J,I) + B(J)*C(J)
10    CONTINUE
20    CONTINUE
```

Note that interleaving can cause poor cache performance because the array is no longer stepped through at a stride of 1. You can improve performance somewhat by adding a CHUNK=*integer_expression* clause. Usually 4 or 8 is a good value for *integer_expression*. Each small chunk has stride 1 to improve cache performance, while the chunks are interleaved to improve load balancing.

Interleaving is one possible scheduling mode. Both interleaving and the SIMPLE scheduling method 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 GSS schedules.

Comparing the output from *pixie* or from *prof* allows you to see how well the load is being balanced so you can compare the different methods of dividing the load. Refer to the discussion of the MP_SCHEDTYPE clause in "Using the CHUNK and MP_SCHEDTYPE Clauses" for more information.

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, some time can be spent waiting for the last processes to complete, even with near−perfect balance.

Run−Time Control of Multiprocessing

A number of features are provided so that you can control the use of multiprocessing at runtime. This section provides a brief explanation of these features, which are documented in the *mp*(3f) reference page.

Using mp_block and mp_unblock

The *mp_block* library function puts the slave threads into a blocked state using the IRIX system function *blockproc*. The slave threads stay blocked until a call is made to *mp_unblock*. These routines are useful if the job has bursts of parallelism separated by long stretches of single processing, as with an interactive program. You can block the slave processes so they consume CPU cycles only as needed, thus freeing the machine for other users. The Fortran system automatically unblocks the slaves on entering a parallel region should you neglect to do so.

Using mp_setup, mp_create, and mp_destroy

The *mp_setup*, *mp_create*, and *mp_destroy* library calls create and destroy threads of execution. This can be useful if the job has only one parallel portion or if the parallel parts are widely scattered. When you destroy the extra execution threads, they cannot consume system resources, but they must be re−created when needed. Frequent creation of threads can degrade performance. The *mp_block* and *mp_unblock* routines should be used in almost all cases.

The *mp_setup* library call takes no arguments. It creates the default number of processes as defined by previous calls to *mp_set_numthreads* (see "Using mp_numthreads and mp_set_numthreads"), by the environment variable MP_SET_NUMTHREADS (see "Environment Variables for Scheduling Control"), or by the number of CPUs on the current hardware platform. *mp_setup* is called automatically when the first parallel loop is entered to initialize the slave threads.

The *mp_create* call takes a single integer argument, the total number of execution threads desired. Note that the total number of threads includes the master thread. Thus, *mp_create*(*n*) creates one thread less than the value of its argument. *mp_destroy* takes no arguments—it destroys all the slave execution threads, leaving the master untouched.

When the slave threads end, they generate a SIGCLD signal. If your program has changed the signal handler to catch SIGCLD, it must be prepared to deal with this signal when *mp_destroy* is executed. This signal also occurs when the program exits; *mp_destroy* is called as part of normal cleanup when a parallel Fortran job terminates.

Using mp_blocktime

The Fortran slave threads wait by "spinning" (repetitive testing of a lock) until there is work to do. This makes them immediately available when a parallel region is reached. However, spinning consumes CPU resources. After a certain maximum amount of spinning, the slaves block themselves through *blockproc*. Once the slaves are blocked, it requires a system call to *unblockproc* to activate the slaves again (refer to the *unblockproc*(2) reference page for details). This slows the startup of a parallel loop.

This trade−off between response time and CPU usage can be adjusted with the *mp_blocktime* call. *mp_blocktime* takes a single integer argument that specifies the number of times to spin before blocking. By default, it is set to 10,000,000; this takes roughly one second. If called with an argument of 0, the slave threads will not block themselves no matter how much time has passed. Explicit calls to *mp_block*, however, do still block the threads.

This automatic blocking is transparent to the program. Blocked threads are automatically unblocked when a parallel region is reached.

Using mp_numthreads and mp_set_numthreads

Occasionally, you may want to know how many execution threads are available (for example, in order to call *mp_setup*, or for a test in a C$DOACROSS IF expression). The *mp_numthreads* function takes no arguments. It returns the total number of execution threads available for this job. The count includes the master thread.

The *mp_set_numthreads* subroutine takes a single−integer argument and changes the default number of threads to the specified value. A subsequent call to *mp_setup* will use the specified value rather than the original defaults.

Note: This call has an effect only when *mp_setup* is called. If the slave threads have already been created, this call will not change their number. To change the number of threads, use *mp_destroy*, then *mp_set_numthreads*, then *mp_setup*.

Using mp_my_threadnum

The *mp_my_threadnum* function takes no arguments. It returns a number indicating the number of thread executing the call. If there are *n* execution threads, the function call returns a value between zero and *n* - 1. The master thread is always thread zero. This function can be useful when parallelizing certain kinds of loops. Most of the time, the loop index variable can be used for the same purpose. Occasionally, the loop index may not be accessible, as, for example, when an external routine is called from within the parallel loop. This routine provides a mechanism for those cases.

Environment Variables for Scheduling Control

The MP_SET_NUMTHREADS, MP_BLOCKTIME, and MP_SETUP environment variables act as an implicit call to the corresponding routines of the same name, but they take effect at program start−up time.

For example, the *csh* command

```
% setenv MP_SET_NUMTHREADS 2
```

causes the program to create two threads regardless of the number of CPUs actually on the machine, just like the source statement

```
CALL MP_SET_NUMTHREADS(2)
```

Similarly, the *sh* commands

```
% set MP_BLOCKTIME 0
% export MP_BLOCKTIME
```

prevent the slave threads from autoblocking, just as does the statement

```
call mp_blocktime (0)
```

For compatibility with older releases, the environment variable NUM_THREADS is supported as a synonym for MP_SET_NUMTHREADS.

To help support networks with several multiprocessors and several CPUs, the environment variable MP_SET_NUMTHREADS also accepts an expression involving integers +, -, *min*, *max*, and the special symbol *all*, which stands for "the number of CPUs on the current machine."

For example, the following command selects the number of threads to be two fewer than the total number of CPUs (but always at least one):

```
% setenv MP_SET_NUMTHREADS max(1,all-2)
```

Environment Variables for Load−Sensitive Scheduling

In an environment with long running jobs and varying workloads, it may be preferable to vary the number of threads during execution of some jobs.

If the environment variable MP_SUGNUMTHD has a non−null value when the program starts, the run−time library creates an additional, asynchronous process that periodically wakes up and monitors the system load. When idle processors exist, this process increases the number of threads, up to the maximum set by MP_SET_NUMTHREADS. When the system load increases, the process decreases the number of threads, possibly to as few as one. When MP_SUGNUMTHD has no value, this feature is disabled and multithreading works as before.

The environment variables MP_SUGNUMTHD_MIN and MP_SUGNUMTHD_MAX are used to limit this feature as desired. When MP_SUGNUMTHD_MIN is set to an integer value between 1 and MP_SET_NUMTHREADS, the process will not decrease the number of threads below that value.

When MP_SUGNUMTHD_MAX is set to an integer value between the minimum number of threads and MP_SET_NUMTHREADS, the process does not increase the number of threads above that value.

If you set any value in the environment variable MP_SUGNUMTHD_VERBOSE, informational messages are written to *stderr* whenever the process changes the number of threads in use.

Calls to *mp_numthreads* and *mp_set_numthreads* are taken as a sign that the application depends on the number of threads in use. The number in use is frozen upon either of these calls; and if MP_SUGNUMTHD_VERBOSE is set, a message to that effect is written to *stderr*.

Environment Variables for RUNTIME Scheduling

These environment variables specify the type of scheduling to use on DOACROSS loops that have their scheduling type set to RUNTIME. For example, the following *csh* commands cause loops with the

RUNTIME scheduling type to be executed as interleaved loops with a chunk size of 4:

```
% setenv MP_SCHEDTYPE INTERLEAVE
% setenv CHUNK 4
```

The defaults are the same as on the C$DOACROSS directive: if neither variable is set, SIMPLE scheduling is assumed; if MP_SCHEDTYPE is set but CHUNK is not set, a CHUNK of 1 is assumed. If CHUNK is set, but MP_SCHEDTYPE is not, DYNAMIC scheduling is assumed.

Using mp_setlock, mp_unsetlock, mp_barrier

The *mp_setlock*, *mp_unsetlock*, and *mp_barrier* subroutines provide convenient (although limited) access to the locking and barrier functions provided by the IRIX functions *ussetlock*, *usunsetlock*, and *barrier*. These subroutines are convenient because you do not need to initialize them; calls such as *usconfig* and *usinit* are done automatically. The limitation is that there is only one lock and one barrier. For most programs, this amount is sufficient. If your program requires more complex or flexible locking facilities, use the *ussetlock* family of subroutines directly.

Using Local COMMON Blocks

Variables in COMMON blocks are static, and if there is an assignment to a static variable within a loop, the loop can't be parallelized.

A special *ld* option 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 loops into parallel form.

Only a named COMMON can be made process−local (blank COMMON may not be made local). The COMMON block must not be initialized by executable code, not by DATA statements.

To create a local COMMON block, give the special loader directive *-Xlocal* followed by a list of COMMON block names. Note that the external name of a COMMON block known to the loader has a trailing underscore and is not surrounded by slashes. For example, the command

```
% f90 −mp a.out −Xlocal foo_
```

makes the COMMON block /FOO/ a local COMMON block in the resulting *a.out* file. You can specify multiple *-Xlocal* options if necessary.

It is occasionally desirable to be able to copy values from the master thread's version of the COMMON block into the slave thread's version. The special directive C$COPYIN allows this. It has the form

`C$COPYIN` *item* `[,` *item* …`]`

Each *item* must be a member of a localized COMMON block. It can be a variable, an array, an individual element of an array, or the entire COMMON block.

For example,

```
!$COPYIN x,y, /foo/, a(i)
```

propagates the values for *x* and *y*, all the values in the COMMON block FOO, and the *i*th element of array *a*. All these items must be members of local COMMON blocks. Note that this directive is translated into executable code, so in this example *i* is evaluated at the time this statement is executed.

Compatibility With sproc

The parallelism used in Fortran is implemented using the IRIX system call *sproc*. It is not a good idea to attempt to use both parallel loops and explicit *sproc* calls. It is possible, but there are several restrictions:

- Any explicit threads you create may not execute $DOACROSS loops. Only the original thread is allowed to do this.
- The calls to routines like *mp_block* and *mp_destroy* apply only to the threads created by *mp_create* or to those automatically created when the Fortran job starts; they have no effect on any user−created threads.
- Calls to routines such as *m_get_numprocs* do not apply to threads created explicitly. However, the Fortran−created slave threads are ordinary subprocesses; so using the system function *kill* with the arguments 0 and a signal number (for example, *kill* (0,9)) to signal all members of the process group will kill the threads used to execute C$DOACROSS.
- If you choose to intercept the SIGCLD signal, you must be prepared to receive this signal when the threads used for the C$DOACROSS loops exit; this occurs when *mp_destroy* is called or at program termination.
- Note in particular that *m_fork* is implemented using *sproc*, so it is not legal to *m_fork* a family of processes that each subsequently executes C$DOACROSS loops. Only the original thread can execute C$DOACROSS loops.

DOACROSS Implementation

This section discusses how multiprocessing is implemented in a loop controlled by C$DOACROSS. This information is useful when you use a debugger or interpret the results of an execution profile.

Loop Transformation

When the Fortran compiler encounters a C$DOACROSS directive, it puts the body of the corresponding DO loop into a separate subroutine and replaces the loop with a call to a special library routine *__mp_parallel_do*.

The newly created routine is named by appending *.pregion* to the name of the original routine, followed by the number of the parallel loop in the routine, where 0 is the first loop. For example, the first parallel loop in a routine named *foo* is named *foo.pregion0*, the second parallel loop is *foo.pregion1*, and so on.

If a loop occurs in the main routine and if that routine has not been given a name by the PROGRAM statement, its name is assumed to be *main*. Any variables declared to be local in the original C$DOACROSS statement are declared as local variables in the created routine. References to SHARE variables are resolved by referring back to the original routine.

Because the created routine is now just a DO loop, the routine uses subroutine arguments to specify which part of the loop a particular process is to execute. The created routine has three arguments: the starting value for the index, the number of times to execute the loop, and a special flag word.

Consider the subroutine in Example 7−38

Example 7−38 Subroutine With Parallel Loop

```
SUBROUTINE EXAMPLE(A, B, C, N)
   REAL A(*), B(*), C(*)
!$DOACROSS LOCAL(I,X)
   DO I = 1, N
      X = A(I)*B(I)
      C(I) = X + X**2
   END DO
   C(N) = A(1) + B(2)
   RETURN
END
```

The compiler generates the new subroutine shown in Example 7−39to contain the parallelized loop code. The name of the generated routine is derived from the containing subroutine, EXAMPLE.

Example 7−39 Generated "pregion" Subroutine

```
SUBROUTINE EXAMPLE.pregion0( _LOCAL_START, _LOCAL_NTRIP, &
_THREADINFO)
   INTEGER*4 _LOCAL_START
   INTEGER*4 _LOCAL_NTRIP
   INTEGER*4 _THREADINFO
   INTEGER*4 I
   REAL X
   INTEGER*4 _DUMMY
   I = _LOCAL_START
   DO _DUMMY = 1,_LOCAL_NTRIP
      X = A(I)*B(I)
      C(I) = X + X**2
      I = I + 1
   END DO
END
```

Executing Created Routines

The set of processes that cooperate to execute the parallel Fortran job are members of a process share group created by *sproc*, anIRIX system call. The process share group is created by special startup routines that are used only when the executable is linked with the *-mp* option, which enables multiprocessing.

The first process is the master process. It executes all the nonparallel portions of the code. The other processes are slave processes; they are controlled by the routine *mp_slave_control.* When they are inactive, they wait in the special routine *__mp_slave_wait_for_work.*

The *__mp_parallel_do* routine divides the work and signals the slaves. The master process then calls the created routine to do its share of the work. When a slave is signaled, it wakes up from the wait loop, calculates which iterations of the spooled *DO* loop it is to execute, and then calls the routine with the appropriate arguments. When the routine returns, the slave reports that it has finished and returns to *__mp_slave_wait_for_work*.

When the master completes its execution of its portion of the spooled routine, it waits in the special routine *mp_wait_for_loop_completion* until all the slaves have completed processing. The master then returns to the main routine and continues execution.

Using PCF Directives

The compiler supports a more general model of parallelism, in addition to the simple loop−level parallelism offered by the C$DOACROSS directive. 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 compiler supports this model through compiler directives, rather than extensions to the source language.

The main concept in this model is the parallel section, which can be any arbitrary section of code (not just a *DO* loop). Within the parallel region, you designate work−sharing constructs to specify how the work is divided among separate threads. The parallel region can also contain a critical section construct, where exactly one process executes at a time.

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 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.

The PCF directives, summarized in Table 7−1, implement the general model of parallelism. They look like Fortran comments: always starting in column one, and beginning with a "C$PAR" in fixed−form source or "!$PAR"’ in free−form source.

The compiler recognizes these directives when multiprocessing is enabled with either the *-mp* or *−pfa* driver option. If multiprocessing is not enabled, the compiler treats these statements as comments.

Table 7−1 Summary of PCF Directives

Directive	Description
C$PAR BARRIER	Ensures that each process waits until all processes reach the barrier before proceeding.
C$PAR CRITICAL SECTION C$PAR END CRITICAL SECTION	Ensures that the enclosed block of code is executed by only one process at a time by using a lock variable.
C$PAR PARALLEL C$PAR END PARALLEL	Encloses a parallel region, which includes work−sharing constructs and critical sections.
C$PAR PARALLEL DO	Precedes a single DO loop for which separate iterations are executed by different processes. This directive is equivalent to the C$DOACROSS directive.

C$PAR PDO C$PAR END PDO	Separate iterations of the enclosed loop are executed by different processes. This directive must be inside a parallel region.
C$PAR PSECTION[S] C$PAR END PSECTION[S]	Parcels out each block of code in turn to a process.
C$PAR SECTION	Signifies a starting line for an individual section within a parallel section.
C$PAR SINGLE PROCESS C$PAR END SINGLE PROCESS	Ensures that the enclosed block of code is executed by exactly one process.
C$PAR &	Continues a PCF directive onto multiple lines.

C$PAR &

Occasionally, the clauses in PCF directives are longer than one line. You can use the C$PAR & directive to continue a directive onto multiple lines. For example,

```
!$PAR PARALLEL local(i,j)
!$PAR& shared(a,n,index_x,index_y,cur_max,
!$PAR& big_max,bmax_x,bmax_y)
```

Parallel Region

A parallel region encloses any number of PCF constructs (described in "Using PCF Directives"). It signifies the boundary within which slave threads execute. Slave threads are created at entry to the region if necessary. A user program can contain any number of parallel regions. The syntax of the parallel region is

```
C$PAR PARALLEL [clause [[,] clause]...]
           code
C$PAR END PARALLEL
```

where valid *clauses* are

```
IF ( logical_expression )
{LOCAL | PRIVATE}(item [,item ...])
{SHARED | SHARE}(item [,item ...])
```

The IF, LOCAL, and SHARED clauses have the same meaning as in the C$DOACROSS directive (refer to "Writing Simple Parallel Loops").

Note: The preferred form of the directive uses no commas between the clauses. The SHARED keyword is preferred over SHARE, and LOCAL is preferred over PRIVATE.

In Example 7–40, all threads enter the parallel region and call the subroutine named *foo*, passing the number of the thread executing the call.

Example 7–40 Subroutine With Parallel Region

```
   subroutine ex1(index)
      integer i
!$PAR PARALLEL LOCAL(i)
```

```
!$PAR PARALLEL LOCAL(i)
      i = mp_my_threadnum()
      call foo(i)
!$PAR END PARALLEL
   end
```

PCF Work–Sharing Constructs

The principal PCF constructs are the work–sharing constructs. (The other types are critical sections and barriers.) The work–sharing constructs direct the application of slave threads to code. The work–sharing constructs are:

- parallel DO
- PDO
- parallel sections
- single process

All master and slave threads synchronize at the bottom of any work–sharing construct. None of the threads continue past the end of the construct until they all have completed execution within that construct.

If specified, the PDO, parallel section, and single process constructs must appear inside of a parallel region, which creates the threads. The parallel DO construct cannot. Specifying a parallel DO construct inside of a parallel region produces a syntax error.

Parallel DO

The parallel DO construct is the same as the C$DOACROSS directive—it calls for parallelizing the single DO loop that immediately follows it.

Conceptually, parallel DO is 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. The syntax of the parallel DO construct is

`C$PAR PARALLEL DO [`*clause* `[[,]` *clause*`]...]`

"Syntax of C$DOACROSS" describes valid values for *clause.* The only difference is in the MP_SCHEDTYPE=*mode* clause. For the C$PAR PARALLEL DO directive, the keyword MP_SCHEDTYPE= is optional; you can simply specify *mode.*

PDO

PDO is a generalization of parallel DO to loops of any kind. Each thread inside the enclosing parallel region executes a separate iteration of the loop within the PDO construct. The syntax of the PDO construct, which can only be specified within a parallel region, is

`C$PAR PDO [`*clause* `[[,]` *clause*`]...]`
   code
`[C$PAR END PDO [NOWAIT]]`

where valid values for *clause* are

```
{LOCAL | PRIVATE} (item[,item ...])
{LASTLOCAL | LAST LOCAL} (item[,item ...]
(ORDERED)
sched
chunk
```

LOCAL, LASTLOCAL, s*ched*, and *chunk* have the same meaning as in the C$DOACROSS directive (refer to "Writing Simple Parallel Loops"). Note in particular that it is legal to declare a data item as LOCAL in a PDO even if it was declared as SHARED in the enclosing parallel region. The (ORDERED) clause is equivalent to a *sched* clause of DYNAMIC and a *chunk* clause of 1. The parentheses are required.

LASTLOCAL is preferred over LAST LOCAL and LOCAL is preferred over PRIVATE.

The END PDO directive is optional. If specified, this directive must appear immediately after the end of a 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 will wait until all have reached the directive before proceeding.

Example 7–41shows an example of the PDO construct.

Example 7–41 Simple PDO Structure

```
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
   enddo
!$PAR END PARALLEL
end
```

The effect of this example could be achieved with a parallel DO or with a C$DOACROSS directive. In fact, the compiler recognizes this as a special case and generates the same (more efficient) code as for a C$ DOACROSS directive.

Parallel Sections

The PCF parallel sections construct is a parallel version of the Fortran 90 SELECT statement. Each block of code is parcelled out in turn to a separate thread. The syntax of the parallel sections construct is

```
C$PAR PSECTION[S] [clause]
   code
[C$PAR SECTION
   code] ...
C$PAR END PSECTION[S] [NOWAIT]
```

The only valid value for *clause* is

`{LOCAL | PRIVATE} (`*item* `[,`*item*`])`

LOCAL is preferred over PRIVATE and has the same meaning as for the C$DOACROSS directive (refer to "Syntax of C$DOACROSS"). Note in particular that 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 will wait until all have reached the END PSECTION directive before proceeding.

Parallel sections must appear within a parallel region. They can contain critical section constructs (described in "Critical Sections") but cannot contain any of the following types of constructs:

- PDO
- parallel DO or C$ DOACROSS
- single process

Each section 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.

Example 7–42illustrates parallel sections. The first thread to enter the parallel sections construct executes the first section; the second thread executes the second section; and a third thread, if one exists, executes the third section. If the parallel region is executed by only two threads, whichever thread first finishes its section executes the remaining section.

Example 7–42 Parallel Sections

```
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
   enddo
!$PAR SECTION
   call normalize(c,n3)
   do i = 1, n3
      c(i) = c(i) + 1.0
   enddo
!$PAR END PSECTION
```

```
!$PAR END PARALLEL
end
```

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 C$PAR END PSECTION directive until all threads are finished.

This example uses DO loops, but a parallel section can contain any code—provide it has no data dependency on other sections. Be aware of the significant overhead of a parallel construct. Make sure the amount of work performed is enough to outweigh the extra overhead.

The sections within a parallel sections 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.

Single Process

The single process construct, which can only be specified within a parallel region, ensures that a block of code is executed by exactly one process. The syntax of the single process construct is

```
C$PAR SINGLE PROCESS [clause]
    code
C$PAR END SINGLE PROCESS [NOWAIT]
```

The only valid value for clause is

`{LOCAL | PRIVATE} (`*item* `[,`*item*`])`

LOCAL is preferred over PRIVATE and has the same meaning as for the C$ DOACROSS directive (refer to "Syntax of C$DOACROSS"). Note in particular that 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 will wait until all have reached the directive before proceeding.

This construct is semantically equivalent to a parallel sections construct with only one section. The single process construct provides a more descriptive syntax.The first thread to reach the single process section executes the code in that block. All other threads wait at the end of the section until the code has been executed.

Critical Sections

The critical section construct protects a block of code with a lock so that it is executed by only one thread at a time. Another process arriving at the critical section must wait until the current process has finished it. Threads do not synchronize at the bottom of a critical section, as they do at the end of a work−sharing construct.

The critical section construct can appear anywhere in a program, inside and outside a parallel region and even within a C$ DOACROSS loop. The syntax of the critical section construct is

```
C$PAR CRITICAL SECTION [ ( lock_variable ) ]
      code
C$PAR END CRITICAL SECTION
```

The *lock_variable* is an optional integer variable that must be initialized to zero. The parenthesis are required around its name. If you do not specify *lock_variable*, the compiler automatically supplies one.

Multiple critical section constructs inside the same parallel region are normally independent of each other. However, if they use the same explicit *lock_variable*, they are linked and only one process can execute in any of the linked critical sections at one time.

Barrier Constructs

A barrier construct ensures that each process waits until all processes reach the barrier before proceeding. There is an implicit barrier at the end of each work–sharing construct (unless NOWAIT is specified). The syntax of the barrier construct is

```
C$PAR BARRIER
```

Restrictions

The three work–sharing constructs, PDO, PSECTION, and SINGLE PROCESS, must be executed by all the threads executing in the parallel region (or 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
```

This code will hang forever when run with enough processes. One or more process will be stuck at the C$PAR END SINGLE PROCESS directive waiting for all the threads to arrive. But threads with numbers less than 6 never take the appropriate branch, and 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.

Note that 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 lexically 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.

A Few Words About Efficiency

The more general PCF constructs are typically slower than the special case parallelism offered by the C$DOACROSS directive. They are slower because of the extra synchronization required. When a C$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 C$DOACROSS loops typically execute faster than a single parallel region with several PDO constructs. Limit your use of the parallel region construct to those few cases that actually need it.

Chapter 8

Compiling and Debugging Parallel Fortran

This chapter gives instructions on how to compile and debug a parallel Fortran program. It contains the following sections:

- "Compiling and Running" explains how to compile and run a parallel Fortran program.
- "Profiling a Parallel Fortran Program" describes how to use the system profiler, *prof*, to examine execution profiles.
- "Debugging Parallel Fortran" presents some standard techniques for debugging a parallel Fortran program.

This chapter assumes you have read Chapter 7, "Optimizing for Multiprocessors," and have reviewed the techniques and vocabulary for parallel processing in the IRIX environment.

Compiling and Running

After you have written a program for parallel processing, you should first debug your program in a single−processor environment by compiling it without parallel optimization. You can also debug your program using the CASEvision/WorkShop debugger, which is sold as a separate product. After your program has executed successfully on a single processor, you can compile it for multiprocessing.

To turn on multiprocessing, use the driver option *-mp*. This option causes the compiler to generate multiprocessing code for the particular files being compiled. When linking, you can combine object files produced with and without the *-mp* option. When you use either *f90* or *ld* to link a program containing any object files compiled with *-mp*, you must again use *-mp* so that the correct libraries are linked.

Using the -static Option

The −*static*driver option causes procedure local variables to be allocated statically, not on the process stack as normal. However, the multiprocessing implementation demands some use of the stack to allow multiple threads of execution to execute the same code simultaneously. Therefore, parallel code regions are effectively compiled with the *-automatic* option, even if the routine enclosing them is compiled with *-static*.

This means that SHARE variables in a parallel section behave with *-static* semantics, but that LOCAL variables in a parallel section do not (see "Using the LOCAL, LASTLOCAL, and SHARE Clauses").

Finally, if a parallel region calls an external procedure, that procedure cannot be compiled with *-static*. As noted under "Parallel Procedure Calls", to call a procedure that uses static variables from multiple, concurrent threads would create race conditions and incorrect results. You can mix static and multiprocessed object files in the same executable; the restriction is that static variables cannot be modified from within a parallel section.

Examples of Compiling

The following examples illustrate compiling code using *-mp*. The following command line

```
% f90 −mp foo.f
```

compiles and links the Fortran program *foo.f* into a multiprocessor executable.

In this example

```
% f90 −c −mp −O2 snark.f
```

the Fortran routines in the file *snark.f* are compiled with multiprocess code generation enabled. The optimizer is also used. A standard *snark.o* binary is produced, which must be linked:

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

Here, the *-mp* option signals the linker to use the Fortran multiprocessing library. The file *bellman.o* need not have been compiled with the *-mp* option, although it could have been.

After linking, the resulting executable can be run like any standard executable. Creating multiple execution threads, and running, synchronizing, and terminating them, are all handled automatically.

When an executable has been linked with *-mp*, the Fortran initialization routines determine how many parallel threads of execution to create. This determination occurs each time the program starts; the number of threads is not compiled into the code. The default is to use whichever is less: 4, or the number of processors that are on the machine. (This number will be the value returned by the system call *sysmp(* MP_NAPROCS*)*; see the *sysmp*(2) reference page.) You can override the default using environment variable MP_SET_NUMTHREADS or using a library call, as discussed under "Run−Time Control of Multiprocessing".

Profiling a Parallel Fortran Program

After converting a program, you need to examine execution profiles to judge the effectiveness of the transformation. Good execution profiles of the program are crucial to help you focus on the loops consuming the most time.

IRIX provides profiling tools that can be used on Fortran parallel programs. Both *pixie* (see the *pixie*(1) reference page) and pc−sample profiling can be used. On jobs that use multiple threads, both methods create a separate profile data file for each thread. You can use the standard profile analyzer *prof* (see the *prof*(1) reference page) to examine this output. The *MIPS Compiling and Performance Tuning Guide* has details about using *prof* and *pixie.*

The profile of a Fortran parallel job is different from a standard profile. As mentioned in "DOACROSS Implementation", 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. Comparing the amount of time spent in each loop by the various threads shows how well the workload is balanced.

In addition to the loops, the profile 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 how parallel the program is.

Debugging Parallel Fortran

This section presents some techniques to assist in debugging a parallel program.

General Debugging Hints

- Debugging a multiprocessed program is much more difficult than debugging a single−processor program. Therefore you should do as much debugging as possible on the single−processor version.
- Try to isolate the problem as much as possible. Ideally, try to reduce the problem to a single C$DOACROSS loop or PCF parallel section.
- Before debugging a multiprocessed program, change the order of the iterations on the parallel DO loop on a single−processor version. If the loop can be multiprocessed, then the iterations can execute in any order and produce the same answer. If the loop cannot be multiprocessed, changing the order frequently causes the single−processor version to fail, and standard single−process debugging techniques can be used to find the problem.

Example 8−1contains a bug: the 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 C$DOACROSS is a mistake.

Example 8−1 Erroneous C$DOACROSS

```
!$doacross local(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, the example could be rewritten as shown in Example 8−2

Example 8−2 Corrected use of C$DOACROSS

```
!$doacross local(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. When a multiprocessed loop gives the wrong answer, make the following checks.

- Check the LOCAL variables when the code runs correctly as a single process but fails when multiprocessed. Carefully 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 LOCAL. Be sure to include the index of any loop nested inside the parallel loop.

A related problem occurs when you need the final value of a variable but the variable is declared LOCAL rather than LASTLOCAL. 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 LOCAL when in fact it should be LASTLOCAL. To combat this problem, simply declare all the LOCAL variables LASTLOCAL when debugging a loop.

- 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.
- Check for the use of uninitialized variables. Some programs assume uninitialized variables have the value 0. This works with the *-static* option, but without it, uninitialized values assume the value left on the stack. When compiling with *-mp*, the program executes differently and the stack contents are different. You should suspect this type of problem when a program compiled with *-mp* and run on a single processor gives a different result when it is compiled without *-mp*. One way to track down a problem of this type is to compile suspected routines with *-static*. If an uninitialized variable is the problem, it should be fixed by initializing the variable rather than by continuing to compile *-static*.
- Try compiling with the *-C* option for range checking on array references. 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.
- If the analysis of the loop was incorrect, one or more arrays that are SHARE may have data dependencies. This sort of error is seen only when running multiprocessed code. When stepping through the code in the debugger, the program executes correctly. In fact, this sort of error often is seen only intermittently, with the program working correctly most of the time.
- The most likely candidates for this error are arrays with complicated subscripts. If the array subscripts are simply the index variables of a DO loop, the analysis is probably correct. If the subscripts are more involved, they are a good choice to examine first.
- If you suspect this type of error, as a final resort print out all the values of all the subscripts on each iteration through the loop. Then use **uniq**() (see the *uniq*(1) reference page) to look for duplicates. If duplicates are found, then there is a data dependency.

Appendix A

Run−Time Error Codes

The error codes listed on Table A−1can be detected by the Fortran 90 run−time support. These codes are all in the range 100−199. In addition, the run−time code can return IRIX error codes in the range of 1−99. IRIX error codes are documented in the *intro*(2) reference page.

When the run−time library detects an error, the following things can occur:

- If the error is detected in an I/O statement that specifies an IOSTAT variable, the code is stored in the variable.
- If the error is detected in an I/O statement that specifies an ERR label, control passes to that label.
- When the error is not in an I/O statement, or when neither the IOSTAT nor ERR clause is given, the error message is displayed on the standard error file and the program terminates.

When the program is terminated by an error, a core file is produced if the *f77_dump_flag* environment variable is defined and set to *y*. The core file can be used with *dbx* to inspect the state of the program at termination.

PERROR and related procedures described in the *perror*(3f) reference page are used to display error messages.

Note: Some error codes in Table A−1are returned by the Fortran 77 run−time libraries, but are not returned by Fortran 90. Such codes are marked with (F77) at the end of the explanation.

In most cases these codes are related to I/O features that are not supported by Fortran 90, such as indexed I/O. They are retained in Table A−1because you may find them coded in Fortran 77 source programs you are porting to Fortran 90.

Table A−1 Run−Time Error Messages

Number	Message/Cause
100	`error in format` Illegal characters in the FORMAT in the latest I/O statement.
101	`out of space for I/O unit table` Out of virtual space that can be allocated for the I/O unit table.
102	`formatted io not allowed` Cannot do formatted I/O on logical units opened for unformatted I/O.
103	`unformatted io not allowed` Cannot do unformatted I/O on logical units opened for formatted I/O.
104	`direct io not allowed` Cannot do direct I/O on sequential file.
105	`sequential io not allowed` Cannot do sequential I/O on this file.
106	`can't backspace file` Cannot perform BACKSPACE/REWIND on file.
107	`null file name` Filename specification in OPEN statement is null.

108 `can'tstat file`
Cannot get descriptive information about this file from file system.

109 `file already connected`
The specified filename has already been opened as a different logical unit.

110 `off end of record`
Attempt to do I/O beyond the end of the record.

111 `truncation failed in endfile`
An error occurred while closing the file.

112 `incomprehensible list input`
Input data for list−directed read contains invalid character for its data type.

113 `out of free space`
Cannot allocate virtual memory space on the system.

114 `unit not connected`
Attempt to do I/O on unit that has not been opened or cannot be opened.

115 `read unexpected character`
Unexpected character encountered in formatted or directed read.

116 `blank logical input field`
Invalid character encountered for logical value.

117 `bad variable type`
Specified type for the namelist element is invalid. This error is most likely caused by incompatible versions of the front end and the run−time I/O library.

118 `bad namelist name`
The specified namelist name cannot be found in the input data file.

119 `variable not in namelist`
The namelist variable name in the input data file does not belong to the specified namelist.

120 `no end record`
$END is not found at the end of the namelist input data file.

121 `namelist subscript out of range`
The array subscript of the character substring value in the input data file exceeds the range for that array or character string.

122 `negative repeat count`
The repeat count in the input data file is less than or equal to zero.

123 `illegal operation for unit`
You cannot set your own buffer on direct unformatted files.

124 `off beginning of record`
Format edit descriptor causes positioning to go off the beginning of the record.

125 `no * after repeat count`
An asterisk (*) is expected after an integer repeat count.

126 `'new' file exists`
The file is opened as new but already exists.

127 `can't find 'old' file`
The file is opened as old but does not exist.

130 `illegal argument`
Invalid value in the I/O control list.

131 `duplicate key value on write`
Cannot write a key that already exists. (F77)

132 `indexed file not open`
Cannot perform indexed I/O on an unopened file. (F77)

133 `bad isam argument`
The indexed I/O library function receives a bad argument because of a corrupted index file or bad run−time I/O libraries. (F77)

134 `bad key description`
The key description is invalid. (F77)

135 `too many open indexed files`
Cannot have more than 32 open indexed files. (F77)

136 `corrupted isam file`
The indexed file format is not recognizable. This error is usually caused by a corrupted file. (F77)

137 `isam file not opened for exclusive access`
Cannot obtain lock on the indexed file. (F77)

138 `record locked`
The record has already been locked by another process. (F77)

138 `key already exists`
The key specification in the OPEN statement has already been specified. (F77)

140 `cannot delete primary key`
DELETE cannot be executed on a primary key. (F77)

141 `beginning or end of file reached`
The index for the specified key points beyond the length of the indexed data file. This error is probably because of corrupted ISAM files or a bad indexed I/O run−time library. (F77)

142 `cannot find requested record`
The requested key for indexed READ does not exist. (F77)

143 `current record not defined`
Cannot execute REWRITE, UNLOCK, or DELETE before doing a READ to define the current record. (F77)

144 `isam file is exclusively locked`
The indexed file has been exclusively locked by another process. (F77)

145 `filename too long`
The filename exceeds 128 characters.

147 `recordtoo long`
Indexed record is too long to read. (F77)

148 `key structure does not match file structure`
Mismatch between the key specifications in the OPEN statement and the indexed file. (F77)

149 `direct access on an indexed file not allowed`
Cannot have direct−access I/O on an indexed file. (F77)

150 `keyed access on a sequential file not allowed`
Cannot use keyed access together with sequential organization. (F77)

151 `keyed access on a relative file not allowed`
Cannot use keyed access together with relative organization. (Note 1)

152 `append access on an indexed file not allowed`
Cannot use append access together with indexed organization. (F77)

153 `must specify record length`
A record length specification is required when opening a direct or keyed

access file.

154 `key field value type does not match key type`
The type of the given key value does not match the type specified in the OPEN statement for that key. (F77)

155 `character key field value length too long`
The length of the character key value exceeds the length specification for that key. (F77)

156 `fixed record on sequential file not allowed`
RECORDTYPE='fixed' cannot be used with a sequential file. (F77)

157 `variable records allowed only on unformatted sequential file`
RECORDTYPE='variable' can be used only with an unformatted sequential file. (F77)

158 `stream records allowed only on formatted sequential  file`
RECORDTYPE='stream_lf' can be used only with a formatted sequential file. (F77)

159 `maximum number of records in direct access file exceeded`
The specified record is bigger than the MAXREC= value used in the OPEN statement.

160 `attempt to write to a read-only file`
User does not have write permission on the file.

161 `must specify key descriptions`
Must specify all the keys when opening an indexed file. (F77)

162 `carriage control not allowed for unformatted units`
CARRIAGECONTROL can be used only on a formatted file. (F77)

163 `indexed files only`
Indexed I/O can be done only on logical units that have been opened for indexed (keyed) access. (F77)

164 `cannot use on indexed file`
Illegal I/O operation on an indexed (keyed) file. (F77)

165 `cannot use on indexed or append file`
Illegal I/O operation on an indexed (keyed) or append file. (F77)

167 `invalid code in format specification`
Unknown code is encountered in format specification.

168 `invalid record number in direct access file`
The specified record number is less than 1.

169 `cannot have endfile record on non-sequential file`
Cannot have an endfile on a direct- or keyed-access file.

170 `cannot position within current file`
Cannot perform *fseek*() on a file opened for sequential unformatted I/O.

171 `cannot have sequential records on direct access file`
Cannot do sequential formatted I/O on a file opened for direct access.

173 `cannot read from stdout`
Attempt to read from stdout (unit 6).

174 `cannot write to stdin`
Attempt to write to stdin (unit 0).

176	`illegal specifier` The I/O control list contains an invalid value for one of the I/O specifiers; for example, ACCESS='INDEXED'.
177	`end-of-record condition occurs with PAD=NO`The iolist contains more items than the record can supply, and the PAD=YES specifier was not given.
178	`EOR= specifier requires ADVANCE=NO`An EOR label can be given only with nonadvancing sequential I/O.
179	`SIZE= specifier requires ADVANCE=NO`The number of characters read is only meaningful with nonadvancing sequential I/O.
180	`attempt to read from a writeonly file` User does not have read permission on the file.
181	`direct unformatted io not allowed` Direct unformatted file cannot be used with this I/O operation.
182	`cannot open a directory` The name specified in FILE= must be the name of a file, not a directory.
183	`subscript out of bounds` The exit status returned when a program compiled with the -C option has an array subscript that is out of range.
184	`function not declared as varargs` Variable arguments used in a call to a subroutine not declared in a $VARARGS directive (refer to the *varargs*(3f) reference page)
185	`internal error` Internal run−time library error.

Appendix B

Converting From Fortran 77

A correct, standard−conforming Fortran 77 program is also a correct, standard−conforming Fortran 90 program. However, the Silicon Graphics, Inc. (SGI) implementations of Fortran 77 permit some nonstandard features that are not supported by Fortran 90.

This appendix lists the problems you may encounter when converting some nonstandard Fortran 77 source features to Fortran 90. The following main topics are covered:

- "Differences in Source Format" covers the use of fixed− versus free−form source and mixed−case letters in source files.
- "Differences in Data Declaration" discusses syntax differences in declaring literals, scalar variables, and structures.
- "Differences in Intrinsic Procedures" discusses the conversion of extended intrinsics, and the use of unsupported specific names.
- "Differences in I/O Processing" documents the many special I/O features not supported in Fortran 90

Note: This appendix addresses *conversion to Fortran 90*. You do not need this information when writing *new* Fortran 90 code. Also, since Fortran 77 object modules can be linked with Fortran 90 object modules, it is always an option to leave all of a program (or specific modules of it) in Fortran 77.

Tip: Refer to "Internet Resources for Fortran 90 Users"; at least one Fortran 77 to Fortran 90 source conversion program is available on the network.

Differences in Source Format

Fortran 77 supports two source formats, 72−column and 120−column. Both forms are accepted by Fortran 90. Specify either the *−col72* (or *−fixedform*) option, or the *−col120* option. If you specify neither, Fortran 90 expects any source file with the suffix *.f* to have the *−col72* format. (See "Specifying Source File Format".)

Fortran 90 supports free−form source input. Specify *−freeform* to force an input file to be treated as free−form source. Fortran 90 assumes *−freeform* for input files with the suffix *.f90*.

Since MIPSpro Fortran 90 supports only 8−bit ASCII characters (see "Character Values and Literals"), there is no question of overflowing the maximum source line length due to nondefault character literals (section 3.3.1 in the *MIPSpro Fortran 90 Handbook*).

MIPSpro Fortran 90 extends the Fortran 90 standard by accepting up to 99 continuation lines.

No Fortran 77 compiler supports Fortran 90 style free−form source, so do not use it for any module that must be backward−compatible.

Letter Case in Source Files

Letter case is ignored in Fortran source input. SGI Fortran compilers remove the difference between lowercase and uppercase by converting all input text (except for literal character constants) to lowercase before processing it. This is the same in Fortran 90 and Fortran 77.

The only visible result of this policy is that the names of external procedures are always in lowercase when they are recorded in the object file. Other languages, such as C, permit mixed−case input, and therefore can have mixed−case entry−point names in their object files. Mixed−case entry point names cannot be linked with Fortran object files.

Some (not all) SGI Fortran 77 compilers support a compile option *−U* specifying that input text should not be forced to lowercase. This has the effect of allowing mixed−case external names to appear in object files. The *−U* option is not supported by Fortran 90. It is not possible to link directly from a Fortran 90 program to a procedure with a mixed−case name.

Differences in Data Declaration

There are a number of small differences in the syntax of data declaration. Most can be converted by simple text changes.

Differences in Scalar Declarations

In Fortran 77, you can specify the precision of a numeric variable by appending an asterisk and a size to the basic type; for example, REAL*16 or LOGICAL*8.

In Fortran 90, the comparable way of specifying precision is by use of the "kind" parameter in the declaration. However, the SGI implementation of Fortran 90 does permit the use of the asterisk−length notation from Fortran 77. In Example B−1, the declarations in each pair are equivalent.

Example B−1 Syntax of Numeric Precision

```
INTEGER (KIND=8) eight_byte_int
INTEGER*8 eight_byte_int
REAL (16) quad_real
REAL*16 quad_real
```

Note: The use of asterisk−length for numeric precision is nonportable, and should not be used in new Fortran 90 code.

Asterisk−length syntax can readily be converted using a tool such as *awk* or *sed*. The *sed* command file in Example B−2 converts asterisk−length declarations to "kind" declarations.

Example B−2 Program to Convert Asterisk Notation

```
s/INTEGER\*\([248]\)/integer (kind=\1)/
s/LOGICAL\*\([248]\)/logical (kind=\1)/
s/REAL\*\([48]\)/real (kind=\1)/
s/REAL\*16/real (kind=16)/
s/COMPLEX\*8/complex (kind=4)/
s/COMPLEX\*16/complex (kind=8)/
s/COMPLEX\*32/complex (kind=16)/
```

The use of "kind=" is optional. In programs where the keywords might have different letter cases, the search patterns need to allow for mixed case, as in Example B−3

Example B−3 Conversion Allowing for Mixed−Case Keywords

```
s/[Rr][Ee][Aa][Ll]\*\([48]\)/real(\1)/
```

Differences in the Syntax of Literals

Literal values are written differently in some cases.

Binary, Octal, and Hexadecimal Literals

Fortran 77 supports a variety of syntax forms when integer literal values are expressed in bases other than decimal. Some of these are supported by Fortran 90 and some are not, as shown in Table B−1

Table B−1 Forms of Integer Literal Values

Number Base	Example of Form	Supported
2 (binary)	`B"00100111"`	Yes
8 (octal)	`O"33"`	Yes
16 (hexadecimal)	`X"1B", X"1b"`	No
16 (Hexadecimal)	`Z"1B", Z'1b'`	Yes
16 (Hexadecimal)	`$1B`	No

Of these forms, the use of prefix dollar−sign for hexadecimal is very common. In Fortran 90, this form causes a syntax error message ("Unexpected symbol"). It is relatively simple to change this form of constant to the supported form using a tool such as *sed* or *perl*. For example, the *sed* command in Example B−4converts hexadecimal literals.

Example B−4 Conversion of $−Form Literal to "Z" Form

```
sed 's/\$\([A-F0-9][A-F0-9]*\)/Z"\1"/g'
```

Conversion of the unsupported "X" form to the "Z" form is equally simple.

Floating−Point Literals

The Fortran 77 syntax for real and double−precision literal values—that is, the use of exponent "E" to mean single−precision and "D" to mean double−precision—is accepted by Fortran 90. Use of these literals is shown in Example B−5

Example B−5 Syntax for Precision of Floating−Point Literals

```
real (kind=4) r4
real (kind=8) r8
r4 = 1.665E-2
r8 = 2.468D12
```

Some SGI Fortran 77 compilers accept the use of exponent−letter "Q" to mean a quad−precision floating−point constant. This usage is also accepted by SGI Fortran 90. InExample B−6 the first assignment uses the Fortran 90 syntax of appending a kind−parameter to the constant, while the second uses the Fortran 77 syntax of an exponent letter of "Q." The two assignments are effectively the same.

Example B−6 Equivalent Syntax for Quad−Precision Literals

```
real (kind=16) rp,rq
rp = 1.2345678901234567890E7_16
rq = 1.2345678901234567890Q7
```

Note: The "Q" exponent is a nonportable extension of Fortran 90. When writing new programs, write literal values with a _16 suffix. "Q" notation can be converted to standard form using *sed*, as shown in Example B−7.

Example B−7 Conversion of Q−Exponent Literal to Suffix−16

```
sed 's/\([.0-9][.0-9]*\)[qQ]\([+-]*[0-9][0-9]*\)/\1e\2_16/g'
```

Character and Hollerith Literals

The syntax of character literal values is standard Fortran 90. Specific rules are listed under "Character Values and Literals". There is only one difference from literal syntax in SGI Fortran 77: the backslash is not supported as an escape character. The backslash in a character literal is treated as a character.

Hollerith literals are not defined in the Fortran 90 standard. (Hollerith literals were not part of the Fortran 77 standard, either). However, Hollerith literals can be used with SGI Fortran 77 and Fortran 90. Example B−8 shows the use of a Hollerith literal.

Example B−8 Hollerith Literal Use

```
integer (8) symbl
symbl = 8Hundefind
```

Note: Be aware that the use of a Hollerith literal in Fortran 90 is highly nonportable. Hollerith literals may not be supported in future versions of Fortran 90. Whenever possible, convert them to character literals stored in character variables.

The use of the Hollerith *edit descriptor* in a FORMAT statement is a separate issue. The Hollerith edit descriptor (*n*H) for FORMAT is defined in the Fortran 90 standard as an obsolescent feature. The compiler issues a warning when it processes a FORMAT statement containing a Hollerith edit code. In contrast, the use of a Hollerith literal in an expression is not defined in the standard at all, and the current compiler processes it without a message.

Differences in Pointer Data

SGI Fortran 77 supports the POINTER statement as a way of declaring two associated names: a name for a pointer, and a name for the variable addressed by the pointer. Whenever the program refers to the *addressed* variable, the compiler inserts code to load the variable's address from the associated *pointer* variable.

Only the pointer variable is defined as part of the program image. It can be treated as an integer in expressions. It is usually assigned a value (an integer that represents a memory address) from one of two sources:

- a value returned by the *malloc*() system function, pointing to a dynamic allocation of memory
- an address returned by the LOC% function, pointing to another Fortran variable

Fortran 90 supports dynamic allocation and pointer−based variables, but the syntax for these features is completely different from that of Fortran 77. One fundamental difference is that, in Fortran 90, there is only one name for an pointer−based variable. This name normally stands for the addressed variable. In the context of a special pointer−assignment statement, the name stands for the pointer that addresses the variable.

There is no straightforward, source−level change you can make to convert a program using POINTER statements to Fortran 90. One reason is that the POINTER mechanism is used for two or more distinct purposes. In order to convert a program, you must determine why it uses POINTER.

- Often the reason is simply to permit dynamic memory allocation and reallocation of variables. In this case, you can rewrite the code to use Fortran 90 allocatable variables. Pointers are not necessary; the compiler generates code to allocate memory automatically.

  For details on allocatable variables, see the *Fortran 90 Handbook*, sections 5.3.3 and 6.5.1.

- When the reason is to permit general addressing, in which one name refers to different data objects at different times—for example, to navigate through a binary tree or similar dynamic structure—then you can rewrite the code to use the Fortran 90 POINTER data attribute and pointer assignment.

  For details on pointers in Fortran 90, see the *Fortran 90 Handbook*, sections 5.4 and 7.5.3.

Occasionally the reason for POINTER use is to manipulate or interrogate general memory addresses, or to do hardware−dependent system programming. Such programs are best left in Fortran 77.

Differences in Declaration of Structures

SGI Fortran 77 compilers accepted the nonstandard STRUCTURE, UNION, and RECORD statements as a way of declaring structures composed of heterogenous data fields. These statements are not supported by Fortran 90. Most of the same purposes are served using Fortran 90 derived types.

Basic Structure Declarations

Example B−9shows a typical structure declaration and its use in Fortran 77.

Example B−9 Fortran 77 Program Using a Structure

```
structure /weather/
   integer*1 month /08/, day /10/, year /89/
   character*20 clouds
   real rainfall
end structure
record /weather/ latest
```

```
      latest.clouds = 'overcast'
      latest.rainfall = 3.12
      print *, latest.clouds, latest.rainfall
      end
```

A comparable Fortran 90 program is shown in Example B−10

Example B−10 Fortran 90 Program Using a Structure

```
type weather
   integer (1) :: month = 04, day = 18, year = 95
   character (20) clouds
   real rainfall
end type weather
type (weather) latest
latest = weather(04,18,95,'overcast',3.12)
print *, latest%clouds, latest%rainfall
end
```

The programs are fundamentally alike. Each declares a structure composed of named fields, then defines and initializes a variable holding one instance of the structure, and finally accesses the fields of the variable.

Compare Example B−9and Example B−10and note the following syntactic differences:

- The Fortran 90 structure opens with TYPE *name* instead of STRUCTURE /*name*/.
- The structure closes with END TYPE *name* instead of END STRUCTURE.
- The Fortran 90 syntax for initial field values is assignment with the equals sign (the same syntax is used to give initial values to any variable). The Fortran 77 syntax for initial field values is similar to the DATA statement.
- The Fortran 90 definition of a structure variable opens with TYPE (*name*) instead of RECORD /*name*/.
- Fortran 90 supports the use of a structure constructor to allow assigning all fields of a structure in one statement.
- In Fortran 90, you access a field using *structure%field* instead of using *structure.field.*

The syntactic differences could be converted using a tool such as *awk* or *perl*.

Equivalenced and Common Structures

There is an important semantic difference between Fortran 77 structures and Fortran 90 structures. In Fortran 77, fields within a structure are always placed in memory in the order they are declared. This is not necessarily true in Fortran 90. By default, the compiler can choose to reorder the fields in memory.

If you use the EQUIVALENCE or COMMON statement in order to set up a storage association between the fields of a structure and other data, you must specify the SEQUENCE statement within the structure declaration. This forces the compiler to keep the fields in their declared order, in memory.

Unions

SGI Fortran 77 compilers support the UNION statement as a way of declaring multiple data types for fields within a structure. The declaration in Example B−11 allows a field to be treated as either a COMPLEX or a pair of REAL values.

Example B−11 Fortran 77 UNION Declaration

```
structure /cxre/
   union
      map
         complex cx
      end map
      map
         real re, im
      end map
   end union
end structure
```

The UNION statement is not supported by Fortran 90. If it is essential to treat a field under different types at different times, you can set up a storage association through the EQUIVALENCE statement.

Differences in Intrinsic Procedures

There are differences in the intrinsic procedures available in SGI Fortran 77 and in Fortran 90. The latter supports fewer specific intrinsic functions and some Fortran 77 extended intrinsics need to be converted.

Differences in Intrinsic Functions

Both Fortran 77 and Fortran 90 support a set of intrinsic functions. Both languages distinguish between generic functions and specific functions. A generic function accept various data types as input. A specific function requires arguments of specific types. In fact, the compiler converts a call on a generic function into a call on the specific function that is appropriate for the type of argument.

In general, Fortran 90 provides a broader range of generic functions, but supports fewer specific intrinsic functions than SGI Fortran 77. The Fortran 90 standard deliberately defines few specific names.

Where a Fortran 77 program uses generic functions, no conversion is needed. Where it uses specific functions by name, it may need to be changed.

Specific Intrinsic Functions for Data Type Conversion

SGI Fortran 77 has a large number of specific functions for data type conversion. For example, the generic function INT returns the integer value of its argument. However, there are also specific functions based on the type of argument and the type of result required, for example JINT, KINT, IIDINT, KIDINT, IIQINT, and so on.

Fortran 90 supports only the generic names for conversion functions, such as INT, REAL, and CMPLX. However, each of these accepts an optional second argument that specifies the desired result type.

For example, the Fortran 77 function IIQINT(*r*), which converts REAL(16) to INTEGER(2), can be replaced by INT(*r*,KIND=2). The function QCMPLX(*r*,*i*), which converts to quad−complex, can be replaced by CMPLX(*r*,*i*,KIND=16).

Calls to the specific data conversion functions of Fortran 77 must be converted in this way, using generic functions with KIND parameters as needed.

Specific Intrinsic Functions for Transcendentals

SGI Fortran 77 supports a number of specific intrinsic names that are used to apply transcendental functions on arguments of particular data types. For example, the CQSIN function returns the sine of a quad−precision complex number.

Fortran 90 supports the generic SIN function and only two specific names, CSIN (complex) and DSIN (double precision). When you find one of the unsupported specific names used, you must determine why it was used.

- If it was used only to be specific about the data type of the argument or result, replace it with the generic name.
- If it is being passed as an argument to another function, you must either leave the module in Fortran 77, or supply a Fortran 90 function of the same name that performs that function.

Degree−Oriented Trigonometric Functions

SGI Fortran 77 supports a set of intrinsic functions to do trigonometric functions in degree values instead of radian values; for example, DCOSD returns the cosine as a double−precision value in degrees.

Fortran 90 does not supply these degree−oriented functions. If you find one used, you must convert the program from using an intrinsic to using an external function that does the same operation. Degree−oriented functions may be found at some Internet sites (see"Internet Resources for Fortran 90 Users"). Example B−12displays a skeleton for a MODULE of functions.

Example B−12 Skeleton of a Module of Trigonometric Functions

```
module degree_trig
   real(16), parameter :: &
      quadpi = 3.141592653589793238462643383279502884197Q0
   real(16), parameter :: dgr_to_rad = (quadpi/180Q0)
   intrinsic cos, sin, tan
contains
   function sind(dgr_argument)
   real(4) sind, dgr_argument
      sind = sin(dgr_to_rad * dgr_argument)
   end function
```

```
   function cosd(dgr_argument)
   real(4) cosd, dgr_argument
      cosd = cos(dgr_to_rad * dgr_argument)
   end function

   function tand(dgr_argument)
   real(4) tand, dgr_argument
      tand = tan(dgr_to_rad * dgr_argument)
   end function

   function dsind(dgr_argument)
   real(8) dsind, dgr_argument
      dsind = sin(dgr_to_rad * dgr_argument)
   end function

   function dcosd(dgr_argument)
   real(8) dcosd, dgr_argument
      dcosd = cos(dgr_to_rad * dgr_argument)
   end function

   function dtand(dgr_argument)
   real(8) dtand, dgr_argument
      dtand = tan(dgr_to_rad * dgr_argument)
   end function
end ! module
```

Tip: Some programmers find it a pleasant exercise to write their own version of CQSIN or DCOSD, but you can save time by using a version that is already written. Refer to "Internet Resources for Fortran 90 Users" for pointers to available, public−domain software libraries.

Converting Extended Intrinsic Procedures

SGI Fortran 77 supports several nonstandard intrinsic subroutines and functions. In Fortran 90, the purposes of most of these procedures are served by standard intrinsic procedures. The correspondence is summarized in Table B−2

Table B−2 Corresponding Intrinsic Procedures

Fortran 77 Intrinsic Name	Fortran 90 Intrinsic Name
DATE	DATE_AND_TIME
ERRSNS	None, see "Converting Calls to ERRSNS"
EXIT	STOP statement; also see "Converting Calls to EXIT"
IDATE	DATE_AND_TIME
MVBITS	MVBITS (same function, now standard)
RAN	RANDOM_NUMBER and RANDOM_SEED

Converting Calls to ERRSNS

The ERRSNS subroutine returns the last I/O error code and the associated unit number. There is no direct conversion for this function. The last error code alone can be retrieved in either of two ways:

- The *ierrno()* function (see the *perror*(3f) reference page) returns the most recent error code from any operation.
- The INQUIRE statement with an IOSTAT= clause returns the error code associated with a specified unit or file.

However, if the program depends on the ability to recall the unit number along with the error code, you must revise its logic.

Converting Calls to EXIT

The EXIT function terminates execution. So does the STOP statement. The difference between them is that CALL EXIT allows you to end the program with a negative integer return code. STOP accepts only positive integers or character strings.

There is no direct conversion for EXIT. However, a C function like the one shown in Example B−13can serve the same purpose.

Example B−13 Substitute for EXIT

```
extern void
exit_ (int *exitstatus)
{
   f_exit();
   exit(exitstatus ? (int)*exitstatus : 0);
}
```

Converting Calls to RAN

The Fortran 77 subroutine RAN performs two purposes: it accepts a seed value, and it returns an updated seed value which is also a pseudorandom number. Fortran 90 separates these two purposes. You call RANDOM_SEED to save, or to restart, a sequence of pseudorandom numbers. You call RANDOM_NUMBER to retrieve one or more new values from the current sequence. When porting a Fortran 77 program that uses RAN, analyze the program logic to find the point, or points, at which it changes the seed value. Convert these to calls on RANDOM_SEED.

The algorithm used by RANDOM_NUMBER is more robust than the one used by RAN (see "Implementation of RANDOM_NUMBER").

Differences in I/O Processing

The Fortran 77 run−time library supports several I/O features that are not part of Fortran 90. The features are summarized, with possible work−around schemes, in the topics that follow. InTable B−3is a list of

Fortran 77 keywords that indicate use of an unsupported I/O feature. When you find one of these words in a Fortran 77 program, use the table to find the discussion of the feature.

Table B−3 Keywords Indicating Use of Unsupported Features

Keyword or Statement Name	Feature and Topic Reference
name .EQ. "UNKNOWN"	"UNDEFINED Instead of UNKNOWN"
−*vms_endfile* (compiler option)	"VMS Endfile"
ACCEPT	"ACCEPT Statement"
ACCESS="APPEND"	"Open File for APPEND"
ACCESS="KEYED"	"Key−Indexed (ISAM) Access"
ASSOCIATEVARIABLE=*var*	"Getting the Number of the Next Record"
CARRIAGECONTROL=*type*	"Carriage Control"
DECODE	"Internal Files in Numeric Arrays"
DEFAULTFILE=*fname*	"Default Filename Prefix"
DEFINE FILE	"Open Direct File With DEFINE"
DELETE [UNIT=]*unit*	"Key−Indexed (ISAM) Access"
DISP[OSE]=*disposition*	"File Disposition"
ENCODE	"Internal Files in Numeric Arrays"
FIND ([UNIT=]*unit*...)	"Seeking a Position With FIND"
NML=*groupname*	"Namelist Data Compatibility"
OPEN...FILE=*numeric_var*	"Numeric Variable for FILE"
OPEN...FORM="BINARY"	"Special File Formats"
OPEN...FORM="SYSTEM"	"Special File Formats"
OPEN...KEY=(...)	"Key−Indexed (ISAM) Access"
OPEN...MAXREC=*n*	"Maximum Records in a Direct File"
OPEN...READONLY	"Enforcing Read−Only Access"
OPEN...RECORDSIZE=*n*	"RECORDSIZE Instead of RECL"
OPEN...SHARED	"SYNC Mode Output"
OPEN...TYPE=*statusvar*	"OPEN TYPE Instead of OPEN STATUS"
ORGANIZATION=	"Key−Indexed (ISAM) Access"
READ...KEY=*key*	"Key−Indexed (ISAM) Access"
RECORDTYPE=*rectype*	"Specifying the Record Type"
REWRITE	"Key−Indexed (ISAM) Access"
TYPE *fmt,iolist*	"TYPE Synonym for PRINT"
UNLOCK	"Key−Indexed (ISAM) Access"

ACCEPT Statement

The ACCEPT statement in Fortran 77 transfers data from the standard input unit to the items specified by the input list. Replace it in Fortran 90 with a READ from unit 5.

Carriage Control

(This topic amplifies the *Fortran 90 Handbook* topic 9.2.10.)

Carriage control is not supported in Fortran 90. Formatted output always starts in column 1, and list−directed output always starts in column 2, leaving a space character in column 1.

The Fortran 77 CARRIAGECONTROL clause can be used with OPEN to specify the type of carriage control to be used with the file. It can be specified with INQUIRE to find out the type of carriage control used by a file.

Since the clause is not supported, remove it from the OPEN statement. If the value specified was NONE and this file uses only formatted output, or if value specified was LIST and this file uses only list−directed output, no further conversion is needed—the Fortran 90 output will be the same as in Fortran 77.

Otherwise, you have to determine if the column−1 carriage control characters are really required by the programs that use the output. If so, you have to modify the WRITE and FORMAT statements used for output so as to generate the needed column−1 data.

If the CARRIAGECONTROL clause appears on an INQUIRE statement, remove it and replace it with an assignment of "UNKNOWN" to the variable.

Default Filename Prefix

The DEFAULTFILE clause of Fortran 77 OPEN establishes a prefix string that is used with the FILE clause to construct the full name of the file to be opened. When STATUS="SCRATCH"is specified, the prefix string is used with the UNIT number to construct a name for the temporary file.

The DEFAULTFILE clause can also be used with INQUIRE to retrieve the value given at OPEN.

This feature is not supported by Fortran 90. You have to revise the program to construct the filename string using character expressions, and pass the full string in the FILE clause. Store the prefix string in a global variable. If the clause appears on an INQUIRE statement, replace it with an assignment from the global variable.

Enforcing Read−Only Access

In Fortran 77, you specify that a file is to be read−only by writing the READONLY clause in the OPEN statement.

The same purpose in Fortran 90 is achieved by writing ACCESS=READ in the OPEN statement. This is a simple textual change.

File Disposition

Fortran 77 supports the use of the DISPOSE=*disposition* clause on the OPEN statement. It also allows the use of DISPOSE=*disposition* clause on a CLOSE statement to override the disposition of the file.

This clause is not supported by Fortran 90. The possible dispositions and their conversions are as follows.

KEEP, SAVE	Default disposition. Simply comment−out the clause.
PRINT	Arrange for printing in some other way, for example by calling the *system()* library function with an *lp* command specifying the filename.
SUBMIT	Arrange for execution in some other way, for example by calling the

	system() library function with a command string that pipes the file to *sh.*
PRINT/DELETE, SUBMIT/DELETE	Not supported. Handle PRINT or SUBMIT as above, and use the *unlink()* system function to delete the file.

For more on system functions, see "Support for IRIX Kernel Functions". For details on the Fortran 90 OPEN statement, see the *Fortran 90 Handbook* section 9.5.

Getting the Number of the Next Record

When using direct access, the program sometimes needs to know the record number at which the file is currently positioned. In Fortran 77 and in Fortran 90, you can get this information with the INQUIRE NEXTREC= clause.

In Fortran 77 you can use OPEN ASSOCIATEVARIABLE= to specify a variable that is automatically updated with the next record number after every access to the file. This feature is not supported in Fortran 90. You have to delete the clause from OPEN, and ensure that the specified variable is updated by INQUIRE before it is used.

Internal Files in Numeric Arrays

Fortran 77 permits storing internal files in numeric arrays, rather than in character variables as is normal for internal files. This is done using the DECODE and ENCODE statements instead of READ and WRITE.

This feature is not supported in Fortran 90. Recode the program to use READ and WRITE (the statement parameters are almost the same) and character variables.

Key−Indexed (ISAM) Access

SGI Fortran 77 supports key−indexed access to files. Key−index access is also called indexed−sequential access, or ISAM. The sign that a file is used in this way is the appearance of ACCESS="KEYED" in the OPEN statement.

Key−indexed access is not supported by Fortran 90. If this type of access is essential to the application, it is probably best to not attempt a conversion to Fortran 90.

The DELETE statement is used only with key−indexed files. It deletes the record last retrieved.

The following clauses of the INQUIRE statement are related to key−indexed access and need to be changed:

ACCESS=	Cannot return the value "KEYED"
KEYED=	Not supported, must be removed
ORGANIZATION=	Not supported, must be removed (ACCESS= returns comparable information for the supported access modes)

Maximum Records in a Direct File

The Fortran 77 OPEN statement accepts the MAXREC= clause to specify the maximum number of records in a direct−access file. This clause is not supported by Fortran 90. Remove the clause. If the program relies on the Fortran run−time library to detect creation of a record beyond the maximum size, you have to revise the program to enforce the limit using program logic.

Namelist Data Compatibility

The form of a namelist data record as supported by MIPSpro Fortran 77 is:

`{ $ | &}`*groupname* *item* `=` *value* `[,...] [/] {$ | &}[END]`

In particular, the "/" terminator is used only when input is to terminate early, and is never written by a namelist WRITE statement. Even when "/" is present, the record must be terminated by a "$" or "&" after it.

The form of a namelist data record as defined for Fortran 90 is:

`&`*groupname* *item* `=` *value* `[,...] /`

That is, the "/" terminator is required on all records; is always written; and no terminating "&" is required. These two formats are not compatible, so namelist data files prepared by or for a Fortran 77application are formally incompatible with a Fortran 90 application.

Two extensions are supported in MIPSpro Fortran 90 to relieve this incompatibility. An input namelist is allowed to start with either "&" or "$" (recognition of "$" is an extension). An input namelist is allowed to end with "/" or "$," and characters between "$" and end of record are ignored (recognition of "$" is an extension).

These extensions permit Fortran 90 to read namelist data prepared by a Fortran 77 program. This allows you to convert a Fortran 77 program to Fortran 90 and still read the data that was written by the Fortran 77 version. A Fortran 77 program cannot read namelist data written by Fortran 90.

Note: Fortran 90 cannot read namelist data written by Fortran 90 when the output contains CHARACTER data, unless the output was written with delimiters (not the default behavior). This is the standard−defined behavior.

Numeric Variable for FILE

SGI Fortran 77 permits the use of a numeric variable as the operand of the FILE= clause of the OPEN statement. (Such a numeric variable might be initialized using a Hollerith literal; see "Character and Hollerith Literals".)

The Fortran 90 standard specifies that the FILE= clause must specify a character expression.

Open Direct File With DEFINE

The DEFINE FILE statement is effectively the same as the OPEN statement with the ACCESS="DIRECT" clause. Rewrite the DEFINE FILE to use OPEN instead.

Open File for APPEND

In Fortran 77 you use the OPEN clause ACCESS="APPEND" to open a file for output at end. To do the same thing in Fortran 90, change

```
OPEN (...ACCESS="APPEND"...)
```

to read

```
OPEN (...ACCESS="SEQUENTIAL",POSITION="APPEND"...)
```

For details on the OPEN statement, refer to the *Fortran 90 Handbook* topic 9.6.

OPEN TYPE Instead of OPEN STATUS

The Fortran 77 OPEN statement permits the clause TYPE= as a synonym for the STATUS= clause. For Fortran 90 use, rewrite TYPE= as STATUS=.

RECORDSIZE Instead of RECL

Fortran 77 permits the RECORDSIZE= clause of OPEN as a synonym for the RECL= clause. Change the word RECORDSIZE to RECL for Fortran 90.

Seeking a Position With FIND

The Fortran 77 FIND statement sets the record position of a direct−access file without transferring any data. It is not supported by Fortran 90. You can achieve the same result using an unformatted, direct−access READ with an empty *iolist*.

Special File Formats

In SGI Fortran 77, the FORMAT clause of OPEN can specify two special modes:

- FORMAT="BINARY" permits reading and writing binary data from character variables.
- FORMAT="SYSTEM" allows input ignoring record boundaries.

These special modes are permitted by MIPSpro Fortran 90. However, neither is supported by the Fortran 90 standard. Either type of file access can also be achieved by using the IRIX kernel functions *read()* and *write()* (see "Support for IRIX Kernel Functions").

Specifying the Record Type

Fortran 77 accepts the RECORDTYPE= clause on the OPEN and INQUIRE statements. On OPEN, the only acceptable value is the default type for the file based on the ACCESS and FORMAT clauses. The record type is returned on INQUIRE.

The RECORDTYPE= clause is not supported by Fortran 90. The run−time support expects the same default record types as Fortran 77 (files are interchangeable between languages). Comment−out the clause on the OPEN statement. If the clause appears on the INQUIRE statement, replace it with an assignment of a constant string to the target variable.

SYNC Mode Output

The Fortran 77 OPEN statement permits the clause SHARED to request that the file be written to disk as soon as possible. This clause is not supported by the Fortran 90 standard.

You can remove the SHARED clause but get the same support using the *fcntl()* library function (see "Support for IRIX Kernel Functions") to set the F_SYNC file descriptor flag.

TYPE Synonym for PRINT

In SGI Fortran 77, you can use the word TYPE as a synonym for the word PRINT. This is not possibly in Fortran 90, where TYPE is the statement used to declare a structure (see "Differences in Declaration of Structures"). Replace TYPE with PRINT.

UNDEFINED Instead of UNKNOWN

Certain clauses of the Fortran 90 INQUIRE statement return the string UNDEFINED in situations where Fortran 77 returned the string UNKNOWN. This is true of the ACCESS, BLANK, and FORM queries (as well as DELIM and POSITION, which will not appear in a Fortran 77 program).

If the Fortran 77 program inquires these values and then compares .EQ. "UNKNOWN," it will always get a result of .FALSE.

Other INQUIRE clauses do return UNKNOWN as in Fortran 77: DIRECT, FORMATTED, SEQUENTIAL, and UNFORMATTED.

VMS Endfile

Normally the end of a sequential file is defined by the IRIX file−size information. All data in the file is part of the file. Under Fortran 77 you can pass the *−vms_endfile* option to the compiler. This causes the run−time code for formatted input to recognize a control−D character in the input stream as an end of file mark.

Fortran 90 does not support this option. You can write a program in C or *perl* to read a file and truncate it (using the system function *ftruncate()* or the perl function *truncate()*) at the length preceding the first control−D. You can write a Fortran 90 program to read a file and copy it, stopping the copy at the first control−D (Fortran 90 does not have access to the *ftruncate()* system function). Either of these methods can be used to clean up a file that uses the VMS endfile convention, so that it can be read using Fortran 90 conventions.

Glossary

alias

A second identifier that accesses the same memory location as some other identifier. In Fortran, you can explicitly create aliases using the EQUIVALENCE statement and by use of the TARGET attribute and pointer assignment. An alias also occurs when the same variable is passed to a procedure in two or more argument positions.

assertion

In Fortran work, a specially−formatted comment line that describes the source program to the compiler. For example, the line C*$* ASSERT NO EQUIVALENCE HAZARD tells the compiler that the code in the following loop never addresses the same memory location under two different, equivalenced, identifiers. (Note that the general term assertion has several other meanings in general data processing speech.)

assumed dependence

A dependence that the compiler assumes may exist, but about which it has no information. An assumed dependence can prevent the compiler from performing many kinds of optimization.

critical section

A portion of a parallel section that can be executed by only one process at a time. You use critical sections to ensure that shared static variables are updated in an orderly way.

data independence

When the value assigned to a variable within a loop does not depend on any value calculated in a different iteration of the loop, the variable has data independence. The compiler needs to verify data independence in order to parallelize a loop.

dependence

When the value of one variable depends on the value of another variable, there is a dependence between them. The compiler locates and analyzes data dependences in order to be able to optimize the program correctly.

directive

A specially−formatted comment line that directs the compiler to treat the program in a particular way. For example the line C*$*IPA enables interprocedural analysis (IPA) over the following code. See also assertion.

driver

In general, a driver is a program that controls or manages something, for example a device driver. Speaking of programming languages, a driver is the program that directs the steps of compilation and linking. The *f90* program is the compiler driver for a Fortran 90 compilation.

dynamically linked

Linked in name only, so that the executable file contains only the information needed to locate the code of a procedure—the name of the DSO that contains it and the name of the entry point. When the executable program is loaded, the DSO is also loaded, and the linkage between them is fixed in memory only.

external name

The name of a subroutine or function that is not defined in an object module, but that is called from that module. External names are recorded as strings in the object module, and can be displayed using the *nm* command.

generic function

An intrinsic function that can be called with arguments of various data types; for example, SIN can be applied to any numeric type. In a CALL, the compiler invokes the correct specific intrinsic function based on the type of arguments used. A generic function's name cannot be passed as an argument because the compiler cannot determine which specific name to pass. See specific function, intrinsic procedure.

inlining

The process of replacing a reference to a function with a copy of the function itself. This is done as an optimization, to remove the overhead of calling the function; however it expands the size of the program, which can slow the program down due to cache contention.

interprocedural analysis (IPA)

The process of inspecting the text of a function with respect to the code from which it is called, in order to get information about the relationships between the arguments and result of a function, and the variables used by the function's caller. This information can enable optimizations that are not otherwise possible.

intrinsic procedure

A function or subroutine that is defined as part of the Fortran language, for example SIN or LEN. Unlike language keywords, names of intrinsic procedures are not reserved; they can be preempted by user procedures or variables. See generic function, specific function.

loop interchange

To modify two or more nested loops exchanging the loop variables of an inner and an outer loop, in order to remove a data dependence or to reduce the stride of array indexing.

loop−invariant if

A Boolean expression, usually in an IF statement, that appears within a loop of some kind yet always has the same value when evaluated because it does not test any variables that are assigned within the loop. Such a loop and the IF that contains it can be moved outside the loop, saving time.

main module

A program unit that is compiled with the expectation that program execution will commence in that module. Only one main module can be linked into any executable.

parallel optimization

An optimization in program logic that makes the program take better advantage of multiple CPUs when they are available.

parallel section

A passage of Fortran code that can be executed in parallel by multiple, concurrent processes. A parallel section can be designated using PCF directives. A single DO loop can be made into a parallel section using the C$DOACROSS directive.

processor dependency

Any detail of compiler or run−time operation that is not defined by the Fortran 90 language standard.

public name

The name of a subroutine or function that is defined in an object module and that can be called from other, separately−compiled, modules. Public names are recorded as strings in the object module, and can be displayed using the *nm* command.

recurrence

A mathematical relationship in which the current value of an expression depends on at least one prior value ($X_t = f(X_{t-1})$). Loops that calculate recurrences are difficult to parallelize.

reduction

A mathematical operation that produces a value as a function of a set of values, for example, taking the sum of all elements of an array. When a reduction is calculated in a loop, it can be difficult to parallelize the loop.

scalar optimization

An optimization that affects the use of scalar, that is, single, variables or the use of a single CPU. Compare to parallel optimization, which affects the use of multiple CPUs.

specific function

An intrinsic function that has specific requirements as to the data types of its arguments and its result; for example, CSIN requires a COMPLEX argument. See generic function, intrinsic procedure.

statically linked

Linked as a physical part of an executable file. The linkage between calls and subprograms is completely fixed at link time. See dynamically linked.

stride

The distance in memory units between array indexes in successive iterations of a loop. The stride depends on which array index varies fastest, and the increment for the index. A stride of 1 visits successive array elements, and has the best performance in cache memory.

thread

An independently−scheduled point of execution. This term has many uses in data processing, for example "POSIX Thread" is one standardized programming interface for creating and controlling threads not yet supported by IRIX. In Fortran 90 and IRIX, a thread is implemented by a lightweight process created by the system call *sproc*().

work−sharing construct

A portion of a parallel section that you designate to be used in a particular way by concurrent processes. All concurrent processes in the parallel section synchronize at the end of the work−sharing construct.